

Microprocessortechniek MC 6809

Programming en interfacing

Ir. J.G. Rouland

Eerste druk



5 6 7 / 95 94

ISBN 90 236 0398 2 / NUGI 169

© 1987 by B.V. Uitgeverij Nijgh & Van Ditmar, Rijswijk, The Netherlands.

Alle rechten voorbehouden. Niets uit deze uitgave mag worden verveelvoudigd, opgeslagen in een geautomatiseerd gegevensbestand, of openbaar gemaakt, in enige vorm of op enige wijze, hetzij elektronisch, mechanisch, door fotokopieën, opnamen, of enig andere manier, zonder voorafgaande schriftelijke toestemming van de uitgever.

Voor zover het maken van kopieën uit deze uitgave is toegestaan op grond van artikel 16B Auteurswet 1912^j het Besluit van 20 juni 1974, St.b. 351, zoals gewijzigd bij het Besluit van 23 augustus 1985, St.b. 471 en artikel 17 Auteurswet 1912, dient men de daarvoor wettelijk verschuldigde vergoedingen te voldoen aan de Stichting Reprorecht (Postbus 882, 1180 AW Amstelveen). Voor het overnemen van (een) gedeelte(n) uit deze uitgave in bloemlezingen, readers en andere compilatiewerken (artikel 16 Auteurswet 1912) dient men zich tot de uitgever te wenden.

WOORD VOORAF

Dit boek behandelt de architectuur en de programmatuur van een microprocessor en laat zien op welke wijzen een microprocessor kan communiceren met zijn omringende bouwstenen.

Hoewel de MC6809-microprocessor uitgebreid wordt besproken, worden de bovengenoemde aspecten in zijn algemeenheid behandeld.

Bij het ontwikkelen van programma's ligt in dit boek de nadruk op de structuur en op de documentatie en niet op het besparen van geheugenruimte en/of executietijd. Vandaar dat alle programma's zijn ontwikkeld aan de hand van structogrammen.

Het boek is in eerste instantie bedoeld voor het HTO. Mogelijk kan het in het laatste leerjaar ook van nut zijn voor het MTO.

De eerste negen hoofdstukken van het boek gaan vooral over de microprocessor zelf. Aan de orde komen het ontstaan van de microprocessor, de binaire en hexadecimale talstelsels, de BCD-code en de ASCII-code, de architectuur van de microprocessor, de diverse adresseringsmethoden, de functie en het gebruik van assemblers, de instructieset van de MC6809, de organisatie van de stack, het ontwerpen van modulaire programma's, het ontwerpen van subroutines en macro's en het afhandelen van interrupts.

Bij de behandeling van de instructies wordt telkens op het nuttig gebruik van die instructies ingegaan. Niet alleen wordt uitgelegd wat de instructies precies doen, maar ook voor welke toepassingen ze gebruikt kunnen worden. Er worden vele voorbeelden gegeven, die zoveel mogelijk aansluiten bij reële situaties, zoals CV-regelingen, verkeersregelingen en beveiligingen van spoorwegovergangen.

De hoofdstukken 10 en 11 laten vooral zien hoe de microprocessor communiceert met omringende componenten. Hierbij komen serieel en parallel, synchroon en asynchroon datatransport aan de orde. Er wordt ingegaan op programmed input/output, input/output op basis van interrupts en direct memory access. Behandeld worden hier de Peripheral Interface Adapter, de Asynchronous Communications Interface Adapter, de Programmable Timer Module, de Direct Memory Access Controller en de Priority Interrupt Controller. Ook wordt aandacht besteed aan tijddiagrammen en aan het uitvoeren van timingberekeningen bij het lezen en schrijven in geheugen en in input/output.

In het laatste hoofdstuk wordt aandacht besteed aan moderne programmeertechnieken. Hierbij komen de diverse methoden van parameteroverdracht aan de orde, het gebruik van local en

global storage en het ontwerpen van positie-onafhankelijke, reentrant en recursieve routines.

Aan het einde van elk hoofdstuk is uitgebreid oefenmateriaal opgenomen in de vorm van opgaven. De uitwerkingen van deze opgaven staan achter in het boek. De oplossingen kunnen eventueel worden getest, indien men in het bezit is van een personal computer of een ontwikkelsysteem met een MC6809-microprocessor.

Veel dank ben ik verschuldigd aan mijn collega's, Ir. D.A.J. Smeets en Ing. H.G.H.J. Muileman, voor hun nuttige suggesties, en aan mijn echtgenote Mieke en onze dochters Carrie, Sonja en Emmy voor de vele uren, die zij mij ongestoord aan dit boek lieten werken.

Eerbeek, 1987

J.G. Rouland

INHOUDSOPGAVE

1 Algemene inleiding

1.1	Algemene doelstellingen	1
1.2	Specifieke doelstelling	1
1.3	Het ontstaan van de microprocessor	2
1.4	Hardware-generaties	10
1.5	Flowdiagrammen	11
1.6	Structogrammen	15
1.7	Software-generaties	17
1.8	Samenvatting	19
1.9	Opgaven	20

2 Talstelsels en codes

2.1	Specifieke doelstelling	21
2.2	Binaire talstelsel	21
2.3	Hexadecimale talstelsel	23
2.4	BCD-code	25
2.5	ASCII-code	26
2.6	Getallen optellen	27
2.7	Carry	28
2.8	Getallen aftrekken	29
2.9	Borrow	29
2.10	Negatieve getallen	29
2.11	Overflow	33
2.12	Niet-gehele getallen	36
2.13	Samenvatting	38
2.14	Opgaven	39

3 De microprocessor

3.1	Specifieke doelstelling	41
3.2	Een computertoepassing	41
3.3	Computerarchitectuur	44
3.4	Processorarchitectuur	48
3.5	Instructiecyclus	51
3.6	Programmeermodel	57
3.7	Samenvatting	60
3.8	Opgaven	61

4 Adresseringsmethoden

4.1	Specifieke doelstelling	62
4.2	Instructieformaten	62
4.3	Inherent addressing	65
4.4	Immediate addressing	66
4.5	Extended (absolute) addressing	69
4.6	Direct (page) addressing	70
4.7	Indexed addressing	72

4.8	Indirect addressing	82
4.9	Relative addressing	89
4.10	Samenvatting	93
4.11	Opgaven	94

5 Assembleren

5.1	Specifieke doelstelling	96
5.2	De functie van assemblers	96
5.3	Assembly-instructies	98
5.4	Assembler directives	106
5.5	Soorten assemblers	111
5.6	Samenvatting	113
5.7	Opgaven	114

6 Instructieset

6.1	Specifieke doelstelling	115
6.2	Inleiding	115
6.3	Datatransportinstructies	120
6.4	Logische instructies	133
6.5	Rekenkundige instructies	153
6.6	Spronginstructies	172
6.7	Samenvatting	189
6.8	Opgaven	191

7 De stack

7.1	Specifieke doelstelling	195
7.2	Organisatie van de stack	195
7.3	Lezen en schrijven op de stack	196
7.4	Stack pointers	197
7.5	Push- en pull-instructies	200
7.6	Samenvatting	205
7.7	Opgaven	206

8 Modulair programmeren

8.1	Specifieke doelstelling	207
8.2	Inleiding	207
8.3	Subroutines	210
8.4	Macro's	217
8.5	Parameteroverdracht	223
8.6	Documentatie	232
8.7	Projectaanpak	246
8.8	Samenvatting	252
8.9	Opgaven	253

9 Interrupts

9.1	Specifieke doelstelling	255
9.2	Begripsvorming	255

9.3	Interruptafhandeling	258
9.4	Meerdere interrupts	262
9.5	Polling	268
9.6	Vectoring	270
9.7	MC6809-interrupts	277
9.8	Parameteroverdracht bij ISR	302
9.9	Samenvatting	304
9.10	Opgaven	305

10 Input/Output

10.1	Specifieke doelstelling	306
10.2	Algemeen	306
10.3	Programmed I/O	309
10.4	Interrupt-I/O	313
10.5	Synchronisatie	315
10.6	Programmed I/O met de PIA	330
10.7	Interrupt-I/O met de PIA	339
10.8	Strobing met de PIA	347
10.9	Handshaking met de PIA	349
10.10	De PIA in following mode	359
10.11	Serietransport met de ACIA	362
10.12	Programmable Timer Module	376
10.13	Direct Memory Access	401
10.14	Direct Memory Access Controller	411
10.15	Samenvatting	426
10.16	Opgaven	428

11 Microprocessor interfaces

11.1	Specifieke doelstelling	431
11.2	Inleiding	431
11.3	Microprocessorsignalen	433
11.4	Priority Interrupt Controller	453
11.5	Read Only Memories	463
11.6	Read/Write Memories	466
11.7	I/O-timing	476
11.8	Samenvatting	491
11.9	Opgaven	492

12 Microprocessor interfaces

12.1	Specifieke doelstelling	493
12.2	Lokaal en globaal geheugen	493
12.3	Parameteroverdracht	506
12.4	Positie-onafhankelijk programmeren	517
12.5	Reentrant routines	532
12.6	Recursive routines	537
12.7	Samenvatting	551
12.8	Opgaven	553

Antwoorden

Appendix A	
ASCII-tabel	603
Appendix B	
Instructionset	604
Appendix C	
PIA (MC6821)	612
Appendix D	
ACIA (MC6850)	614
Appendix E	
PTM (MC6840)	616
Appendix F	
DMAC (MC6844)	619
Appendix G	
IC-pinning	622
Index	624

1 ALGEMENE INLEIDING

1.1 Algemene doelstellingen

Microprocessoren zijn heden ten dage niet meer weg te denken uit de meeste elektrotechnische apparaten. Elektrotechnici dienen dan ook inzicht te hebben in de werking en de toepasbaarheid van microprocessoren en moeten deze componenten kunnen gebruiken als programmeerbare digitale bouwstenen. Microprocessoren vinden uitgebreide toepassing bij allerlei besturingen en regelingen, in apparaten en systemen waarbij grote aantallen (meet)gegevens worden verwerkt, en in digitale computers. Vandaar dat het ook voor meet-, regel- en besturingstechnici van groot belang is om vertrouwd te zijn met het gebruik van microprocessoren.

Na het bestuderen van dit boek kent u de werking van microprocessoren en kunt u deze toepassen bij besturingen en regelingen en bij het verwerken van gegevens. U zult in staat zijn microprocessoren via diverse input/outputschakelingen te laten communiceren met hun omgeving. Verder hebt u een goed inzicht gekregen in de werking van digitale computers.

1.2 Specifieke doelstelling

Na het bestuderen van dit eerste hoofdstuk bezit u de volgende volgende kennis:

- * U weet hoe de microprocessor is ontstaan vanuit de digitale techniek.
- * U kent de functie van een microprocessor.
- * Het is u duidelijk wat bedoeld wordt met hardware en software.
- * U kent de betekenis van de termen geheugen, input/output, bus, computer en programma.
- * U weet wat er met de diverse generaties computerhardware wordt bedoeld.
- * De basisstructuren van flowdiagrammen kent u, zodat u deze diagrammen kunt lezen en begrijpen.
- * U kent de basisstructuren van structogrammen, zodat u deze later kunt toepassen bij het ontwerpen van programma's.
- * De voordelen en nadelen van flowdiagrammen en structogrammen zijn u bekend.
- * U weet wat er met gestructureerd programmeren wordt bedoeld

- en wat top-down en bottom-up benaderingen zijn.
- * U kent de begrippen object code, source code, hexadecimal loader, assembly, assembler, hogere taal, compiler, interpreter, niet-procedurele taal en program generator.
- * U weet wat er wordt verstaan onder de verschillende generaties computersoftware.

1.3 Het ontstaan van de microprocessor

1.3.1 Inleiding

Met behulp van de eenvoudige digitale poortschakelingen (zoals AND's, OR's en inverters) en flipflops kunnen complexere bouwstenen worden gerealiseerd, bijvoorbeeld tellers en vergelijkerschakelingen. Met al deze componenten kunnen zeer grote en ingewikkelde digitale schakelingen worden ontworpen. Zulke schakelingen kenmerken zich doorgaans door een enorme hoeveelheid verbindingen tussen het grote aantal toegepaste componenten.

Door het toepassen van Programmable Logic Arrays (PLA's) en Read Only Memories (ROM's) kan hierin wat meer structuur worden gebracht en wordt het aantal verbindingen en componenten drastisch verminderd.

Het grootste probleem bij het ontwerpen van complexe digitale schakelingen is te ontdekken welke logische functies gerealiseerd moeten worden. De volgende paragraaf laat een methode zien om de gewenste functies in kaart te brengen.

1.3.2 Specifieke oplossing voor een probleem

Stel dat u een (eenvoudige) digitale regeling moet ontwerpen voor een centrale verwarmingsinstallatie. U moet een schakeling bouwen die ervoor zorgt, dat de kamer op de gewenste temperatuur blijft. De gewenste temperatuur wordt door de bewoner ingesteld op de thermostaat. Met een temperatuuropnemer wordt de kamertemperatuur gemeten. Een digitale comparator vergelijkt nu de ingestelde temperatuur met de werkelijke temperatuur. Hiertoe moeten beide temperaturen eerst worden omgezet in digitale waarden met behulp van een Analooog-Digitaal-Converter (ADC). Een digitale comparator kan immers alleen digitale signalen verwerken.

Is de kamertemperatuur lager dan de ingestelde temperatuur, dan zal de hoofdbrander moeten ontstoken. Dit gebeurt door een gasklep te openen. De waakvlam zorgt dan voor het ontsteken van het vrijkomende gas. Is de kamertemperatuur hoger dan de

ingestelde waarde, dan moet de gasklep worden gesloten. Hierboven beschreven we de meest elementaire regeling. Om te voorkomen dat de gasklep zeer vaak open en dicht gaat, bouwen we een vertraging in (van bijvoorbeeld 2 minuten). Na het openen of sluiten van de gasklep wordt er 2 minuten gewacht, voordat de volgende temperatuurvergelijking plaatsvindt. Pas daarna kan de gasklep opnieuw worden geopend of gesloten.

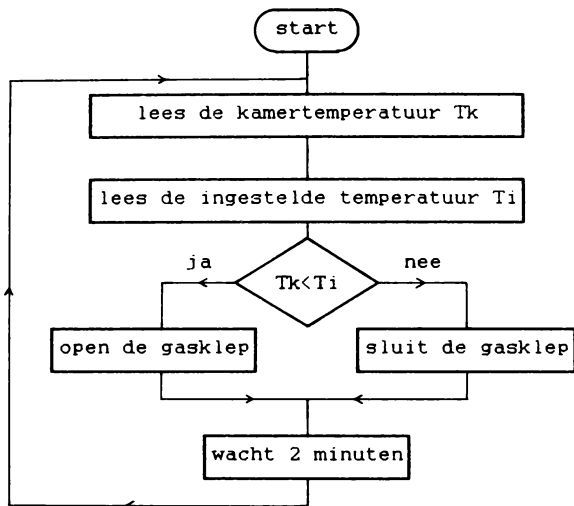


Fig. 1.1 Flowchart van CV-regelaar

Alvorens aan het ontwerp te beginnen, gaan we eerst de gewenste functie op een overzichtelijke wijze weergeven. We kunnen dit bijvoorbeeld doen met behulp van een zogenaamd stroomschema of flowdiagram (ook wel flowchart genoemd). In zo'n flowdiagram geven we in gewone taal aan wat er achtereenvolgens dient te gebeuren. We bekommeren ons hierbij nog helemaal niet om de realisatie.

Fig. 1.1 geeft het flowdiagram van de CV-regelaar, zoals die door u ontworpen dient te worden.

Een flowdiagram wordt van boven naar beneden en van links naar rechts doorlopen. Indien een andere richting wordt gewenst, moet dit worden aangegeven met behulp van pijlen.

Tijdens het doorlopen van het flowdiagram van fig. 1.1 van boven naar beneden en het realiseren van de gepasseerde blokken, ontstaat het schema van fig. 1.2 van links naar rechts. In fig. 1.2 ziet u dat de kamertemperatuur T_k wordt gemeten met behulp van een temperatuuropmeter (ook wel temperatuursensor genoemd). Deze zet de temperatuur T_k om in een spanning U_k .

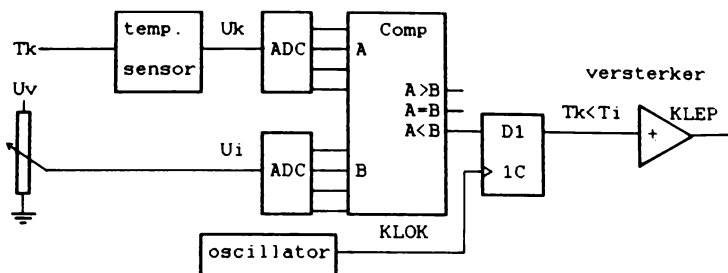


Fig. 1.2 Schema van CV-regelaar

De gewenste temperatuur T_i wordt ingesteld met behulp van een potentiometer (een weerstand met een instelbare aftakking). Hoe hoger de aftakking wordt ingesteld, des te hoger is de ingestelde spanning U_i . Deze spanning U_i is een maat voor de ingestelde temperatuur T_i . U_k en U_i worden met behulp van ADC's omgezet in digitale signalen en aangeboden aan een digitale comparator. Deze comparator vergelijkt de waarden van U_k en U_i (en dus die van T_k en T_i) met elkaar. Als de $A < B$ -uitgang van de comparator gelijk is aan 1, betekent dit dat A kleiner is dan B , ofwel $T_k < T_i$.

Op elke positieve flank van de oscillatoruitgang KLOK wordt het signaal $A < B$ ingelezen in een flipflop. De uitgang van deze flipflop wordt via een versterker verbonden met de elektronica die de gasklep aanstuurt. Als het signaal $T_k < T_i$ gelijk is aan 1, zal het signaal KLEP hoog worden en de gasklep open gaan. De hoofdbrander gaat nu aan, zodat het water in de CV-ketel wordt opgewarmd. De kamertemperatuur is immers te laag. Heeft de kamer de gewenste temperatuur bereikt ($T_k > T_i$ of $T_k = T_i$), dan is de $A < B$ -uitgang gelijk aan 0 en wordt het signaal $T_k < T_i$ eveneens gelijk aan 0 op de volgende positieve flank van KLOK. Hierdoor wordt het signaal KLEP laag, zodat de gasklep dicht gaat en de hoofdbrander dooft.

Merk op dat het signaal KLEP alleen kan veranderen tijdens de positieve flank van KLOK. Door nu de tijd tussen de opgaande flanken van KLOK gelijk aan 2 minuten te maken, gebeurt het openen en sluiten van de gasklep slechts een keer per 2 minuten. Hierdoor wordt de vertraging van 2 minuten gerealiseerd. We kiezen derhalve een oscillator met een frequentie van 1/120 Herz.

De hier gegeven oplossing is erg specifiek. Hij realiseert exact de gevraagde functies, maar niet meer. Met veranderingen en uitbreidingen is geen rekening gehouden bij het ontwerp.

Blijkt na enige tijd dat het beter is de gasklep slechts één keer per 3 minuten van stand te laten veranderen, dan moet er een andere oscillator worden gekozen. Indien deze de

aansluitingen op precies dezelfde plaats heeft als de oude oscillator, dan hoeven we deze slechts te vervangen. Zijn de aansluitingen echter niet exact gelijk, dan zullen we ook de printbedrading moeten wijzigen en dus een nieuwe printplaat moeten maken.

Willen we later energie besparen door de temperatuur van het water in de radiatoren zo laag mogelijk te houden, dan wordt de zaak complexer en moet vrijwel het gehele ontwerp opnieuw worden gemaakt. Er moet dan immers een sensor voor de buitentemperatuur en een sensor voor de watertemperatuur worden aangebracht. Hoe hoger de buitentemperatuur is, des te lager kan de watertemperatuur zijn. Een digitale schakeling moet volgens een bepaalde functie de juiste watertemperatuur (behorende bij de gemeten buitentemperatuur) berekenen. Voorts moet de brander doven als de watertemperatuur hoger wordt dan de gewenste watertemperatuur (bij de heersende buitentemperatuur).

Bovendien is het verband tussen de watertemperatuur en de buitentemperatuur afhankelijk van een aantal factoren, zoals de grootte van de kamer en de mate van isolatie.

We concluderen hieruit dat een bepaalde functie nogal eens aan verandering onderhevig kan zijn, doordat er aanvullende wensen ontstaan of doordat de omstandigheden wijzigen. Dit zou betekenen dat in ons voorbeeld van een digitale CV-regelaar een groot aantal verschillende schakelingen nodig zijn. Bovendien zouden deze waarschijnlijk regelmatig moeten worden aangepast of uitgebreid.

Voor het realiseren van een geheel andere functie, bijvoorbeeld een verkeersregeling, zou een vrijwel volledig andere digitale schakeling nodig zijn.

De soort signaalbewerkingen zijn echter in vrij hoge mate aan elkaar gelijk. Bij een verkeersregeling hebben we niet te maken met temperatuuropnemers, maar met verkeersopnemers (bijvoorbeeld in de vorm van magnetische lussen in het wegdek, die een elektrisch signaal afgeven, dat groter is naarmate zich meer auto's boven de lus bevinden). De signalen van deze sensoren moeten ook weer via ADC's worden omgezet in digitale signalen. Deze moeten dan met bepaalde tussenpozen worden ingelezen en met elkaar worden vergeleken om te bepalen welke straat groen licht krijgt en hoe lang. In dit voorbeeld moeten de signalen worden gestuurd naar de verkeerslichten in plaats van naar een gasklep.

1.3.3 Universele oplossing voor een probleem

Microprocessor

Zouden we een schakeling ontwerpen, die een aantal standaardbewerkingen kan uitvoeren, dan zou zulk een schakeling

voor een groot aantal verschillende ontwerpen kunnen worden toegepast.

De standaardbewerkingen zijn:

- gegevens binnenhalen
- onthouden
- rekenen
- vergelijken
- besturen

We moeten de schakeling meedelen in welke volgorde de verschillende bewerkingen moeten worden uitgevoerd. Schakelingen die deze standaardbewerkingen kunnen uitvoeren, zijn er tegenwoordig. Ze worden processoren genoemd. Processor is verwerker. Er worden immers signalen verwerkt. Denk bijvoorbeeld aan het binnenhalen, vergelijken en wegsturen van signalen. Men spreekt van microprocessors als de genoemde processoren zijn geïntegreerd binnen één chip (een zeer klein stukje halfgeleidermateriaal). Een geïntegreerde schakeling heet IC (Integrated Circuit).

We moeten zo'n microprocessor meedelen welke bewerkingen hij moet verrichten en in welke volgorde. Dit noemen we programmeren. Het programma slaan we op in een geheugen. Een programma bestaat uit een rij instructies. Een instructie is een opdracht om een bepaalde bewerking uit te voeren. Wijzigen nu de eisen die we aan de CV-regelaar stellen, dan kan de schakeling (de hardware) onveranderd blijven. We hoeven alleen het programma (de software) aan te passen, mits er voldoende aansluitingen zijn ten behoeve van sensoren en stuursignalen. We kunnen dezelfde hardware bovendien voor een geheel ander probleem (bijvoorbeeld een verkeersregeling) toepassen.

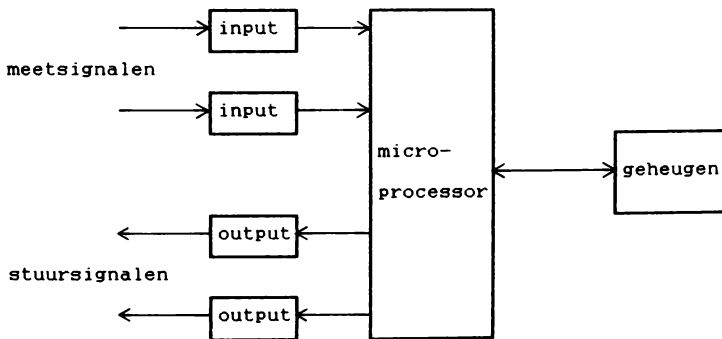


Fig. 1.3 Microprocessor als regelaar

Fig. 1.3 laat zien hoe een microprocessor kan worden gebruikt als regelaar. De microprocesor haalt de instructies, die hij moet uitvoeren, uit het geheugen. Voor het nemen van de juiste

beslissingen bij het besturen van een proces heeft de microprocessor de waarden van een aantal meetsignalen (bijvoorbeeld temperaturen) nodig. Deze gegevens worden binnengehaald via speciale invoerelektronica, de zogenaamde inputschakeling. De stuursignalen (voor bijvoorbeeld kleppen, motoren en lampen) worden via speciale uitvoerelektronica, de zogenaamde outputschakeling, naar buiten gestuurd.

Serieel en parallel datatransport

In fig. 1.3 is het geheugen en elke input- en outputschakeling via aparte verbindingen met de microprocessor verbonden. Per verbinding is slechts één draad getekend. Een analogo signaal wordt echter met behulp van een ADC vertaald in een aantal digitale signalen (bijvoorbeeld 8). Zo'n groepje bij elkaar behorende digitale signalen kan via één draad naar de microprocessor worden verzonden. De signalen moeten dan na elkaar worden overgestuurd. Men spreekt in dit geval van serietransport. Het voordeel hiervan is het geringe aantal draden. Daartegenover staat het grote nadeel van de lange duur van het transport. Immers om een gegeven over te zenden, moeten er telkens een aantal (bijvoorbeeld 8) digitale signalen na elkaar over dezelfde draad worden verstuurd. Om snel te werken, kunnen microprocessoren een groepje digitale signalen tegelijk binnenhalen of verzenden. Men spreekt hier van paralleltransport (zie ook fig. 1.4). Geheugen, input- en outputschakelingen worden dicht bij de processor geplaatst, opdat de verbindingen kort zijn. Het datatransport is dan snel en goedkoop.

De input- en outputschakelingen zijn meestal gecombineerd in één component. Zo'n schakeling wordt input/outputschakeling genoemd en kortweg aangeduid met I/O.

Bij een I/O-schakeling kunnen, naar keuze, een aantal signalen als input en een aantal signalen als output fungeren. Hierdoor kan de microprocessor via één type bouwsteen met een groot aantal verschillende digitale schakelingen communiceren.

Sterverbinding

Elke I/O-schakeling via verschillende groepjes draden met de microprocessor te verbinden (in een soort sterconfiguratie), heeft het gevolg dat de microprocessor een groot aantal in/uitgangen moet hebben. Het aantal schakelingen waarmee een processor kan communiceren, is dan beperkt.

Busverbinding

Om het aantal aansluitingen van de microprocessor te beperken, worden het geheugen en alle I/O-schakelingen met dezelfde in/uitgangen van de processor verbonden (zie fig. 1.4). De signalen worden via dezelfde groep draden getransporteerd. Microprocessor, geheugen en input/outputschakelingen zijn met elkaar verbonden via die draden. Men spreekt hier van een bus.

Zo'n processorbus is te vergelijken met een autobus. Reizigers kunnen op diverse plaatsen (halten) langs het bustraject instappen of uitstappen.

Op soortgelijke wijze kunnen de signalen van de microprocessor gebruik maken van de processorbus om naar het geheugen of naar een van de I/O-schakelingen te gaan. Omgekeerd kunnen de signalen van de I/O of van het geheugen de microprocessor via de bus bereiken.

Behalve besparing van draden heeft een busconfiguratie het voordeel dat het systeem modulair van opbouw is. Dit wil zeggen dat er op eenvoudige wijze een apparaat (module) kan worden bijgeschakeld door dit via een extra I/O-schakeling aan de bus te koppelen. Het systeem is dus gemakkelijk uitbreidbaar. In principe kunnen op deze manier een onbeperkt aantal I/O-schakelingen met de processor worden verbonden.

De processor communiceert via dezelfde groep in/uitgangen met diverse schakelingen. Deze schakelingen moeten kunnen onderscheiden met wie de processor gegevens wil uitwisselen. Hiertoe krijgt elke I/O-schakeling en elke geheugenschakeling een (verschillend) adres. Door het adres van de gewenste schakeling via de bus te verzenden, deelt de processor mee met wie hij wil communiceren. Gebeurt dit via de buslijnen waarover ook de gegevens vervoerd worden, dan moet eerst het adres en daarna de data worden verzonden. Er wordt in dit geval gesproken van een "multiplexed" adres/databus. De meeste processoren hebben een gescheiden adresbus en databus. Verder is er nog een controlbus voor besturingssignalen. Denk hierbij onder andere aan timingsignalen. De schakelingen moeten weten wanneer adres en data geldig zijn.

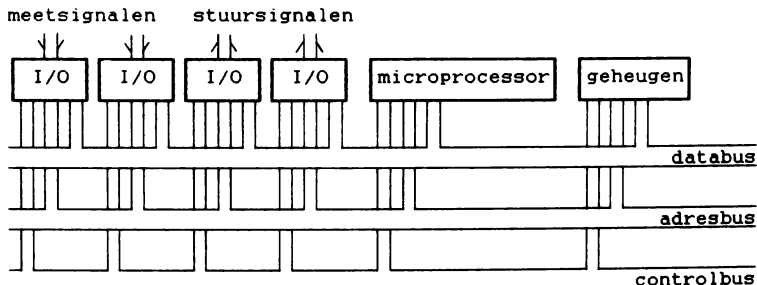


Fig. 1.4 Busconfiguratie

Fig. 1.5 laat zien hoe de processor in een busconfiguratie communiceert met de diverse schakelingen in het geval van een CV-installatie.

De instructies, die de processor nodig heeft voor het regelen van het verwarmingsproces, worden uit het geheugen gehaald. De (analoge) meetsignalen Tk, Tb en Tw (kamer-, buiten- en

watertemperatuur) en de (analoge) waarde van de ingestelde temperatuur T_i moeten eerst met behulp van ADC's worden omgezet in digitale signalen. De processor kan alleen digitale signalen verwerken. Deze digitale ingangssignalen worden via de I/O door de processor binnengehaald. Het uitgangssignaal KLEP wordt via de databus naar een I/O-schakeling gestuurd. Veelal moet dit signaal nog worden versterkt, omdat de digitale elektronische bouwstenen slechts weinig vermogen kunnen leveren.

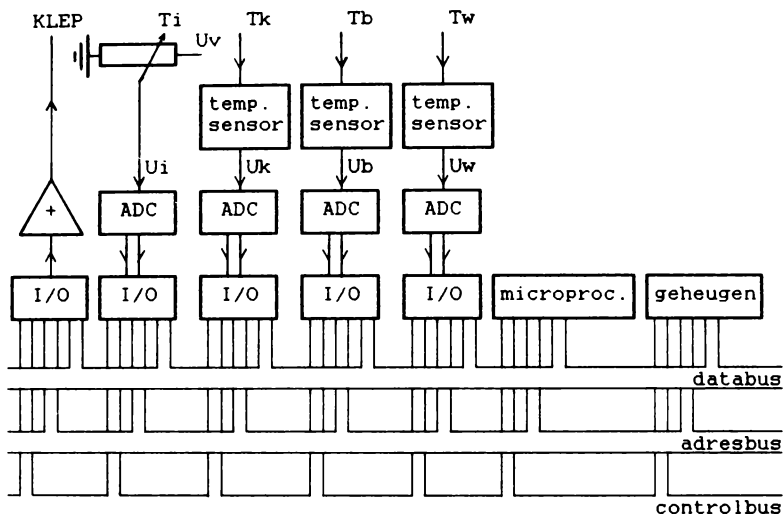


Fig. 1.5 Microprocessor als CV-regelaar in een busconfiguratie

Databus

De databus dient voor het transport van de gegevens tussen de processor en het geheugen (voor opslag van instructies en gegevens) en tussen de processor en de I/O (voor het transport van meetgegevens en stuursignalen).

Adresbus

De adresbus dient voor het vervoer van het adres. De microprocessor moet immers meedelen met wie hij wil communiceren (bijvoorbeeld met het geheugen voor het ophalen van een nieuwe instructie of met een van de I/O-circuits voor het lezen van de buitentemperatuur T_b , de watertemperatuur T_w , enzovoorts of voor het wegsturen van het signaal KLEP).

Controlbus

De controlbus wordt gebruikt voor het verzenden van een aantal besturingssignalen. Er moet onder andere aan geheugen en I/O worden verteld of de data weggestuurd dan wel binnengehaald

moet worden. Verder moet er een kloksignaal zijn om te zorgen dat alles op het juiste moment gebeurt. Zo zijn er nog een aantal besturingssignalen waarop we hier niet verder ingaan. Ze komen in hoofdstuk 11 uitgebreid aan de orde.

Gebruiken we dezelfde hardware voor een verkeersregelaar, dan hoeven we slechts de verkeerssensoren en de verkeerslichten in plaats van de temperatuuropnemers en het signaal KLEP met de I/O-circuits te verbinden. Natuurlijk moeten we een ander programma in het geheugen laden.

Computer

Processor, geheugen en I/O worden samen computer genoemd (zie fig. 1.6). Een microcomputer is een computer waarvan de processor een microprocessor is. Bevindt de microcomputer zich op één chip, dan spreekt men van een single-chip computer. Een single-board computer bevindt zich op meerdere chips, maar wel op één enkele kaart.

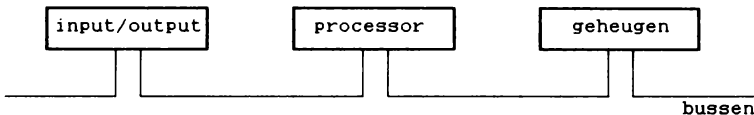


Fig. 1.6 Computer

1.4 Hardware-generaties

De eerste computers werden met behulp van relais gebouwd (in 1943). Ze waren zeer groot en energieverblindend. Aan deze machines was veel onderhoud nodig.

In 1943 werden ook reeds de eerste elektronische computers gebouwd. Ze werden opgebouwd met behulp van elektronenbuizen en worden gerekend tot de eerste-generatie computers. Vanuit technologisch standpunt gezien zouden de elektromechanische computers tot de eerste-generatie gerekend moeten zijn. Dit is echter niet gebeurd. We kunnen deze computers derhalve alleen met nulde-generatie aanduiden.

De buizencomputers gebruikten veel energie en namen enkele tientallen kubieke meters ruimte in beslag. Bovendien waren ze erg onbetrouwbaar.

Na de intrede van de transistoren (in 1948) werden de computers (vanaf 1954) met behulp van deze componenten gebouwd en spraken van tweede-generatie computerhardware. Deze computers gebruikten veel minder ruimte en energie en waren veel sneller en betrouwbaarder dan de buizenmachines.

Vervolgens werden de transistoren in grotere aantallen geïntegreerd op één chip. Hiermee was er sprake van een nieuwe hardware-technologie. Zo ontstond in het begin van de jaren zeventig de derde-generatie computerhardware.

Intussen ontwikkelde Intel in 1971 een microprocessor, een processor die op één enkele chip geïntegreerd was. Hiermee zijn we aangekomen in het tijdperk van de VLSI's (Very Large Scale Integration). Er werden steeds grotere geheugens (van honderdduizenden elementen) en complexere I/O-schakelingen op één chip geïntegreerd. Computers die van deze bouwstenen gebruik maken, behoren tot de vierde-generatie computerhardware.

Door de miniaturisering zijn de computers steeds kleiner en lichter geworden. Ten gevolge van de kortere verbindingen en de snellere schakelingen werden ze steeds sneller. Door het kleiner wordende aantal componenten en het afnemende aantal verbindingen, alsmede ten gevolge van het lagere energieverbruik en derhalve lagere dissipatie (warmte-ontwikkeling), nam de betrouwbaarheid sterk toe. Er bestaan tegenwoordig ook parallelle computers. Dit zijn computers waarin een aantal (enkele tot enkele tienduizenden) microprocessoren tegelijk aan het werk zijn.

Tegenwoordig is men druk doende met het onderzoek naar de zogenaamde vijfde-generatie computers. Of hierin nieuwe hardwaretechnologieën zullen worden toegepast, is nog niet duidelijk. De vernieuwingen zullen in ieder geval liggen op het gebied van de software en de toepassingsmogelijkheden.

Technologisch onderzoek geschiedt op het gebied van Josephsonschakelaars (die met suprageleiding werken), lichtschakelaars, lichtgeleiders en biochips. Lichtgeleiders (glasvezels) worden reeds vrij veel toegepast in de communicatietechniek.

1.5 Flowdiagrammen

In paragraaf 1.3 hebt u gezien hoe een microprocessor als standaardbouwsteen kan worden toegepast in zeer uiteenlopende digitale schakelingen.

In deze paragraaf zult u zien hoe de microprocessor kan worden geprogrammeerd voor de functie van CV-regelaar. We maken gebruik van de instructies van de MC6809-microprocessor. Deze instructies worden in hoofdstuk 6 uitgebreid besproken. We geven hier slechts een korte verklaring. Instructies worden met hoofdletters geschreven.

Het ontwerpen van een programma kan, net als het ontwerpen van een schakeling, geschieden aan de hand van een flowdiagram. Wederom wordt het flowdiagram hierbij van boven naar beneden doorlopen. Nu wordt elk blok, dat we passeren, vervangen door een of meerdere instructies. Zo ontstaat (aan de hand van het flowdiagram van fig. 1.1) het onderstaande programma voor de

START	LDA	TK	lees de kamertemperatuur Tk
	CMPA	TI	vergelijk Tk met de ingestelde
*			temperatuur Ti
*	BGE	SLUIT	spring als Tk groter of gelijk
			is aan Ti
	LDB	#1	open de
	STB	KLEP	gasklep
	BRA	WACHT	ga verder naar WACHT
SLUIT	LDB	#0	sluit de
	STB	KLEP	gasklep
WACHT	BSR	W2MIN	spring naar een subroutine,
*			waarin 2 minuten wordt gewacht
	BRA	START	ga naar START

Verklaring van de instructies:

LDA = Load A; A en B zijn registers in de microprocessor;
 LDB = Load B; laad A, respectievelijk B met ...
 STB = Store B; berg de inhoud van B op in ...
 CMPA = CoMPare A; vergelijk de inhoud van A met ...
 BGE = Branch if Greater or Equal; spring bij groter of gelijk
 naar ...
 BRA = BRanch Always; ga naar ...
 (branch = aftakking, zie fig. 1.1)
 BSR = Branch to SubRoutine; voer subroutine ... uit
 * = regel die alleen commentaar bevat, moet beginnen met *

We gaan nu iets dieper in op de flowcharts.

Stroomschema's hebben drie basisstructuren:

1. Volgorde: Actie B wordt uitgevoerd na actie A
 (zie fig. 1.7).
2. Selectie: Actie F wordt uitgevoerd als de conditie false
 (of keuze) (onwaar) is, en actie T als de conditie true
 (waar) is. Men spreekt hier van een enkelvoudige
 selectie (zie fig. 1.8).
 Bij een meervoudige selectie wordt actie i
 uitgevoerd als de conditie de waarde i heeft.
 Men spreekt hier ook wel van een case-structuur
 (zie fig. 1.9).
3. Herhaling: Een herhaling wordt bereikt door terug te
 (lus of springen in het flowdiagram. Indien men deze
 iteratie) herhaling een beperkt aantal keer wil laten
 uitvoeren, moet er in de herhalingslus een
 selectie voorkomen. Er moet een stopconditie
 zijn (bijvoorbeeld een tellerstand).
 Er zijn twee basisstructuren van een herhaling:
 a. een do-while-structuur:
 Hierbij wordt eerst de conditie getest.
 Indien deze true is, wordt actie T uitgevoerd
 (zie fig. 1.10). Actie T wordt 0, 1 of
 meerdere keren uitgevoerd.

- b. een repeat-until-structuur:
Eerst wordt actie F uitgevoerd. Vervolgens wordt de conditie getest. Zolang deze false is, wordt actie F herhaald (zie fig. 1.11). Actie F wordt dus tenminste één keer uitgevoerd.

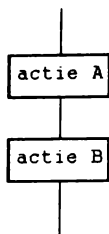


Fig. 1.7 Volgorde

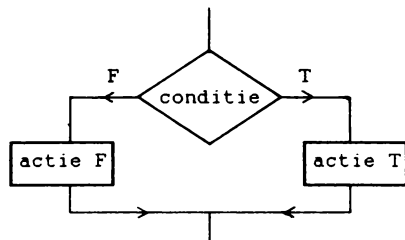


Fig. 1.8 Enkelvoudige selectie

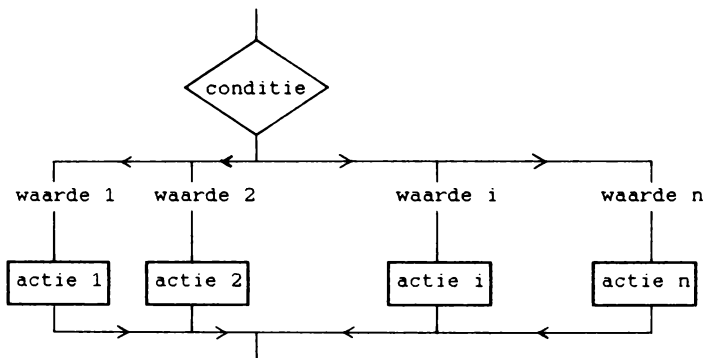


Fig. 1.9 Meervoudige selectie

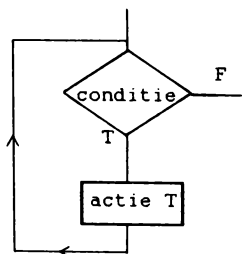


Fig. 1.10 Do-while-structuur

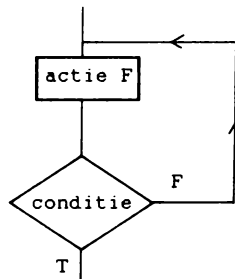


Fig. 1.11 Repeat-until-structuur

Het gebruik van de symbolen van stroomdiagrammen en hun betekenis ligt vast in de beschrijving van NEN 3283.

Een probleem bij het gebruik van flowdiagrammen is dat hierbij gemakkelijk ongestructureerde programma's kunnen ontstaan. Men laat zich gemakkelijk verleiden tot het veelvuldig gebruiken van sprongen. Er ontstaat dan een wirwar van lijnen (spagettistructuur) en een programma dat vrijwel onleesbaar is. Een dergelijk programma testen of aanpassen aan gewijzigde omstandigheden is praktisch ondoenlijk. Fig. 1.12 toont een dergelijk ongestructureerd programma, ook wel spagettiprogramma genoemd.

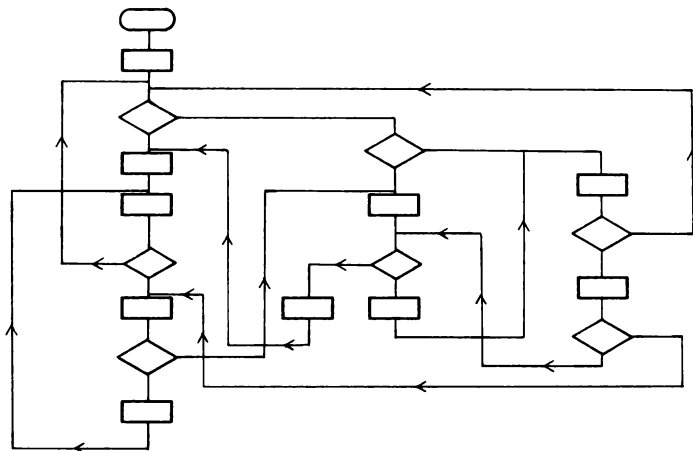


Fig. 1.12 Spagettiprogramma

Het zal u uit fig. 1.12 duidelijk zijn, waarom een dergelijk ongestructureerd programma ook wel spagettiprogramma wordt genoemd.

In de beginperiode van de software werden veel van dit soort programma's geschreven. Dit kwam vooral omdat men op deze wijze programma's kan ontwerpen, die een minimum aan geheugenruimte nodig hebben. Geheugens waren in die tijd erg duur. Tegenwoordig zijn geheugens echter zeer goedkoop. Daartegenover zijn manuren erg duur. Dit betekent dat het zeer belangrijk is overzichtelijke programma's te ontwerpen. Dergelijke programma's zijn immers aanzienlijk gemakkelijker te testen en te onderhouden (aan te passen aan gewijzigde functionele specificaties). Het is dus van het grootste belang goed gestructureerde programma's te maken. Om de leesbaarheid nog extra te vergroten, moeten deze programma's zijn voorzien van voldoende commentaar. Natuurlijk is het best mogelijk om uitgaande van flowdiagrammen gestructureerd te programmeren. Men moet er zich echter zeer bewust op toeleggen en het gebruik van sprongen zoveel mogelijk vermijden.

1.6 Structogrammen

Een uiterst handig hulpmiddel om tot een gestructureerd programma te komen, is het Nassi-Shneidermann-diagram, ook wel programmastructuur-diagram (PSD) of structogram genoemd. Gebruiken we een structogram om de functie van het te ontwerpen programma vast te leggen, dan is het onmogelijk te komen tot een spaghettiprogramma. Een structogram laat alleen gesloten blokken toe. Deze hebben slechts één ingang en één uitgang. Men kan niet uit deze blokken wegspringen of van elders midden in zo'n blok springen.

Fig. 1.13 toont het structogram van de CV-regelaar. Vergelijk dit met het flowdiagram van fig. 1.1, opdat u de overeenkomst ziet.

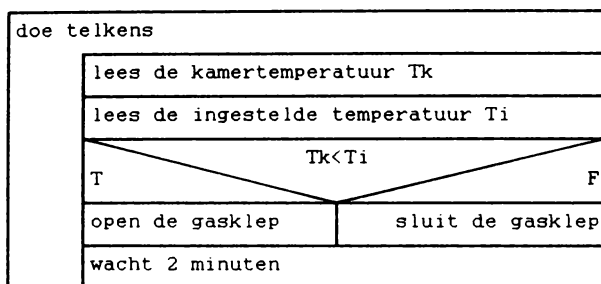


Fig. 1.13 Structogram van CV-regelaar

Ook het structogram kent de basisstructuren volgorde, selectie en herhaling. De beschrijving van de symbolen van structogrammen en de norm voor de notatie van de basisstructuren kunt u vinden in NEN 1422.

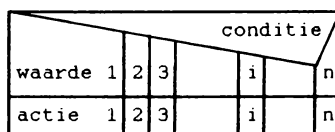
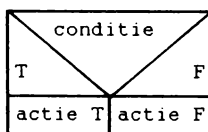
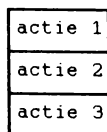


Fig. 1.14 Volgorde

Fig. 1.15 Enkelvoudige selectie

Fig. 1.16 Meervoudige selectie

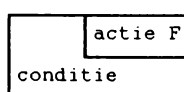
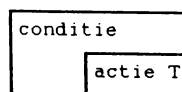


Fig. 1.17 Do-while-structuur

Fig. 1.18 Repeat-until-structuur

Structogrammen worden altijd van boven naar beneden doorlopen. Eerst komt actie 1, daarna actie 2 en vervolgens actie 3 (zie fig. 1.14).

Actie T wordt uitgevoerd als de conditie true is, en actie F als de conditie false is (zie fig. 1.15). Bij de meervoudige selectie wordt actie i uitgevoerd indien de conditie de waarde i heeft (zie fig. 1.16).

Bij de do-while-structuur wordt actie T uitgevoerd, zolang de conditie true is (zie fig. 1.17). Bij de repeat-until-structuur wordt actie F uitgevoerd, totdat de conditie true is, dus zolang de conditie false is (zie fig. 1.18).

De lus in de do-while-structuur kan 0 of meer keren worden doorlopen. Indien de lus tenminste één keer doorlopen wordt, bestaat het gevaar dat hij oneindig vaak wordt uitgevoerd. Een lus kan alleen onderbroken worden, indien een van de acties binnen die lus de waarde van de conditie wijzigt. Alleen dan kan de conditie van false naar true gaan (of omgekeerd).

Een methode om gestructureerde programma's nog overzichtelijker te maken, is het zogenaamde modulaire programmeren. Dit betekent dat men zijn programma opbouwt uit een aantal modules. Dit zijn programmadelen, die min of meer een afzonderlijke eenheid vormen. Elke module wordt apart ontworpen en getest. We kunnen hierbij twee programmeringstechnieken onderscheiden. De eerste is de top-down benadering. Hierbij bepaalt men eerst de algemene probleemstructuur. Het probleem wordt vervolgens verdeeld in deelproblemen. Deze deelproblemen worden daarna weer verder gedetailleerd in nog kleinere problemen. Dit gaat zolang door totdat de omvang van de problemen zodanig is, dat ze vrij gemakkelijk in een programma zijn te vertalen. De tweede methode is de bottom-up benadering. Deze werkt juist omgekeerd. Hier ontwikkelt men eerst modules voor deelproblemen. Deze modules worden op een hoger niveau met elkaar geïntegreerd, totdat men uiteindelijk de oplossing voor zijn probleem heeft gevonden.

In het algemeen verdient het de voorkeur een programma volgens de top-down aanpak te ontwerpen en het volgens de bottom-up methode uit te testen.

In hoofdstuk 8 wordt dieper op het modulaire programmeren ingegaan.

Structogrammen leiden meestal tot programma's die meer geheugenruimte in beslag nemen en die een grotere executietijd hebben. Het niet-gestructureerde gebruik van flowdiagrammen geeft grotere vrijheden. Hierdoor kan er efficiënter worden geprogrammeerd met betrekking tot geheugenruimte en executietijd. Het grote nadeel van dit soort programma's is dat ze nauwelijks wijzigbaar zijn en moeilijk te begrijpen. Bij het ontwikkelen van software gaat het in de eerste plaats vrijwel altijd om besparing van manuren en niet om besparing van geheugenruimte of executietijd. Het is van het grootste belang

dat software goed is gedocumenteerd en gemakkelijk te wijzigen, te testen en uit te breiden. In dit boek zal het ontwikkelen van programma's derhalve steeds geschieden aan de hand van structogrammen.

Bij programma's die erg snel moeten zijn of in een beperkte geheugenruimte moeten passen, verdient het aanbeveling om het programma modulaair op te zetten en de modules voldoende efficiënt te maken. Gooi niet de hele structuur van het programma overboord.

1.7 Software-generaties

Om een computer te programmeren, moeten de instructies en de gegevens worden weergegeven in binaire vorm. In het volgende hoofdstuk wordt behandeld hoe decimale getallen omgezet kunnen worden in binaire getallen en hoe letters kunnen worden weergegeven in een code (ASCII) van (8) binaire tekens.

Adressen van geheugenplaatsen en I/O-adressen in computers worden voorgesteld door getallen. Deze zijn zonder meer te vertalen in binaire getallen.

De bewerkingen (zoals LDA # en BGE) worden bij processoren echter niet voorgesteld door de ASCII-waarden van de letters. Elke bewerking (operatie genoemd) heeft zijn eigen code van 8 bits (Binaire Tekens, 0 of 1). Deze codes (operatiecodes of kortweg opcodes genoemd) worden door de fabrikant van de microprocessor opgegeven. We kunnen ze domweg opzoeken in de data sheet (blad met gegevens) van de microprocessor. Zo is bijvoorbeeld 10000110 de opcode van LDA # en 00101100 die van BGE.

Merk op dat dit veel efficiënter is, omdat het vervangen van de letters door de bijbehorende ASCII-representatie veel meer geheugenruimte kost. Elke letter, uitgedrukt in ASCII, bevat 8 bits. De vier tekens van bijvoorbeeld LDA # gebruiken derhalve 32 bits, terwijl de door de fabrikant vastgestelde binaire opcode slechts 8 bits telt.

Eerste-generatie software

U zult begrijpen dat het programmeren van een computer in binaire code (machinecode) een erg tijdrovende zaak is met een zeer grote kans op fouten. Al heel gauw heeft men hexadecimal loaders ontwikkeld. Deze zijn in staat een programma dat in hexadecimale code wordt ingevoerd, te vertalen in binaire code en deze binaire code vervolgens in het geheugen van de computer op te slaan. Hierdoor kan een programma sneller en met minder fouten worden ingevoerd. Bij de hexadecimale representatie van een binaire code wordt elk groepje van vier binaire tekens vervangen door één hexadecimaal teken. Dit levert dus een reductie op van een factor vier bij het invoeren van een programma. Bovendien wordt het invoeren minder eentonig. Het in te typen programma bestaat nu niet meer uit lange rijen van

nullen en enen, maar uit opeenvolgingen van de cijfers 0 t/m 9 en de letters A t/m F. Hierdoor neemt de kans op vergissingen af.

De hexadecimale code van LDA # is nu 86, die van BGE is 2C. Men spreekt bij het programmeren in hexadecimale code over eerste-generatie software.

Tweede-generatie software

Een aanzienlijke vereenvoudiging werd bereikt door het ontwikkelen van assemblers. Een assembler is een programma dat een assemblyprogramma vertaalt in binaire code (in machinetaal dus). Een assemblyprogramma is een programma dat is geschreven in zogenaamde mnemonics. Mnemonics of geheugensteuntjes zijn symbolische opcodes, zoals LDA # en BGE.

Een programma in machinetaal wordt object program genoemd, een door de programmeur geschreven programma heet source program. Men spreekt bij het programmeren in assembly van tweede-generatie software. Het programmeren in assembly vereist kennis van de microprocessor. Elke microprocessor heeft zijn eigen instructies en registers. Dit betekent dat het op te lossen probleem moet worden vertaald in zeer kleine instructies, zoals LDA # en BGE. Dit soort instructies is machinegericht in plaats van probleemgericht.

Derde-generatie software

Nog weer later ontstonden de probleemgerichte talen, ook wel procedurele talen of hogere programmeertalen genoemd. Deze talen worden tot de derde-generatie software gerekend. De instructies van hogere programmeertalen liggen veel dichterbij de problemen, die wij moeten oplossen. In Pascal bijvoorbeeld kan men de uitkomst Z van een vermenigvuldiging van de getallen X en Y eenvoudig schrijven als $Z := X * Y$. Een compiler is een standaardprogramma dat een programma, geschreven in een hogere taal, vertaalt in een programma in machinetaal. Dit machinetaalprogramma kan door de computer worden uitgevoerd.

Een andere mogelijkheid is een zogenaamde interpreter te gebruiken in plaats van een compiler. Bij toepassen van een interpreter wordt het sourceprogramma in het geheugen van de computer geladen. De interpreter vertaalt nu elke instructie apart in machinetaal, waarna de computer de vertaalde instructie uitvoert. Vervolgens vertaalt de interpreter de volgende instructie, enzovoorts.

Het voordeel van een interpreter is dat een programma instructie voor instructie kan worden uitgevoerd. Dit is vooral gemakkelijk bij het testen. We kunnen kleine stukjes programma testen, zonder dat het gehele programma gereed is. Kleine wijzigingen in het programma kunnen onmiddellijk worden uitgetest.

Bij gebruik van een compiler moet het programma na elke wijziging eerst in zijn geheel worden vertaald.

Een nadeel van een interpreter is dat het uitvoeren van het programma veel meer tijd kost dan bij toepassen van een compiler. Komt in een programma bijvoorbeeld een herhalingslus voor, die 1000 keer wordt doorlopen, dan moeten alle instructies in die lus ook 1000 keer worden vertaald. Dit kost vanzelfsprekend extra tijd.

Bij een compiler worden eerst alle instructies vertaald. Als de computer het programma daarna gaat uitvoeren, hoeven alleen de instructies in de genoemde lus 1000 keer te worden uitgevoerd. Het vertalen is immers al geschied. De door de compiler vertaalde instructies staan in het geheugen van de computer.

Een voordeel van een zogenaamde hogere programmeertaal als Pascal is dat een programma in zo'n taal vrij gemakkelijk te schrijven is. Bovendien is zo'n programma machine-onafhankelijk en derhalve "portable". Dit betekent dat het in principe op elk type computer kan worden uitgevoerd. Een programma in assembly kan alleen op die computers worden uitgevoerd, die dezelfde processor bezitten.

Een nadeel van hogere programmeertalen is dat ze meer geheugenruimte in beslag nemen en vooral dat de uitvoering ervan meer tijd kost. Ook gaat het besturen van signalen op bitniveau meestal aanzienlijk omslachtiger.

Vierde-generatie software

Tegenwoordig vindt de vierde-generatie software opgang. Het gaat hierbij om zogenaamde niet-procedurele talen. In de procedurele talen moet men altijd nog aan de computer meedelen hoe een bepaald probleem opgelost moet worden. Men moet als het ware de procedures opgeven. Bij niet-procedurele talen hoeft men alleen het probleem te specificeren, niet de wijze van oplossen. Hiertoe zijn er program generators ontwikkeld. Deze genereren uit de probleembeschrijving een computerprogramma.

Vijfde-generatie software

Er is druk onderzoek gaande naar de vijfde-generatie software. Hierbij gaat het om software, waarin de problemen omschreven kunnen worden op een manier, die zeer dicht aanligt tegen het gebruik van een natuurlijke taal (zoals bijvoorbeeld Engels). De vijfde-generatie computer zal zich kenmerken door een hoge mate van gebruikersvriendelijkheid.

1.8 Samenvatting

Een processor is een universele digitale schakeling, die standaardbewerkingen kan uitvoeren:

- gegevens binnenhalen
- onthouden
- rekenen

vergelijken
besturen

Programmeren is het in de juiste volgorde zetten van de bewerkingen, die de processor moet uitvoeren om een bepaalde taak te volbrengen.

Het gebruik van structogrammen leidt tot goed gestructureerde programma's. De basisstructuren zijn:

1. volgorde
2. selectie, keuze of beslissing:
 - enkelvoudig
 - meervoudig
3. herhaling, lus of iteratie:
 - do-while
 - repeat-until

Computergeneraties:

	hardware-generaties	software-generaties
1	buizencomputers	hexadecimale programma's
2	transistorencomputers	assemblyprogramma's
3	LSI-technologie	procedurele talen
4	VLSI-technologie	niet-procedurele talen
5		zeer gebruikersvriendelijke software

Fig. 1.19 Hardware- en software-generaties

1.9 Opgaven

- 1 Ontwerp een structogram, waarin een verkeerslicht (dat groen is) op rood wordt gezet. Hierbij moet het licht eerst geel worden, en pas drie seconden later rood.
- 2 Ontwerp een structogram, waarin een verkeerslicht (dat rood is) groen wordt, indien er verkeer aanwezig is. Het licht moet rood blijven, als er geen verkeer is.
- 3 Ontwerp een structogram voor het oplossen van het volgende probleem:
Een bel bij een automatische knipperlichtinstallatie bij een spoorwegovergang moet stil zijn, totdat er een trein nadert. De bel moet luiden, totdat de trein voorbij is en er geen tweede trein nadert.

2 TALSTELSELS EN CODES

2.1 Specifieke doelstelling

Na het bestuderen van hoofdstuk 2 moet u het volgende weten of kunnen:

- * Het principe kennen van het weergeven van getallen in de diverse talstelsels (decimaal, binair en hexadecimaal).
- * De BCD-code begrijpen.
- * Getallen omzetten van het ene stelsel in het andere.
- * Weten hoe de ASCII-code in elkaar zit en een ASCII-boodschap (met behulp van een ASCII-tabel) kunnen vertalen in klare tekst.
- * Optellen en aftrekken in de verschillende stelsels.
- * Carry, borrow en overflow berekenen.
- * Positieve en negatieve getallen weergeven in sign-and-magnitude notatie en in two's complement notatie.
- * Niet-gehele getallen weergeven in fixed point notatie en in floating point notatie.

2.2 Binaire talstelsel

Alvorens uit te leggen hoe een getal op binaire wijze kan worden weergegeven, beschouwen we eerst het vertrouwde decimale (tientallige) stelsel. Voorlopig beperken we ons tot de gehele, positieve getallen.

Een decimaal getal is opgebouwd uit een aantal eenheden, tientallen, honderdtallen, duizendtallen, tienduizendtallen, enzovoorts. Het decimale getal 5827 kan ontleed worden in:

$$5 \cdot 1000 + 8 \cdot 100 + 2 \cdot 10 + 7 \cdot 1 = 5 \cdot 10^3 + 8 \cdot 10^2 + 2 \cdot 10^1 + 7 \cdot 10^0$$

Verkort kunnen we schrijven:

$$5827 = \sum_{i=0}^n a_i \cdot 10^i \text{ met } a_3=5, a_2=8, a_1=2 \text{ en } a_0=7.$$

Hierin geldt:

- i = de exponent van het grondtal 10
- 10^i = het gewicht van het cijfer a_i
- a_i = het cijfer dat aangeeft hoe vaak het gewicht 10^i is begrepen in het getal

Het cijfer a_i wordt ook wel digit genoemd. De digit is de vinger, waarmee wordt aangewezen hoeveel keren het betreffende

gewicht is begrepen in het getal.

In het decimale talstelsel kan het cijfer a_i de waarden 0 t/m 9 aannemen. Dit zijn tien verschillende cijfers. Het grondtal 10 zelf wordt geschreven als 10, een 1 gevolgd door een 0 ($1 \cdot 10^1 + 0 \cdot 10^0$).

Een willekeurig (geheel, positief) getal A kan als volgt worden weergegeven in een talselsel met het grondtal g:

$$A = \sum_{i=0}^n a_i \cdot 10^i$$

Het grondtal in het binaire (tweetaillige) stelsel bedraagt 2 en het binaire cijfer a_i kan slechts een van de twee waarden 0 en 1 aannemen. Het grondtal 2 wordt binair geschreven als $\%10$ ($1 \cdot 2^1 + 0 \cdot 2^0$). Om de binaire getallen te kunnen onderscheiden van decimale, laten we de binaire getallen voorafgaan door het procent-teken.

Zoals het decimale stelsel de gewichten een, tien, honderd, duizend, enzovoorts kent, zo werkt het binaire stelsel met eenheden, tweetaallen, viertallen, achttallen, zestientallen, enzovoorts.

Voorbeeld 2.1

$$\%11010 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 16 + 8 + 2 = 26$$

Een binair cijfer wordt bit genoemd. Dit is de samentrekking van binary digit. De linker bit van een getal vertegenwoordigt het hoogste gewicht en wordt derhalve aangeduid met MSB (Most Significant Bit). Het rechter cijfer wordt LSB genoemd (Least Significant Bit).

Het omrekenen van een decimaal getal naar een binair getal kan geschieden door het decimale getal in machten van twee te ontleden. Hierbij wordt gebruik gemaakt van de formule

$$A = a_n \cdot 2^n + \dots + a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$$

Voorbeeld 2.2

$38 - 1 \cdot 2^5 = 6$	dit levert $\%1\dots$
$6 - 0 \cdot 2^4 = 6$	dit levert $\%10\dots$
$6 - 0 \cdot 2^3 = 6$	dit levert $\%100\dots$
$6 - 1 \cdot 2^2 = 2$	dit levert $\%1001\dots$
$2 - 1 \cdot 2^1 = 0$	dit levert $\%10011\dots$
$0 - 0 \cdot 2^0 = 0$	dit levert $\%100110$

Een andere methode voor het omrekenen van een decimaal getal naar een binair getal gaat uit van de volgende berekening:

$$\begin{aligned} A &= a_n \cdot 2^n + \dots + a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0 = \\ &= (a_n \cdot 2^{n-1} + \dots + a_3 \cdot 2^2 + a_2 \cdot 2^1 + a_1) \cdot 2 + a_0 = \\ &= P \cdot 2 + a_0 \end{aligned}$$

Dit betekent dat bit a_0 gelijk is aan de rest, die ontstaat door het getal A te delen door 2.
Door nu het quotiënt P wederom door 2 te delen, vinden we de restterm a_1 :

$$\begin{aligned} P &= a_n \cdot 2^{n-1} + \dots + a_3 \cdot 2^2 + a_2 \cdot 2^1 + a_1 = \\ &= (a_n \cdot 2^{n-2} + \dots + a_3 \cdot 2^1 + a_2) \cdot 2 + a_1 = \\ &= Q \cdot 2 + a_1 \end{aligned}$$

Op deze wijze kunnen we doorgaan, totdat tenslotte de MSB gevonden is.

Voorbeeld 2.3

38/2 = 19, rest 0	dit levert %.....0
19/2 = 9, rest 1	dit levert %....10
9/2 = 4, rest 1	dit levert %...110
4/2 = 2, rest 0	dit levert %..0110
2/2 = 1, rest 0	dit levert %.00110
1/2 = 0, rest 1	dit levert %100110

Met één bit kunnen slechts twee getallen worden onderscheiden, namelijk 0 en 1. Met 2 bits kunnen we vier verschillende getallen maken: %00, %01, %10 en %11. Door weer een bit toe te voegen, wordt het aantal combinaties nogmaals verdubbeld. Immers we kunnen aan de vier bestaande combinaties een 0 toevoegen of een 1. Zo kunnen we met behulp van n bits 2^n combinaties van nullen en enen maken. Het grootste getal, bestaande uit n bits, is 2^{n-1} . Bedenk hierbij dat het kleinste getal 0 is, zodat we in totaal 2^n verschillende getallen hebben.

2.3 Hexadecimale talstelsel

Het hexadecimale (zestientallige) stelsel heeft als grondtal 16. De digit a_i kan derhalve 16 verschillende waarden aannemen, te weten 0 t/m 15. Elk hexadecimaal cijfer mag slechts uit één teken bestaan, omdat we anders de verschillende cijfers niet van elkaar kunnen onderscheiden. Het zou immers kunnen zijn dat elk cijfer apart een digit voorstelt, of dat twee cijfers samen een digit vormen. Vandaar dat de cijfers 10 t/m 15 in het hexadecimale stelsel worden voorgesteld door de hoofdletters A t/m F.

Hexadecimale getallen zullen we laten voorafgaan door het dollarteken. Op deze manier kunnen we ze onderscheiden van getallen uit andere talstelsels.

Voorbeeld 2.4

$$\begin{aligned} \$10 &= 1 \cdot 16^1 + 0 \cdot 16^0 = 16 \\ \$C7 &= 12 \cdot 16^1 + 7 \cdot 16^0 = 192 + 7 = 199 \\ \$A0E &= 10 \cdot 16^2 + 0 \cdot 16^1 + 14 \cdot 16^0 = 2560 + 14 = 2574 \end{aligned}$$

Het omzetten van decimale getallen in hexadecimale getallen gaat op soortgelijke wijzen als het omzetten van decimale naar binaire getallen.

We moeten nu bepalen hoe vaak de gewichten 16^i begrepen zijn in het getal. Denk hierbij aan de formule

$$A = a_n \cdot 16^n + \dots + a_3 \cdot 16^3 + a_2 \cdot 16^2 + a_1 \cdot 16^1 + a_0 \cdot 16^0$$

Voorbeeld 2.5

$$\begin{aligned} 1200 - 4 \cdot 16^2 &= 176 && \text{dit levert } \$4.. \\ 176 - 11 \cdot 16^1 &= 0 && \text{dit levert } \$4B. \\ 0 - 0 \cdot 16^0 &= 0 && \text{dit levert } \$4B0 \end{aligned}$$

De andere methode gaat als volgt:

$$\begin{aligned} A &= a_n \cdot 16^n + \dots + a_3 \cdot 16^3 + a_2 \cdot 16^2 + a_1 \cdot 16^1 + a_0 \cdot 16^0 = \\ &= (a_n \cdot 16^{n-1} + \dots + a_3 \cdot 16^2 + a_2 \cdot 16^1 + a_1) \cdot 16 + a_0 = \\ &= P \cdot 16 + a_0 \end{aligned}$$

We vinden a_0 als de restterm, die optreedt bij het delen van A door het grondtal 16. Delen van het quotiënt P door 16 levert de restterm a_1 :

$$\begin{aligned} P &= a_n \cdot 16^{n-1} + \dots + a_3 \cdot 16^2 + a_2 \cdot 16^1 + a_1 = \\ &= (a_n \cdot 16^{n-2} + \dots + a_3 \cdot 16^1 + a_2) \cdot 16 + a_1 = \\ &= Q \cdot 16 + a_1 \end{aligned}$$

Delen we Q nu door 16, dan vinden we als restterm a_2 , enzovoorts.

Voorbeeld 2.6

$$\begin{aligned} 1200/16 &= 75, \text{ rest } 0 && \text{dit levert } \$..0 \\ 75/16 &= 4, \text{ rest } 11 && \text{dit levert } \$B0 \\ 4/16 &= 0, \text{ rest } 4 && \text{dit levert } \$4B0 \end{aligned}$$

Er geldt: $\$F = 15 = \%1111$

Dit betekent dat het grootste hexadecimale cijfer gelijk is aan het grootste getal van 4 bits. Hieruit volgt dat elk groepje van 4 bits kan worden weergegeven met één hexadecimaal cijfer. De hexadecimale code kan dus worden gebruikt als een compacte schrijfwijze van de binaire code. Zo ook kan elk hexadecimaal getal eenvoudig in binaire code worden weergegeven, door elk hexadecimaal cijfer te vervangen door 4 bits.

Voorbeeld 2.7

$$\begin{aligned} \$D9 &= \%1101\ 1001 \\ \$62 &= \%0110\ 0010 = \%110\ 0010 \\ \$1F5 &= \%0001\ 1111\ 0101 = \%1\ 1111\ 0101 \\ \%1011\ 0010 &= \%1011\ 0010 = \$B2 \\ \%111\ 0001 &= \%0111\ 0001 = \$71 \\ \$1\ 1000 &= \%0001\ 1000 = \$18 \end{aligned}$$

Voor de overzichtelijkheid verdelen we de binaire codes, voor

zover mogelijk, in groepjes van 4 bits, gescheiden door een spatie. Een groep van 4 bits wordt ook wel nibble genoemd. Een groep van 8 bits noemt men byte.

De getallen moeten na omzetten van hexadecimaal naar binair of omgekeerd, dezelfde waarde behouden. Dit betekent dat de eenheden als eenheden moeten blijven tellen. Hieruit volgt dat, te beginnen bij de eenheden, in linker richting groepjes van 4 bits gevormd moeten worden. Het rechter groepje bits vormt de eenheden in het zestientallige stelsel. Het volgende groepje bits aan de linkerzijde levert de zestientallen op, het daaropvolgende groepje de tweehonderdzesenvijftigtallen, enzovoorts. Dit komt overeen met de gewichten van de betreffende bits (zie fig. 2.1):

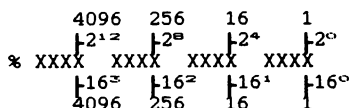


Fig. 2.1 Gewichten van binaire en hexademale cijfers

Het omrekenen van binair naar decimaal en omgekeerd, gaat sneller via hexadecimaal. Dit komt doordat de vertaling tussen binair en hexadecimaal uit het hoofd kan geschieden, terwijl de omzetting van decimaal naar hexadecimaal minder rekenwerk vergt dan die van decimaal naar binair. Er zijn immers minder delingen en vermenigvuldigingen nodig.

Voorbeeld 2.8

$$\begin{aligned} \%1100\ 0101\ 1110 &= \$C5E = 12 \cdot 16^2 + 5 \cdot 16^1 + 14 \cdot 16^0 = \\ &= 3072 + 80 + 14 = 3166 \end{aligned}$$

Dit kan ook volgens de berekening:

$$\$C5E = (12 \cdot 16 + 5) \cdot 16 + 14 = 3166$$

$$679 = 2 \cdot 16^2 + 10 \cdot 16^1 + 7 \cdot 16^0 = \$2A7 = \% 10\ 1010\ 0111$$

2.4 BCD-code

U hebt gezien dat het omzetten van binaire getallen in decimalen vrij moeizaam is. Er bestaat een eenvoudiger manier om decimale getallen binair weer te geven. Hiervoor wordt de BCD-code gebruikt. BCD is de afkorting van Binary Coded Decimals. Dit betekent dat we blijven werken in het tientallige stelsel. De decimale cijfers worden echter, elk apart, binair weergegeven. Per decimaal cijfer zijn hiervoor 4 bits nodig.

Voorbeeld 2.9

$$745 = \text{BCD } 0111\ 0100\ 0101$$

$$903 = \text{BCD } 1001\ 0000\ 0011$$

$$\text{BCD } 0010\ 1000 = 28$$

$$\text{BCD } 1\ 0110\ 0101 = 165$$

De BCD-code wordt vaak gebruikt bij de weergave van resultaten. We zien immers de uitkomsten graag in decimale vorm. Zeven-segment-displays worden aangestuurd door getallen in BCD-code. Processoren rekenen doorgaans niet in BCD-code. De rekenschakelingen zouden hierdoor complexer worden, immers in BCD-code worden de bitposities niet allemaal volgeteld. Na BCD 1001 komt BCD 1 0000. In het binaire talstelsel wordt gewoon doorgeteld via %1010 en na %1111 volgt pas de stand %1 0000. Hieruit blijkt tevens dat er in de BCD-code weinig efficiënt met bits wordt omgesprongen. Het getal 1023 kost 10 bits in het binaire talstelsel en 13 in BCD-code:

1023 = %11 1111 1111 = BCD 1 0000 0010 0011

2.5 ASCII-code

Het is niet voldoende om alleen getallen in computers op te slaan. Er moeten ook teksten worden opgeborgen, bijvoorbeeld ten behoeve van voorraadbeheer, personeelsadministratie en commentaar bij sourceprogramma's. Voor dit doel wordt de ASCII-code zeer veel toegepast. ASCII staat voor American Standard Code for Information Interchange. Deze code gebruikt 7 bits.

Appendix A bevat de volledige ASCII-tabel.

De codes %000 0000 t/m %001 1111 worden gebruikt voor speciale commando's, zoals Start of Text, End of Text en Carriage Return.

De decimale cijfers 0 t/m 9 worden in ASCII weergegeven door de codes %011 0000 t/m %011 1001. Deze komen overeen met de binaire representaties, voorafgegaan door %011.

De hoofdletters A t/m Z worden achtereenvolgens weergegeven door de codes %100 0001 t/m %101 1010 en de kleine letters a t/m z door

%110 0001 t/m %111 1010.

De resterende codes worden gebruikt voor de representatie van de leestekens (! " , . ; enzovoorts) en voor een aantal speciale tekens (zoals # \$ % & =).

Computers werken meestal met 8 bits (of veelvouden daarvan). De meest linker bit is niet van belang. Hij kan eenvoudig 0 worden gemaakt of worden gebruikt als pariteitsbit.

Indien boodschappen over grotere afstanden worden getransporteerd, kan er wel eens een bit verkeerd overkomen. Het meezenden van een extra pariteitsbit signaleert dit. Stel dat er wordt afgesproken dat de pariteit altijd even moet zijn. Dit betekent dat de som van de 8 bits (dus alle bits, inclusief de pariteitsbit zelf) even moet zijn. Dit kan worden bereikt door de pariteitsbit 0 te maken als de som van de 7 databits even is, en door de pariteitsbit 1 te maken als de som van de 7 databits oneven is.

Voorbeeld 2.10

We versturen de boodschap "Pas op!" in ASCII en voegen een even pariteitsbit toe (zie appendix A). Deze bit moet de pariteit even maken:

	zonder pariteitsbit	met even pariteitsbit
p	% 101 0000	%0101 0000
a	% 110 0001	%1110 0001
s	% 111 0011	%1111 0011
spatie	% 010 0000	%1010 0000
o	% 110 1111	%0110 1111
p	% 111 0000	%1111 0000
!	% 010 0001	%0010 0001

Zou nu door een storing onderweg bijvoorbeeld bit 0 van karakter a veranderen, dan wordt dit ontvangen als %1110 0000 (in plaats van %1110 0001). De ontvanger ontdekt nu dat de pariteit oneven is en concludeert dat er een fout is opgetreden. Er kan nu eventueel om herhaling van het bericht verzocht worden.

Veranderen een oneven aantal bits, dan leidt dit tot een verandering van de pariteit. Bij veranderen van een even aantal bits blijft de pariteit gelijk, hetgeen dus niet tot een pariteitsfout leidt. U ziet dat niet alle fouten op deze wijze gedetecteerd worden. Er bestaan andere methoden om fouten te detecteren of zelfs te corrigeren. Bij deze methoden moeten er echter meer bits worden toegevoegd. We gaan hier niet verder op in, omdat dit buiten het kader van dit boek valt.

2.6 Getallen optellen

We zullen nu eens gaan kijken hoe we binaire en hexadecimale getallen kunnen optellen.

Bij het optellen van decimale getallen gaan we als volgt te werk. Eerst tellen we de minstwaardige cijfers op (de eenheden). De uitkomst noteren we (op de plaats van de eenheden), mits deze kleiner is dan het grondtal 10. Is de som van deze cijfers 10 of groter, dan trekken we eerst het grondtal 10 eraf en schrijven het resultaat hiervan op. We moeten nu een tiental onthouden. Dit moeten we immers optellen bij de som van de tientallen. De som van de tientallen en de eventuele "1-onthouden" noteren we op de plaats van de tientallen, eventueel na aftrekken van 10, indien deze som groter of gelijk is aan 10. In dit laatste geval moeten we weer 1 onthouden, die we nu moeten optellen bij de som van de honderdtallen, enzovoorts. Om de "1-onthouden" niet te vergeten, schrijven we deze boven de cijfers waarbij hij moet worden opgeteld.

Voorbeeld 2.11

```
(0110 1-onthouden)
 3625
 2794 +
 6419
```

Binaire getallen optellen gaat op dezelfde manier. Nu is echter het grondtal 2 en een cijfer kan hoogstens 1 zijn. Is een tussenuitkomst groter of gelijk aan 2, dan moeten we het grondtal 2 eraf trekken en het verschil noteren. Bovendien moeten we weer 1 onthouden om deze bij de som van de naastliggende hogere cijfers op te tellen.

Voorbeeld 2.12

```
(%111 101 1-onthouden)
% 10 1101
 % 11 1101 +
%110 1010
```

Het optellen van hexadecimale getallen gaat op dezelfde manier. Bedenk dat nu het grondtal 16 is en het grootste cijfer 15.

Voorbeeld 2.13

```
($1100 1-onthouden)
$ 5AC2
 $ A919 +
$103DB
```

Normaliter schrijven we het rijtje met "1-onthouden" niet op. Dit hebben we hier slechts gedaan omwille van de duidelijkheid.

2.7 Carry

Bij computers hebben de registers een beperkte omvang, bijvoorbeeld 8 bits. Dit betekent dat de som van twee getallen wel eens te groot zou kunnen zijn, dit wil zeggen dat deze niet in het register past. De bit die bij een bewerking buiten het register terechtkomt, wordt carry genoemd. Dit is bij een 8-bits register (b0 t/m b7) in feite de transportbit (ofwel de "1-onthouden") van b7 naar de niet-bestaande b8.

Voorbeeld 2.14

%1000 1011	%1100 1101
<u>%0101 0010</u> +	<u>%1001 0001</u> +
(0)%1101 1101	(1)%0101 1110
carry=0	carry=1

De carry wordt in een apart register in de microprocessor opgeslagen. Dit komt echter later ter sprake (zie hoofdstuk 3).

2.8 Getallen aftrekken

Aftrekken gaat op soortgelijke wijze als optellen. Alleen moeten we hier 1 lenen in plaats van 1 onthouden. In het tientallige stelsel lenen we een tiental, bij het binaire talstelsel lenen we een tweetal en bij het hexadecimale talstelsel lenen we een zestiental.

Voorbeelden 2.15

(011 1-lenen)	(%11 100 1-lenen)	(\$011 1-lenen)
2358	%10 1001	\$4A30
<u>1079</u> -	<u>%01 1101</u> -	<u>\$1952</u> -
1279 -	%00 1100	\$30DE -

2.9 Borrow

We hebben nu gezien dat we bij aftrekken soms 1 moeten lenen van het naastliggende hogere cijfer. Dit gaat niet als het naastliggende hogere cijfer niet bestaat. Denk hierbij bijvoorbeeld aan de niet-bestaande b8 van een 8-bits register of aan de niet-bestaande b16 van een 16-bits register. We spreken hier van de borrow. Een borrow treedt dus op als we 1 moeten lenen van een niet-bestaande bit. Dit is onder andere het geval als we een positief getal verminderen met een groter getal.

Voorbeeld 2.16

%0100 1100	\$7C
<u>%1011 0100</u> -	<u>\$46</u> -
(1)%1001 1000	(0)\$36
borrow=1	borrow=0

Tot nu toe hebben we alleen positieve getallen beschouwd. We spreken hier van unsigned numbers (getallen zonder teken). Bij aftrekken kunnen er negatieve getallen ontstaan. Ook deze moeten weergegeven kunnen worden. Hiervoor bestaan diverse methoden. Twee ervan worden in de volgende paragraaf behandeld.

2.10 Negatieve getallen

Sign-and-magnitude notatie

In het tientallige stelsel zijn we gewend onderscheid te maken tussen positieve en negatieve getallen door een speciaal teken voor het getal te plaatsen: een "+" voor een positief getal en een "-" voor een negatief getal. De grootte van het getal staat achter het teken. We noemen deze notatie derhalve "sign-and-magnitude".

In het binaire stelsel kunnen we deze tekens niet zonder meer weergeven. We kennen immers alleen de 0 en de 1. De afspraak is

dat de "+" aangegeven wordt met een 0 en de "-" met een 1. Het teken staat ook hier links van de waarde van het getal. Dit betekent dat de linker bit van een getal de tekenbit is.

Voorbeeld 2.17

%01110 = +14	\$ 7 = + 7	(aangenomen dat \$7 en
%11110 = -14	\$ F = - 7	\$F 4-bits getallen zijn)
% 111 = - 3	\$A2 = -34	(aangenomen dat \$A2 en
% 0111 = + 7	\$61 = +97	\$61 8-bits getallen zijn)

Bij 6-bits getallen zal -3 in sign-and-magnitude notatie geschreven dienen te worden als %100011 en +7 als %000111.

Nu is het werken met sign-and-magnitude getallen vrij omslachtig. Stel dat we twee sign-and-magnitude getallen moeten optellen. We gaan dan als volgt te werk:

Hebben beide getallen hetzelfde teken (+ dan wel -), dan tellen we de absolute waarden (de grootten) van deze getallen bij elkaar op en zetten vervolgens het teken (+ respectievelijk -) voor de uitkomst.

Zijn de tekens verschillend, dan trekken we het getal met de kleinste absolute waarde af van het getal met de grootste absolute waarde en zetten het teken van het getal met de grootste absolute waarde voor de uitkomst.

U ziet dat dit inderdaad vrij omslachtig is. Bij het aftrekken van twee sign-and-magnitude getallen moeten we een soortgelijke procedure volgen.

Two's complement notatie

Nu bestaat er een andere notatie voor getallen, die zowel positief als negatief kunnen zijn. Voor binaire getallen is dit de two's complement notatie. Het rekenen met getallen in deze notatie zal blijken zeer eenvoudig te zijn.

Om de two's complement notatie goed te begrijpen, moet u zich realiseren dat een negatief getal niets anders is dan een even groot positief getal dat is afgetrokken van 0. Zo is het getal -17 eigenlijk 0-(17). Zo gaan we ook te werk bij two's complement getallen.

Een positief getal wordt gewoon voorgesteld door de waarde van het getal, echter voorafgegaan door een 0 om aan te geven dat het getal positief is (net als bij de sign-and-magnitude notatie). Zo schrijven we +5 in two's complement notatie als 0101.

Een negatief getal schrijven we in de two's complement notatie door de absolute waarde (die per definitie positief is en dus voorafgegaan moet worden door een 0) af te trekken van 0. Dit betekent dat we de two's complement notatie van -5 vinden door +5 (= %0101) af te trekken van 0 (= %0000). We gaan hierbij uit van een 4-bits getal, waarbij de meest linker bit (b3) de tekenbit is.

%0000	(0)
<u>%0101</u> -	(<u>5</u>)
%1011	(-5)

Kennelijk is %1011 de two's complement schrijfwijze voor -5. Dat het getal negatief is kunnen we zien aan de tekenbit. Deze is inderdaad gelijk aan 1. De grootte van het getal kunnen we niet zonder meer zien. Trekken we het negatieve getal af van 0, dan vinden we de absolute waarde van het getal:

%0000	(0)
<u>%1011</u> -	(<u>5</u>)
%0101	(+5)

We vinden inderdaad als uitkomst van deze aftrekking een positief getal (de linker bit is immers 0) en derhalve staat de grootte (5) er achter.

Nu vraagt u zich misschien af wat er met de borrow is gebeurd. Immers als we een getal aftrekken van een kleiner getal, ontstaat er een borrow. We moeten 1 lenen van een cijferpositie die niet bestaat. Bij two's complement getallen moeten we deze borrow weglaten. Immers alles wat links van de tekenbit staat, is onzin. Het teken neemt altijd de meest linker positie in. Dit betekent ook dat een carry, die ontstaat bij het optellen van two's complement getallen, niet mag worden meegerekend. Immers ook deze carry komt terecht op een positie links van de tekenbit en doet derhalve niet mee.

U kunt het ook op een andere manier bekijken. Bij het bepalen van de two's complement notatie van een negatief getal moet de grootte van dit getal worden afgetrokken van 0. Hierbij moet 0 worden geschreven als een rij van oneindig veel nullen. Bij dit aftrekken ontstaan er links oneindig veel enen. Tellen we bij het aldus verkregen verschil een groter positief getal op, dan krijgen we links oneindig veel nullen. De uitkomst is immers positief.

Al die enen en nullen bevatten geen informatie en worden derhalve weggelaten. Alleen de meest linker 0 of 1 van de aaneengesloten rij nullen en enen moet worden behouden als de tekenbit.

% (000....00) 0000	(0)	% (111....11) 1011	(-5)
<u>% (000....00) 0101</u> -	(+5)	<u>% (000....00) 0111</u> +	(+7)
% (111....11) 1011	(-5)	% (000....00) 0010	(+2)

Uiterste waarden

Het grootste positieve getal bestaat uit de tekenbit (=0) gevolgd door allemaal enen. Dit geldt zowel voor de sign-and-magnitude notatie als voor de two's complement notatie. Een getal van n bits heeft n-1 posities voor de grootte van het getal. Dit betekent dat het grootste getal gelijk is aan $+2^{n-1}-1$.

Een sign-and-magnitude getal en een two's complement getal van 8 bits kunnen dus maximaal $+2^7-1 = +127$ bedragen.

Een 8-bits sign-and-magnitude getal kan niet minder bedragen dan -127 . Ook hier wordt het teken immers gevolgd door de grootte.

Bij negatieve two's complement getallen ligt het anders. We vinden hier de grootte van een negatief getal door dit af te trekken van 0. Dit betekent dat de absolute waarde maximaal is, als het getal dat van 0 wordt afgetrokken, minimaal is. Dit getal moet dus zoveel mogelijk nullen bevatten en bestaat derhalve uit een 1 (vanwege het minteken), gevolgd door allemaal nullen. Het meest-negatieve 8-bits two's complement getal is dus $\%1000\ 0000$. De absolute waarde wordt gevonden door dit getal af te trekken van 0:

```
%0000 0000
%1000 0000 -
%1000 0000 -
```

De grootte bedraagt dus $2^7 = 128$. Dit betekent dat de waarde van het negatieve 8-bits getal in two's complement notatie gelijk is aan -128 .

Algemeen geldt dat de meest-negatieve waarde van een two's complement getal van n bits gelijk is aan -2^{n-1} .

Een sign-and-magnitude getal van 8 bits ligt tussen de grenzen -127 en $+127$, een two's complement getal tussen -128 en $+127$. U ziet dat two's complement getallen een iets groter bereik hebben dan sign-and-magnitude getallen. Dit is niet zo verwonderlijk als u bedenkt, dat het getal 0 in de laatste notatie op twee manieren kan worden weergegeven, namelijk als $+0$ en als -0 . Dit is respectievelijk $\%0000\ 0000$ en $\%1000\ 0000$. In de two's complement notatie is 0 altijd positief: $\%0000\ 0000$.

Rekenen met two's complement getallen

Er is opgemerkt dat het rekenen in two's complement notatie erg eenvoudig is. We zullen dit aan de hand van enkele voorbeelden laten zien.

Voorbeeld 2.18

$\%0011$	$(= +3)$	$\%1011$	$(= -5)$
$\%0010$	$(= +2)$	$\%0110$	$(= +6)$
$\%0101$	$(= +5)$	$\%0001$	$(= +1)$

Zoals u ziet kloppen de beide uitkomsten van voorbeeld 2.18. Laten we nu eens kijken hoe het gaat bij aftrekken.

Voorbeeld 2.19

$\%0010$	$(= +2)$	$\%0000$	$(= 0)$
$\%0110$	$(= +6)$	$\%1100$	$(= -4)$
$\%1100$	$(= -4)$	$\%0100$	$(= +4)$

De uitkomst van (+2)-(+6) is negatief, immers de linker bit van het resultaat is 1. We vinden de grootte door dit negatieve getal af te trekken van 0. Dit levert +4, hetgeen betekent dat het resultaat van de berekening -4 is. Dit is correct.

Voorbeeld 2.20

```
%0011      (= +3 )
%1100 _    (= -4 _)
%0111      (= +7 )
```

Ook hier zien we tot ons genoegen dat de uitkomst juist is.

Nu is het bepalen van de two's complement notatie van een negatief getal een nogal omslachtige zaak. Dit komt doordat we bij het aftrekken van 0 steeds enen moeten lenen. Nu kunnen we 0 (= %0000) ook schrijven als %1111 + 1:

```
%1111
  _1_ +
%0000
```

Dit betekent dat de volgende berekening geldt:

```
(negatief getal in two's complement) =
= 0 - (absolute waarde) =
= (%1111 + 1) - (absolute waarde) =
= (%1111 - (absolute waarde)) + 1
```

Het aftrekken van %1111 is veel eenvoudiger, daar er nooit geleend hoeft te worden. Zo kunnen we de two's complement notatie van -3 als volgt bepalen:

```
%1111
%0011 _    (= +3)
%1100
  _1_ +
%1101      (= -3)
```

Kijkt u nog eens naar het laatste voorbeeld, dan ziet u dat een getal aftrekken van %1111 in feite neerkomt op het inverteren van dat getal. U kunt dus zeggen:

```
(negatief getal in two's complement) =
= (inverse van absolute waarde) + 1
```

2.11 Overflow

Aangezien computers werken met registers met een beperkte breedte, kunnen er problemen ontstaan bij berekeningen. We hebben dit al gezien bij het optellen van unsigned numbers. Er kan dan immers een carry ontstaan. Die carry vertegenwoordigt een gewicht, behorende bij de bit die net niet in het register past.

Ook bij het rekenen met two's complement numbers kunnen we een soortgelijk probleem verwachten.

Voorbeeld 2.21

```
%0100    (= +4 )
%0110 +  (= +6 +)
%1010    (= -6 )
```

De uitkomst van (+2)+(6) levert hier een negatief getal op, immers de tekenbit is 1. Dit is natuurlijk fout, want de som van twee positieve getallen is altijd positief. De uitkomst (+10) past niet in 4 bits. (+10 = %01010.) De uitkomst loopt als het ware over het 4-bits register heen. Men spreekt hier van overflow. Ook het optreden van een overflow wordt (net als het optreden van een carry) in een speciaal register in de microprocessor bewaard, maar hierover meer in hoofdstuk 3. De overflow is dus in feite een two's complement overflow, terwijl de carry is op te vatten als een unsigned overflow. In het eerste geval gaat het immers om een two's complement resultaat dat niet in het register past, in het tweede geval loopt de unsigned uitkomst over. In het vervolg zullen we steeds spreken over overflow, respectievelijk carry.

Overflow treedt alleen op, zodra er een tekenfout in de uitkomst ontstaat. In alle andere gevallen is er geen overflow.

Voorbeeld 2.22

%1110 (= -2)	%1011 (= -5)
<u>%1101</u> + (= -3 +)	<u>%1001</u> + (= -7 +)
%1011 (= -5)	%0100 (= +4)
teken is goed, dus overflow=0	teken is fout, dus overflow=1
%0101 (= +5)	%0110 (= +6)
<u>%1001</u> - (= -7 -)	<u>%1111</u> - (= -1 -)
%1100 (= -4)	%0111 (= +7)
teken is fout, dus overflow=1	teken is goed, dus overflow=0

Als de waarde van de overflow gelijk aan 1 is, moeten we in feite met meer bits werken om de juiste uitkomst te verkrijgen. Laten we dit eens bekijken voor de berekening van (+5)-(-7). Met een extra bit vinden we dan:

```
%00101    (= +5 )
%11001 -  (= -7 -)
%01100    (= +12 )
teken is goed, dus overflow=0
```

Overflow treedt op als er voor de two's complement uitkomst niet voldoende bits beschikbaar zijn. Dit komt tot uitdrukking in de tekenbit. Als de uitkomstruimte te klein is, wordt er bij de tekenbit in feite 1 geleend (bij aftrekken) of 1 bijgeteld

(bij optellen). In beide gevallen krijgt het teken hierdoor de verkeerde waarde.

Bij het aftrekken van twee getallen met hetzelfde teken en bij het optellen van twee getallen met een verschillend teken, kan de uitkomst positief of negatief zijn. Er kan kan derhalve geen verkeerd teken optreden, zodat de overflow 0 zal zijn. Dit klopt ook, want in deze gevallen is de uitkomst kleiner dan de getallen zelf. Als de getallen in het register passen, zal de uitkomst in dit geval zeker passen.

De uitkomst kan alleen te groot zijn bij het optellen van getallen met hetzelfde teken of bij het aftrekken van getallen met een verschillend teken.

Fig. 2.2 laat zien in welke gevallen wel en in welke gevallen geen overflow kan optreden.

overflow = 0			overflow = 1		
getal 1 + getal 2 = som			getal 1 + getal 2 = som		
+	+	+	+	+	-
+	-	+/-			
-	+	+/-			
-	-	-	-	-	+
-----			-----		
getal 1 - getal 2 = verschil			getal 1 - getal 2 = verschil		
+	+	+/-			
+	-	+	+	-	-
-	+	-	-	+	+
-	-	+/-			

Fig. 2.2 Overflow bij optellen en aftrekken van two's complement getallen

In de praktijk van de microprocessoren breiden we de getallenruimte in geval van overflow niet met één bit uit, maar met een heel register. Past een resultaat niet in een 8-bits register (bij een microprocessor die met 8-bits registers werkt), dan nemen we er een 8-bits register bij, zodat we nu een ruimte hebben gekregen van 16 bits om in verder te rekenen. Hetzelfde doen we als er een carry optreedt bij het rekenen met unsigned numbers.

Een getal dat met een 0 begint (aan de linkerzijde), kunnen we naar links uitbreiden met een onbeperkt aantal nullen. Hierdoor verandert de waarde van dat getal niet. We moeten de hogere gewichten telkens 0 keer meetellen. Bovendien blijft de linker bit 0, waardoor ook het teken van het (positieve) getal niet verandert.

Een negatief getal in two's complement notatie (dat dus met een 1 begint), kunnen we naar links uitbreiden met een onbeperkt aantal enen. Hierdoor blijft de tekenbit 1. Het getal was en blijft door toevoegen van enen dus negatief. Ook de grootte van het getal zal niet veranderen. Om de grootte te vinden

inverteren we alle bits en tellen bij het resultaat hiervan 1 op. Door het inverteren gaan alle voorafgaande enen in nullen over en dragen derhalve niet bij tot de grootte van het getal.

2.12 Niet-gehele getallen

Nu u de schrijfwijzen van positieve en negatieve getallen kent, rest ons nog in te gaan op de niet-gehele getallen. Laten we eerst eens kijken hoe dat bij de decimale getallen gaat:

$$24.81 = 2 \cdot 10^1 + 4 \cdot 10^0 + 8 \cdot 10^{-1} + 1 \cdot 10^{-2}$$

Dit betekent dat we een getal in het tientallige stelsel als volgt kan worden weergegeven:

$$A = \sum_{i=-m}^n a_i \cdot 10^i \quad (m \text{ en } n \text{ zijn gehele, positieve getallen})$$

Vervangen we het grondtal 10 in het grondtal g , dan krijgen we de algemene formule voor niet-gehele getallen:

$$A = \sum_{i=-m}^n a_i \cdot g^i$$

In het tweetallige stelsel krijgen we dus te maken met halven, kwarten, achtsten, enzovoorts, in plaats van met tienden, honderdsten, duizendsten, enzovoorts. Zo zullen we in het hexadecimale talstelsel de gewichten $1/16$, $1/256$, $1/4096$, enzovoorts tegenkomen.

Voorbeeld 2.23

$$\begin{array}{ll} \%10110.101 = 22.625 & \$A0.8 = 160.5 \\ \%01111.010 = 15.25 & \$34.4 = 52.25 \end{array}$$

Negatieve breuken kunnen worden weergegeven in sign-and-magnitude notatie en in two's complement notatie.

Voorbeeld 2.24

$$\begin{array}{l} -3.25 = 111.01 \text{ in sign-and-magnitude notatie} \\ \quad = 100.11 \text{ in two's complement notatie} \end{array}$$

Op de een of andere wijze moet de plaats van de binaire komma worden aangegeven. Door hiervoor een vaste positie af te spreken, kost dit geen extra ruimte. Stel we spreken af dat de binaire punt staat tussen b_3 en b_4 bij een binair getal. Dan is de waarde van het unsigned number $\%10010110$ gelijk aan 9.375. We spreken hier van fixed point notatie.

Voor het noteren van zeer grote of zeer kleine getallen hebben we veel cijfers en dus veel geheugenruimte nodig. Bovendien wordt de verwerkingstijd van deze getallen erg groot. Er kunnen slechts een beperkt aantal cijfers tegelijk verwerkt worden. Denk bijvoorbeeld aan het optellen van twee decimale getallen.

We tellen telkens twee cijfers bij elkaar op (te beginnen bij de minstwaardige cijfers). Na deze bewerking een aantal malen te hebben herhaald, hebben we de som van de twee getallen verkregen. Hoe groter de getallen zijn (dus hoe meer cijfers de getallen bevatten), des te langer duurt de bewerking. Nu bestaat er een kortere notatie voor getallen. Deze kost dus minder geheugenruimte en minder verwerkingstijd. Het is de zogenaamde floating point notatie. Hierbij wordt een getal voorgesteld door een macht van het grondtal en een fractie. Indien nodig moeten de fracties worden afgerond!

Voorbeeld 2.25

10500000	= $10^{+8} \cdot 0,105$	%11000000	= $2^{+7} \cdot \%0,1110$
0,00000073	= $10^{-6} \cdot 0,730$	%0,0000101	= $2^{-6} \cdot \%0,1010$
45791,0234	= $10^{+5} \cdot 0,458$	%110,101	= $2^{+5} \cdot \%0,1101$
21,745	= $10^{+2} \cdot 0,217$	%100,1101	= $2^{+3} \cdot \%0,1010$

Bij de twee laatste rijen van voorbeeld 2.25 is telkens afgerond.

De "0," bevat bij de floating point notatie geen informatie. Dit betekent dat we deze "0," kunnen weglaten, hetgeen geheugenruimte bespaart.

In het decimale talstelsel moet de fractie groter of gelijk zijn aan 0,1 en kleiner dan 1.

In het binaire talstelsel moet de fractie groter of gelijk zijn aan 0,5 en kleiner dan 1.

Een uitzondering hierop vormt het getal 0. Voor dit getal is de fractie natuurlijk gelijk aan 0.

Ook hier kunnen we weer gebruik maken van de sign-and-magnitude notatie of van de two's complement notatie voor het aangeven van positieve en negatieve getallen. Hierbij kunnen de fractie en de exponent afzonderlijk positief of negatief zijn.

Voorbeeld 2.26

Eerst komt de exponent in 4 bits en daarachter de fractie in 6 bits.

25 = %11001	= $2^{+5} \cdot \%0,11001$	= %0101011001 (sign-and-magn.)
		= %0101011001 (two's complement)
-80 = -%1010000	= $2^{+7} \cdot (-\%0,101)$	= %0111110100 (sign-and-magn.)
		= %0111110100 (two's complement)
0,125 = %0,001	= $2^{-2} \cdot \%0,1$	= %1010010000 (sign-and-magn.)
		= %1110010000 (two's complement)
-0,0390625 = -%0,0000101	= $2^{-6} \cdot (-\%0,101)$	= %1100110100 (sign-and-magn.)
		= %1100110100 (two's complement)

Fixed point getallen hebben een constante absolute nauwkeurigheid. Deze is gelijk aan de helft van het gewicht van de minstwaardige bit (mits er op de juiste wijze is afgerond). De relatieve nauwkeurigheid is bij kleine getallen erg klein. Floating point getallen hebben een vrijwel constante relatieve

- Carry = de "1-onthouden" die niet in het register past
- Borrow = de "1-lenen" buiten het register
- Overflow = het signaal dat aangeeft, dat het two's complement resultaat niet in het register past; overflow treedt op als het resultaat een verkeerd teken heeft.

Sign-and-magnitude notatie: teken, gevolgd door grootte
Two's complement notatie: teken, gevolgd door grootte,
bij positief getal
uitkomst van (0 - grootte),
bij negatief getal

Tekenbit (= linker bit) = 0 voor +
= 1 voor -

Fixed point notatie: komma heeft vaste positie in het getal
Floating point notatie: exponent wordt gevolgd door de
genormeerde fractie (dit is een fractie
waarvan het meestwaardige cijfer
ongelijk aan 0 is)

2.14 Opgaven

- 1 a. Zet 725 om in het hexadecimale en binaire talstelsel en in de BCD- code.
- b. Zet \$1AF om in het decimale en binaire talstelsel en in de BCD-code.
- c. Bereken de decimale en de hexadecimale waarde van %1001 1101 0010 en schrijf dit getal in de BCD-code.
- d. Bereken de decimale, hexadecimale en binaire waarde van BCD 1000 0111 0101.
- 2 Codeer (met behulp van appendix A) de boodschap "Zend 3 bytes." in ASCII.
- 3 Bereken in het binaire talstelsel de som van %0110 1101 en %0100 1101.
- 4 Bereken in het hexadecimale talstelsel de som van \$2C9 en \$791.
- 5 Wat is de waarde van de carry bij het optellen van \$9A en \$3C?
- 6 Verminder in het binaire talstelsel %1100 1011 met %0111 0101.
- 7 Trek in het hexadecimale talstelsel \$37 af van \$73.

- 8 Wat is de waarde van de borrow als we 351 verminderen met 718?
- 9 Geef de (binaire) waarde van +94 en van -57 in 8-bits sign-and-magnitude notatie.
- 10 Geef de waarden van +27 en -111 in 8-bits two's complement notatie.
- 11 Bepaal de grenzen van de gehele 4-bits getallen voor unsigned, sign-and-magnitude en two's complement.
- 12 Geef de decimale waarde van de unsigned numbers %1101,111 en \$50,A.
- 13 Geef de fixed point notatie van de getallen +36,75 en -11,0625 in het two's complement als de grootte van de binaire getallen 12 bits bedraagt en de binaire komma zich bevindt tussen de bits b4 en b5.
- 14 Wat is de decimale waarde van de floating point getallen A en B als deze getallen gegeven zijn in sign-and-magnitude notatie, waarbij de 5-bits exponent gevolgd wordt door een 7-bits fractie.
A = %101100110100
B = %010011011001
- 15 Wat is de decimale waarde van de floating point getallen X en Y als deze getallen gegeven zijn in two's complement notatie, waarbij de 5-bits exponent gevolgd wordt door een 7-bits fractie.
X = %000111000101
Y = %111011110000

3 DE MICRO-PROCESSOR

3.1 Specifieke doelstelling

Na hoofdstuk 3 bestudeerd te hebben, moet u tot het volgende in staat zijn:

- * Weten wat terminals, keyboards, video displays en disk units zijn.
- * De opbouw van een computer en van een microprocessor kennen.
- * De functie van de belangrijkste onderdelen van computer en processor kennen.
- * Beschrijven wat een instructiecyclus is en in welke stappen een instructie door de microprocessor wordt uitgevoerd.
- * De functie van een monitorprogramma en een operating system kennen.
- * Het programmeermodel van de MC6809 kennen.
- * De betekenis van de bits van het condition code register kennen.

3.2 Een computertoepassing

Om de functies en het nut van de diverse onderdelen van een microprocessor beter te kunnen begrijpen, gaan we eerst een voorbeeld van een computerconfiguratie bekijken.

We kiezen hier een bedrijf met een centrale computer, die zorg draagt voor het bijhouden van de voorraadadministratie van het magazijn, de benodigde informatie verstrekt aan de assemblagerobots in de fabriek, de regeling van de centrale-verwarmingsinstallatie verzorgt en de orderadministratie en de personeelsadministratie bijhoudt (zie fig. 3.1).

Om de voorraad in een magazijn van een productiebedrijf bij te houden, moeten we de computer kunnen meedelen welke artikelen en hoeveel er in het magazijn binnenkomen en wat er uitgaat. Hiertoe hebben we een zogenaamde terminal nodig, die met de computer is verbonden. Een terminal bestaat uit een keyboard (toetsenbord) en een video display unit, ook wel monitor genoemd (een soort televisie-apparaat). Via het keyboard kunnen we de gegevens, omtrent de veranderingen in de voorraad, aan de computer meedelen. Deze berekent dan de nieuwe voorraad en geeft eventueel een melding op het video display, als een bepaald artikel dreigt op te raken. Ook kan de computer via het

video display de voorraadadministratie laten zien. Dit doet hij natuurlijk alleen als hij daarom gevraagd wordt. De opdracht om zulks te doen, moeten we de computer weer meedelen via het keyboard.

De computer is niet volledig bezet met het bijhouden van de voorraadadministratie. We zouden hem tegelijkertijd de assemblerobots kunnen laten aansturen. Deze robots zorgen bijvoorbeeld voor het in elkaar lassen van de onderdelen van een autochassis. De computer zorgt ervoor, dat de robots hiertoe de juiste bevelen krijgen. Verder kan de computer ook zorg dragen voor het transport van de juiste onderdelen tussen magazijn en robots.

Daarnaast zou de computer ook nog de regeling van de centrale-verwarmingsinstallatie van het bedrijf kunnen verzorgen. Hij zorgt er onder andere voor, dat op de juiste momenten de waterpomp loopt en de hoofdbranders aan- en uitgaan. Hiervoor dienen van tijd tot tijd de diverse temperaturen te worden gemeten. Denk aan het voorbeeld van de processor als regelaar in paragraaf 1.3.

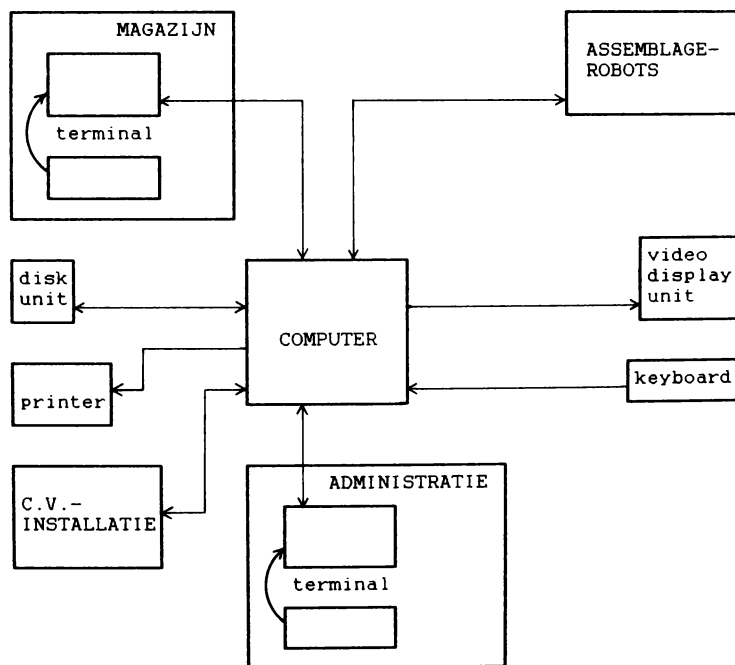


Fig. 3.1 Computerconfiguratie

Verder zou de nimmer vermoeibare en razendsnelle werker de orderadministratie en de personeelsadministratie kunnen bijhouden. De gegevens, die hij hiervoor nodig heeft, worden hem toespeeld via een keyboard. Op het video display kunnen we zien of de juiste gegevens zijn ingevoerd. Een video display en een keyboard kunnen ook elk afzonderlijk op de computer zijn aangesloten. Bij een terminal vormen video display en keyboard een eenheid, die als zodanig met de computer is verbonden door middel van een enkele kabel.

Vaak willen we de uitvoergegevens van de computer (zoals de magazijnvoorraad en de salarisadministratie) op papier hebben om nog eens rustig te kunnen nalezen of om iemand ter informatie toe te zenden. Hiervoor wordt een printer gebruikt. Dit is een soort elektronische typemachine, die razendsnel teksten op papier kan afdrukken (in de orde van grootte van 100 tekens per seconde!).

We kunnen niet alle gegevens, die een computer dagelijks verwerkt, in zijn geheugen laten staan. Hiervoor zou dit geheugen immers veel te groot moeten zijn. Om dit probleem op te lossen, maken we gebruik van zogenaamde achtergrondgeheugens. Zo'n achtergrondgeheugen is bijvoorbeeld een disk unit (schijfveeneenheid). Hierin stoppen we een schijf met magnetisch materiaal, waarop we informatie vanuit het computergeheugen kunnen schrijven (net als bij een muziekcassette). Deze informatie kunnen we door de computer laten teruglezen en in zijn eigen geheugen laten opbergen om er mee verder te werken. Programma's en gegevensbestanden worden op disks bewaard. Zodra bepaalde programma's en databestanden nodig zijn om te verwerken, worden ze in het geheugen van de computer geladen. Denk bijvoorbeeld aan de programma's voor het magazijnbeheer, de orderadministratie of de personeelsadministratie en de hierbij benodigde databestanden van de voorraden, de orders en het personeel.

Voor de activiteiten, die door de computer in ons voorbeeld moeten worden uitgevoerd, zijn diverse programma's nodig, die niets met elkaar te maken hebben. Nu kan een computer die slechts één processor bevat, maar één opdracht tegelijk uitvoeren. Dit betekent dat de processor beurtelings een korte tijd met het ene, dan weer met een ander programma bezig is. Hierbij is er natuurlijk een systeem nodig, dat ervoor zorgt dat deze afwisseling correct verloopt. Tevens moet worden voorkomen dat er programma's over elkaar heen worden geschreven in het geheugen en moeten de programma's elk een stukje eigen werkgeheugen krijgen toegewezen, zodat ze niet elkaars gegevens vernietigen. Het systeem dat dit alles regelt, wordt door de computerfabrikant als een standaardsoftwarepakket geleverd en heet operating system.

- Het operating system is de grote baas binnen de computer, die
- zorg draagt voor het laden van programma's in geheugendelen die nog vrij zijn,
 - (vrije) geheugendelen toewijst aan programma's en gegevensbestanden,
 - de werkzaamheden van de processor (of processoren) indeelt,
 - fouten afhandelt,
 - de invoer en uitvoer bestuurt en
 - de gegevens beheert.

Bij het afhandelen van fouten moet u denken aan het zorgen voor de juiste acties als

- er in een programma bijvoorbeeld wordt gedeeld door 0,
- een programma te lang duurt ten gevolge van fouten in dat programma,
- er randapparaten nodig zijn, die echter geen voedingsspanning hebben of
- als er naar een disk unit wordt geschreven, zonder dat er een disk in zit.

U hebt nu gezien welke nuttige dingen een computer zoal kan doen.

Tevens hebt u de belangrijkste peripherals (randapparaten) leren kennen: terminal, keyboard, video display unit, printer, disk unit.

Nu zal de architectuur van de computer zelf behandeld worden.

3.3 Computerarchitectuur

Zoals in paragraaf 1.3 al is opgemerkt, bestaat elke computer uit drie onderdelen (zie fig. 3.2).

3.3.1 De processor

De processor verwerkt de gegevens. Hij zorgt voor het uitvoeren van de opdrachten, zoals het maken van berekeningen, het doen van vergelijkingen en het binnenhalen en wegsturen van gegevens.

In paragraaf 3.4 wordt de processor uitgebreid behandeld.

3.3.2 Het memory

Het memory (geheugen) bevat de gegevens waarmee de computer zijn werk moet doen. Het computergeheugen noemt men ook wel het werkgeheugen. De computer moet natuurlijk weten wat hij moet doen, waar hij de gegevens moet halen en waar hij de resultaten moet opbergen. Hiertoe moet hij opdrachten (instructies) krijgen. Deze instructies staan achter elkaar in het

werkgeheugen van de computer. De instructies samen worden het programma genoemd.

Het werkgeheugen bestaat uit een aantal elektronische geheugenschakelingen. Elke geheugenschakeling kan tegenwoordig al zo'n 32 kbytes informatie bevatten ($k = \text{kilo} = 1000$, dus $32 \text{ kbytes} = 32.000 \text{ bytes}$; bij computers is 1 k eigenlijk $2^{10} = 1024$, zodat 32 kbytes precies 32.768 bytes zijn). Met een geheugen van 32 kbytes kunnen we 32.768 tekens onthouden. Dit komt ongeveer overeen met de tekst van 10 pagina's van A4-formaat. Overigens gaan er op een disk enkele honderden tot enkele tienduizenden kbytes in plaats van 32 .

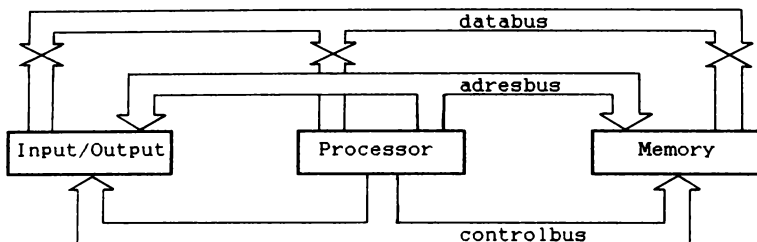


Fig. 3.2 Computerarchitectuur

Het computergeheugen bestaat vrijwel altijd uit een RAM-deel en een ROM-deel. RAM betekent Random Access Memory, een geheugen dat willekeurig toegankelijk is. Dit betekent dat we de gewenste informatie meteen ter beschikking krijgen na het aanbieden van het juiste adres. Dit in tegenstelling tot een Serial Access Memory (SAM), zoals een cassette. Hierbij moet, na het aanbieden van het adres, in het algemeen eerst een stuk band moet worden doorgespoeld of teruggespoeld om dat adres te bereiken.

De eerste halfgeleidergeheugens waren RAM's. Het waren tevens geheugens die gelezen en beschreven kunnen worden, de zogenaamde Read/Write Memories (RWM's). Het nadeel van deze geheugens is dat de informatie verdwenen is, als de voedingsspanning weg is geweest. Later werden er halfgeleidergeheugens ontwikkeld, die de informatie behielden, nadat de voedingsspanning weg was. Deze geheugens kunnen slechts één keer met informatie worden gevuld (door de fabrikant) en zijn daarna niet meer te wijzigen. Ze worden dan ook Read Only Memories (ROM's) genoemd. Ook deze zijn echter random accessible.

Doorgaans gebruikt men de term RAM in plaats van RWM, omdat dit nu eenmaal historisch zo gegroeid is.

Weer later kwamen er PROM's, Programmable Read Only Memories, die door de gebruiker zelf eenmalig te programmeren zijn. Daarna werden er EPROM's ontwikkeld (Erasable PROM's), die de

eigenschap hebben dat ze met behulp van ultraviolet licht kunnen worden gewist en daarna opnieuw geprogrammeerd. Weer later verschijnen de EEPROM's (Electrically Erasable PROM's) of EAROM's (Electrically Alterable ROM's), die elektrisch herprogrammeerbaar zijn. Echter het schrijven gaat hierbij veel langzamer dan het lezen. Bovendien zijn er voor het schrijven hogere spanningen nodig. Momenteel is men druk doende met het ontwikkelen van NOVRAM's (Non-Volatile RAM's). Dit zijn RAM's die hun informatie behouden, nadat de voedingsspanning weggevallen is en waarbij het lezen en schrijven ongeveer even snel gaat en met dezelfde spanningen.

Programma's die steeds in het werkgeheugen van de computer aanwezig moeten zijn, en gegevens die niet veranderen, worden in de ROM gezet. Veranderlijke gegevens worden in de RAM (RWM) gezet. Denk hierbij aan een verkeersregelprogramma, dat eindeloos moet worden uitgevoerd en na een voedingsstoring automatisch moet worden opgestart. Dit programma komt dus in ROM te staan. Daarnaast is er ook RAM nodig om de variabele gegevens omtrent het verkeersaanbod op te slaan.

3.3.3 De Input/Output

De input/output (invoer/uitvoer) bestaat uit elektronische schakelingen, die de processor in staat stellen met zijn omgeving te communiceren. In het voorbeeld van fig. 3.1 bestaat de omgeving van de computer uit terminals, assemblagerobots, disk unit, printer, enzovoorts. We hebben al gezien, dat we de computer moeten meedelen, wat hij voor ons moet gaan doen. We moeten hem bijvoorbeeld vertellen welk programma hij van de disk moet gaan halen en uitvoeren en welke gegevens hij via de printer moet afdrukken. Deze opdrachten (commando's) worden via het keyboard aan de computer medegedeeld. We zien dat de communicatie met keyboard, disk unit, printer en dergelijke, van wezenlijk belang is. Deze communicatie verloopt via de input/output van de computer.

Ook voor de communicatie met de randapparatuur is software nodig. Er moet een programma zijn, dat voortdurend kijkt of er een toets op het keyboard is ingedrukt en dat de ingetoetste informatie binnenhaalt. Dit programma moet vervolgens een beperkt aantal commando's, die via het keyboard worden ingevoerd, herkennen. Verder moeten de gewenste teksten op het display zichtbaar worden gemaakt. De software die voor dit alles verantwoordelijk is, wordt door de computerfabrikant bijgeleverd en heet monitor (niet te verwarren met video display!). De monitor is de standaardprogrammatuur die noodzakelijk is voor het gebruik van de hardware systeemdelen. Het monitorprogramma is een onderdeel van het operating system.

3.3.4 De bussen

Processor, memory en input/output zijn met elkaar verbonden via drie zogenaamde busverbindingen. Hierover gaat het verkeer tussen deze onderdelen.

Adresbus

De adresbus vervoert de adressen. De computer moet middels een adres meedelen, naar welke plek in het geheugen of naar welk randapparaat de gegevens moeten worden gebracht, of waar vandaan de gegevens moeten worden opgehaald. Een adres is een getal, te vergelijken met een huisnummer. Bij processoren is het adres een binair getal, bestaande uit 16 bits (althans bij de MC6809-microprocessor). Met 16 bits kunnen we 64 k plaatsen adresseren. De adresbits zouden na elkaar (serieel) over één adreslijn kunnen worden verstuurd. Dit zou echter vrij veel tijd vergen, vandaar dat de computer 16 adreslijnen bevat. Nu kan het gehele adres in een keer overgestuurd worden. We spreken hier van paralleltransport. De 16 adreslijnen samen vormen de adresbus. Het adres wordt vanaf de processor naar het geheugen of de I/O gestuurd, dus steeds in dezelfde richting. We noemen de adresbus derhalve unidirectioneel. Buiten de computer, waar de verbindingen lang zijn en dus duur, worden gegevens vaak wel na elkaar over één enkele lijn getransporteerd. We spreken dan van serietransport.

Databus

De gegevens zelf worden over de databus vervoerd. De gegevens bestaan meestal uit groepjes van 8 bits. Ook hier worden, net als bij het adres, de databits tegelijk verstuurd. Hiervoor zijn in dit geval 8 datalijnen nodig die samen de databus vormen. Bestaan de getallen uit meer dan 8 bits, dan moeten ze na elkaar in groepjes van 8 bits worden getransporteerd. Het datatransport over deze bus gaat van processor naar geheugen of I/O en omgekeerd. Er vindt hier dus verkeer in twee richtingen plaats. We zeggen dat de databus bidirectioneel is. Een processor die een 8-bits brede (externe) databus heeft, noemen we een 8-bits processor. Een processor met een 16-bits brede (externe) databus heet derhalve een 16-bits processor. Nu zijn er ook processoren die een uitwendige databus hebben van 8 bits, maar die inwendig kunnen rekenen met 16-bits getallen. Deze processoren, zoals de MC6809, worden aangeduid als 8/16-bits processoren.

Controlbus

De controlbus (besturingsbus) zorgt voor het transport van de besturingssignalen. Zo moet er een signaal zijn, dat meedeelt wanneer er een nieuw adres op de adresbus staat en wanneer de data op de databus geldig is. Dit gebeurt met behulp van een kloksignaal. Verder moet er een signaal zijn, dat aangeeft of de gegevens op de databus naar de processor moeten worden gebracht of er juist vandaan komen. Dit wordt aangegeven met

behulp van het zogenaamde read/write signaal. Als geldt dat $R/\bar{W}=0$, moet er worden geschreven (data van processor naar memory of I/O) en als $R/\bar{W}=1$, moet er worden gelezen (data van memory of I/O naar processor). Er zijn nog meer besturingssignalen, maar daar gaan we nu niet verder op in. De besturingssignalen worden elk over een aparte lijn verzonden. Deze besturingslijnen samen vormen de controlbus.

Nu u de computerarchitectuur kent, wordt de processor, het hart van de computer, behandeld.

3.4 Processorarchitectuur

Het blokschema van een processor wordt gegeven in fig. 3.3. De registers zorgen voor de interne geheugenfunctie van de microprocessor. Ze dienen om de data en de adressen te onthouden.

De besturing zorgt ervoor dat op het juiste moment data en adressen in de registers worden opgeborgen en dat het rekenorgaan opdrachten en gegevens ontvangt.

Het rekenorgaan voert de gewenste bewerkingen uit. Dit zijn rekenkundige berekeningen, logische bewerkingen, vergelijkingen en beslissingen.

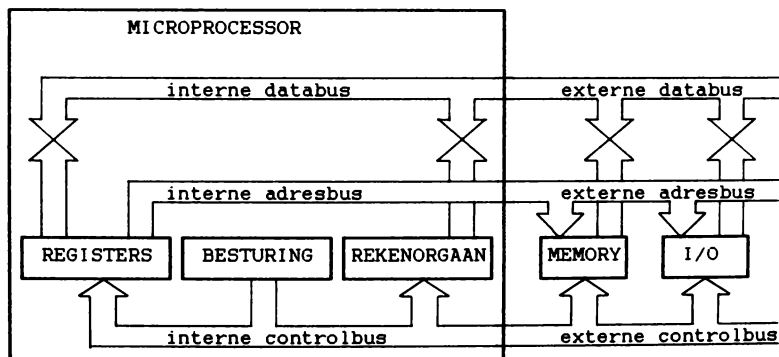


Fig. 3.3 Blokschema van de microprocessor

De drie externe bussen, die u in de vorige paragraaf hebt leren kennen, lopen door in de processor, maar zijn door buffers gescheiden van de inwendige bussen. Deze buffers zijn registers, die de informatie op het juiste moment overnemen en gedurende de benodigde tijd vasthouden. Op deze manier wordt voorkomen, dat de informatie die alleen intern van belang is (bijvoorbeeld tijdens het rekenen), naar memory en I/O gaat (zie fig. 3.4).

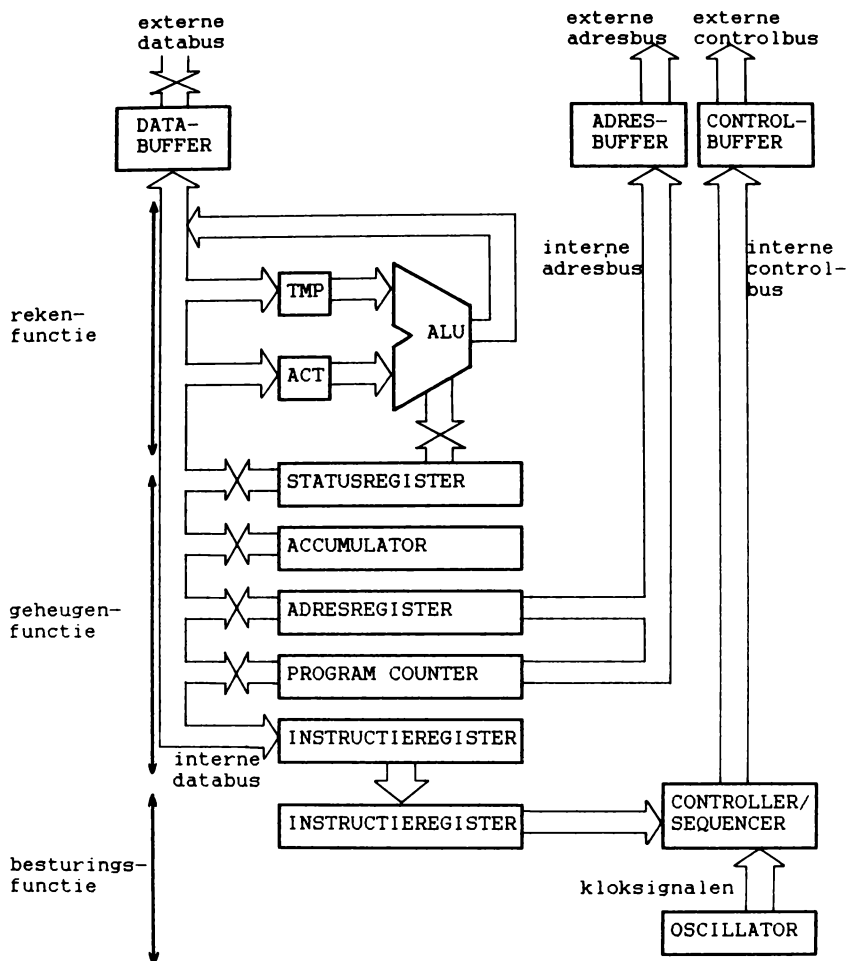


Fig. 3.4 Processorarchitectuur

We zullen eerst de diverse onderdelen van de processor bespreken en vervolgens laten zien hoe de processor een programma uitvoert en hoe de diverse onderdelen van de processor hierin hun functie vervullen.

De ALU (Arithmetic and Logic Unit) is het hart van de processor. Hierin vinden de berekeningen en de logische bewerkingen plaats. Denk bijvoorbeeld aan optellen en aftrekken, aan het vergelijken van twee getallen, aan zetten (1 maken), resetten (0 maken) of invertieren van bits. Dit zijn

nuttige bewerkingen bij het bijhouden van voorraden, bij het openen en sluiten van gaskleppen enzovoorts.

De registers ACT (tijdelijke accumulator) en TMP (tijdelijk register, temporary register) zijn nodig om de gegevens op te slaan die in de ALU worden verwerkt. Denk bijvoorbeeld aan het optellen van twee getallen. We moeten de ingangsgegevens via dezelfde databus, dus na elkaar, aan de ALU toevoeren. Bovendien moet het resultaat van de bewerking via deze databus naar de accumulator worden getransporteerd. De tijdelijke registers ACT en TMP zijn nodig om de gegevens aan de ingang van de ALU gedurende de bewerking te bewaren. Bij processoren die inwendig uitgerust zijn met twee of drie databussen, zou dit niet allemaal nodig zijn. Deze processoren zijn weliswaar sneller, maar ook complexer en vergen meer chipruimte. De meeste processoren maken derhalve gebruik van de één-bus-architectuur.

Het statusregister, ook wel flag register of condition code register genoemd, bevat informatie over de status of toestand van de processor. Er bevinden zich gegevens in omtrent het resultaat van de laatste bewerking. Zo wordt er in aangegeven of de uitkomst van een bewerking 0 is of negatief. Ook wordt er aangegeven of er een carry is opgetreden of dat er een overflow is geweest. (Zie paragraaf 2.7 voor wat betreft de carry en paragraaf 2.11 voor de overflow.) De verschillende bits van het statusregister maken melding van het al dan niet optreden van de diverse condities. Deze bits worden ook wel flags (vlaggen) genoemd. Wordt bijvoorbeeld de carry-flag omhoog gehouden, dan betekent dit dat de carry 1 is. Is de zero-flag gestreken ($Z=0$), dan betekent dit dat de uitkomst ongelijk aan 0 is, enzovoorts.

De accumulator is een opslagregister. Hierin worden gegevens en resultaten opgeslagen voor verdere bewerkingen of voor opslag in het geheugen van de computer.

Het adresregister is het register dat het adres bevat van het gegeven dat voor een bewerking moet worden opgehaald uit het geheugen.

We hebben al gezien dat een programma achter elkaar in het geheugen staat. De program counter of instructieteller houdt bij hoe ver de uitvoering van het programma is gevorderd. Dit betekent dat de program counter het adres bevat van de instructie, die als volgende aan de beurt is. Telkens als er een instructie is opgehaald, wordt de program counter door de processor zelf verhoogd. De program counter wijst als het ware telkens naar de volgende uit te voeren instructie in het geheugen.

Het instructieregister bevat de instructiecode of operatiecode

van de instructie, die voor uitvoering uit het geheugen is opgehaald.

De processor moet nagaan om welke instructie het gaat, met andere woorden hij moet kijken welke bewerkingen er moeten worden uitgevoerd. De instructiedecoder decodeert de instructiecode, die in het instructieregister staat. Dit betekent dat de instructiedecoder nagaat wat er allemaal dient te gaan gebeuren. Hij geeft deze informatie vervolgens door aan de controller/sequencer.

De controller/sequencer zorgt ervoor, dat de diverse onderdelen van de processor de juiste opdrachten krijgen. Hij laat bijvoorbeeld de accumulator weten, wanneer hij informatie van de databus moet opnemen of wanneer hij juist informatie op de databus moet zetten. De ALU wordt meegedeeld of hij moet optellen of een bepaald bit moet resetten of iets dergelijks. De program counter krijgt van de controller/sequencer de opdracht zijn inhoud te verhogen, het adresregister wordt opgedragen zijn inhoud op de adresbus te zetten, enzovoorts. Verder zorgt de controller/sequencer ervoor dat de signalen op de controlbus de juiste waarden hebben. Denk aan het kloksignaal en aan het read/write signaal. De controller/sequencer is dus de grote bestuurder van de informatiestromen in de processor. Opdat alles op de juiste momenten geschiedt, krijgt hij kloksignalen aangeboden door de oscillator.

Nu de functies van de diverse onderdelen van de processor bekend zijn, wordt getoond hoe deze onderdelen hun taak vervullen bij het uitvoeren van een instructie.

3.5 Instructiecycclus

Hoe de processor te werk gaat bij het uitvoeren van een instructie, wordt uitgelegd aan de hand van de instructie ADDA \$332F.

Hierin is \$332F het adres van de geheugenplaats, waarvan de inhoud moet worden opgeteld bij de inhoud van accumulator A.

We zullen in het vervolg de inhoud van een geheugenplaats of de inhoud van een register telkens aangeven door het adres van die geheugenlocatie, respectievelijk de naam van het register, tussen haakjes te plaatsen. We kunnen dus de betekenis van de instructie ADDA \$332F als volgt weergeven:

(A) := (A) + (\$332F)

Het teken "!=" moeten we uitspreken als "wordt gelijk aan". Er staat dus: De inhoud van A wordt gelijk aan de inhoud van A plus de inhoud van de geheugenplaats met adres het \$332F.

Laten we als voorbeeld aannemen dat (A)=\$46 en dat (\$332F)=\$E2. Dit betekent dat door het uitvoeren van de instructie de getallen \$46 en \$E2 worden opgeteld.

ADDA staat natuurlijk als een binaire code in het geheugen van de computer. Deze code is \$BB.

De instructie ADDA \$332F bestaat dus uit 3 bytes, die achter elkaar in het werkgeheugen van de computer staan. Stel dat ADDA (dus \$BB) op de geheugenlocatie met adres \$2A49 staat, dan geldt:

(\$2A49) = \$BB

(\$2A4A) = \$33

(\$2A4B) = \$2F

Als het programma op een gegeven moment is uitgevoerd tot aan de instructie ADDA \$332F, dan zal de program counter PC op dat moment wijzen naar deze instructie. Dit betekent dat de inhoud van de program counter op dat moment gelijk is aan \$2A49. We kunnen dus schrijven:

(PC) = \$2A49

Het uitvoeren van de instructie gaat nu als volgt:

De controller/sequencer laat de program counter zijn inhoud (hier dus \$2A49) op de adresbus te zetten. Hij maakt zelf de read/write-lijn 1. Er moet immers worden gelezen in het geheugen.

We zullen het kloksignaal, dat eveneens door de controller/sequencer op de controlbus wordt gezet en dat voor de juiste timing zorgt, hier verder buiten beschouwing laten. De controller/sequencer geeft aan de adresbuffer en aan de controlbuffer de opdracht de informatie van de interne adresbus, respectievelijk controlbus over te nemen.

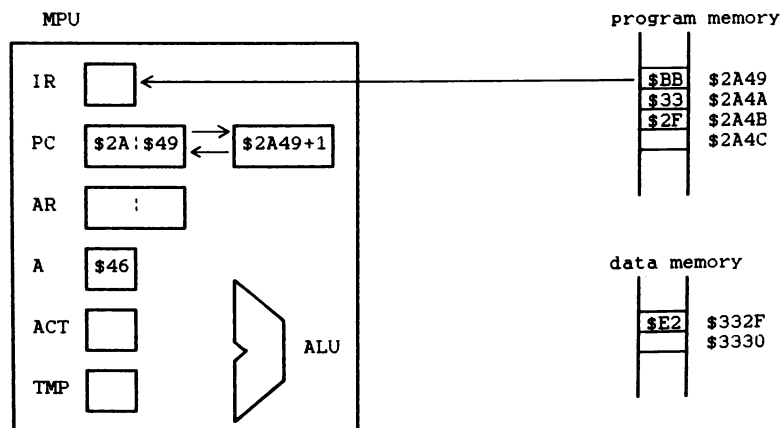


Fig. 3.5 Opcode binnenhalen

Het geheugen (dat is verbonden met de externe processorbussen en dus het adres \$2A49 en $R/\bar{W}=1$ ontvangt) zal nu de inhoud van locatie \$2A49 op de databus zetten. Dit betekent dat de informatie op de databus gelijk wordt aan \$BB (zie fig. 3.5).

Vervolgens geeft de controller/sequencer aan de databuffer de opdracht om de inhoud van de externe databus over te nemen en hij laat het instructieregister IR de informatie op de interne databus overnemen. Daarna geldt dus

(IR) = \$BB

De instructie wordt nu door de instructiedecoder gedecodeerd en deze gedecodeerde informatie wordt aan de controller/sequencer doorgegeven. Deze weet nu dat het om een optelinstruction gaat, waarbij de inhoud van een geheugenlocatie moet worden opgeteld bij de inhoud van accumulator A.

Derhalve geeft de controller/sequencer aan accumulator A alvast de opdracht om zijn inhoud (het getal \$46) via de interne databus in het register ACT te zetten. Het optellen van de twee getallen moet immers in de ALU geschieden.

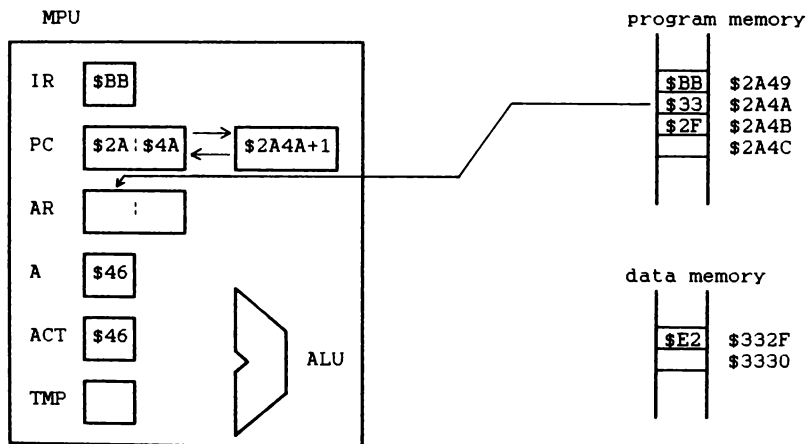


Fig. 3.6 Hoog-adres binnenhalen

Daarna moet eerst het adres van het getal, dat bij A moet worden opgeteld, binnengehaald worden. Hiertoe krijgt de program counter het bevel om zijn inhoud (nu \$2A4A) op de interne adresbus te zetten, waarna de adresbuffer deze informatie doorgeeft aan de externe adresbus. Ook nu geeft de controlbuffer het signaal $R/\bar{W}=1$ door naar buiten. Wederom krijgt de program counter de opdracht zijn inhoud met 1 te verhogen. Deze wordt dus \$2A4B. Het geheugen zet intussen de inhoud van de geheugenlocatie met

adres \$2A4A op de databus. Dit is \$33, het meest-significante deel van het adres van het op te tellen getal. Via de databuffer wordt dit adresdeel (in opdracht van de sequencer/controller) in het meest-significante deel (het zogenaamde hoog-deel) van het adresregister AR gezet (zie fig. 3.6).

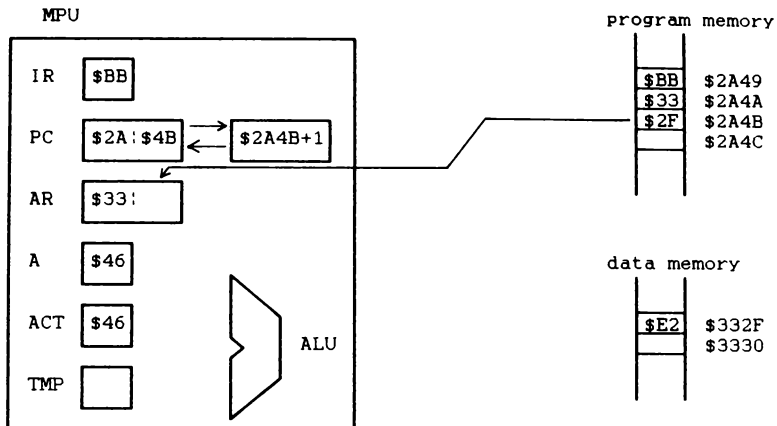


Fig. 3.7 Laag-adres binnenhalen

Vervolgens wordt wederom de inhoud van de program counter (nu \$2A4B), evenals het signaal $R/\bar{W}=1$, via de betreffende buffers aan het geheugen aangeboden.

De program counter verhoogt wederom zijn inhoud met 1 en wordt derhalve gelijk aan \$2A4C.

Het geheugen zet de inhoud van de locatie \$2A4B op de databus. Dit is het minst-significante deel van het adres van de gezochte data. Deze informatie (\$2F) wordt via de databuffer in het laag-deel van het adresregister AR gezet. Er geldt dan:

$$(AR) = \$332F$$

Hiermee hebben we het adres gevonden van het getal, dat bij A moet worden opgeteld (zie fig. 3.7).

Vervolgens moet deze data nog worden binnengehaald.

Hiertoe geeft de controller/sequencer het bevel aan het adresregister om zijn inhoud op de interne adresbus te zetten. Dit adres wordt daarna via de adresbuffer doorgegeven aan het geheugen. Ook nu weer heeft de controller/sequencer ervoor gezorgd dat het read/write signaal 1 is.

Het geheugen zet de inhoud van locatie \$332F op de databus. Deze informatie (het getal \$E2) wordt nu in opdracht van de controller/sequencer via de databuffer in het register TMP geladen (zie fig. 3.8).

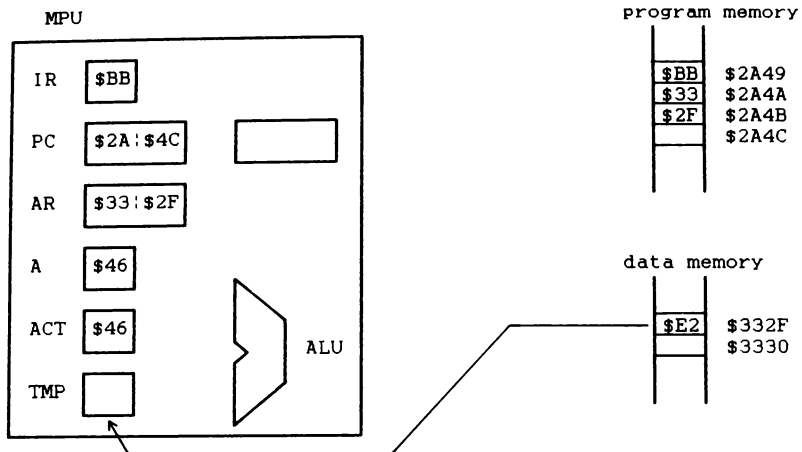


Fig. 3.8 Data binnenhalen

Tenslotte moet de eigenlijke bewerking (het optellen) nog worden uitgevoerd en de uitkomst moet in accumulator A worden geplaatst.

De controller/sequencer geeft nu aan de ALU de opdracht om de optelling uit te voeren (zie fig. 3.9). Het resultaat hiervan ($\$46 + \$E2 = \$28$) wordt daarna in A gezet. Bij deze optelling treedt een carry op, zodat de carry flag in het statusregister wordt gezet. Overflow treedt hier niet op, hetgeen betekent dat de overflow bit in het statusregister 0 wordt gemaakt.

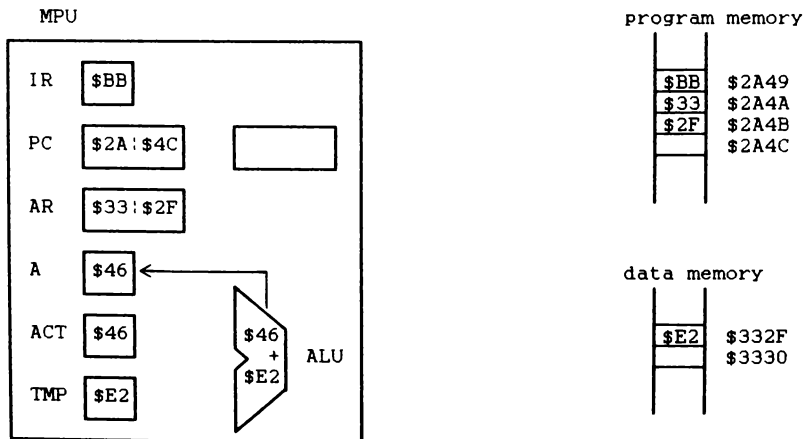


Fig. 3.9 Operatie uitvoeren

Hiermee is de gehele instructie uitgevoerd. De controller/sequencer zal nu de opdracht geven aan de program counter om zijn inhoud op de interne adresbus te zetten. Dit is \$2A4C, het adres van de volgende instructie. Het spel begint nu weer van voren af aan: De instructie wordt opgehaald, gedecodeerd en uitgevoerd. Hierna wordt de daaropvolgende instructie opgehaald, enzovoorts.

U hebt nu geleerd hoe een computer een programma uitvoert. U hebt gezien dat de program counter steeds het adres bevat van de volgende instructiebyte, die moet worden opgehaald. Voorts hebt u gezien welke belangrijke rol de controller/sequencer bij het hele gebeuren speelt. Hij stuurt in feite alle onderdelen van de processor aan en krijgt hiertoe zijn gegevens van de instructiedecoder. Het uitvoeren van de bewerkingen vindt plaats in de ALU. Ten behoeve van het uitvoeren van een instructie werken alle onderdelen van de processor samen onder besturing van de controller/sequencer.

Het ophalen en uitvoeren van een instructie wordt instructiecyclus genoemd.

Het ophalen van de instructiecode (in ons voorbeeld \$BB), het verhogen van de program counter met 1, het plaatsen van de instructiecode in het instructieregister en het decoderen ervan door de instructiedecoder, wordt de fetch cyclus (ophaalcyclus) genoemd.

Het ophalen van de gegevens, nodig voor het uitvoeren van de bewerking, en het uitvoeren van de bewerking wordt executiecyclus genoemd.

Fetch cyclus en executiecyclus samen vormen de instructiecyclus. De instructiecyclus, op deze manier onderverdeeld, wordt de instructiecyclus van Von Neumann genoemd (zie fig. 3.10).

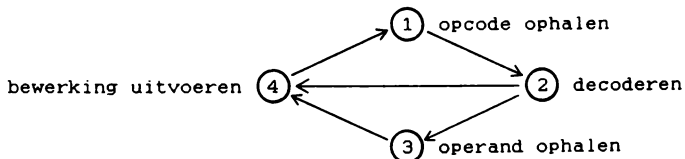


Fig. 3.10 Instructiecyclus van Von Neumann

De fetch cyclus bestaat dus uit de fasen 1 en 2. De opcode (operatiecode) is de instructiecode. Deze geeft aan wat voor bewerking er dient te worden uitgevoerd. In ons voorbeeld was dat het optellen van een getal, waarvan het adres in de instructie gegeven wordt, bij de inhoud van accumulator A.

De executiecyclus bestaat uit de fasen 3 en 4 of alleen uit fase 4. De operand bevat verdere gegevens, die nodig zijn voor het uitvoeren van de instructie. In ons geval is de operand het

adres van het getal, waarmee de bewerking (de optelling bij A) moet worden uitgevoerd. Soms is er geen operand nodig. In dat geval staan alle gegevens, die de processor nodig heeft bij het uitvoeren van een instructie, in de opcode. Dit is bijvoorbeeld het geval bij de instructie CLRA (clear A). Hier moet de inhoud van accumulator A gelijk aan 0 worden gemaakt. Verdere gegevens zijn hierbij vanzelfsprekend niet nodig.

3.6 Programmeermodel

Een aantal registers van de (micro)processor hebt u in de twee vorige paragrafen leren kennen. We zullen nu kennis maken met nog enkele registers. Het gaat in deze paragraaf om de registers die we als programmeur rechtstreeks kunnen gebruiken als hulp bij het uitvoeren van bewerkingen. Deze registers kunnen we als het ware direct aanroepen met behulp van instructies. Ze worden dus bij het programmeren (in machinetaal of in assembly) toegepast. Deze registers worden derhalve samen het programmeermodel van de microprocessor genoemd. Het programmeermodel van de MC6809 wordt gegeven in fig. 3.11.

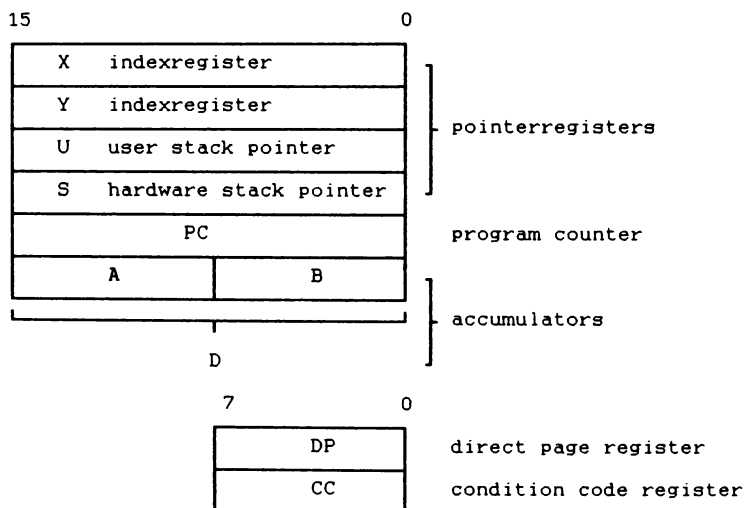


Fig. 3.11 Het programmeermodel van de MC6809

Indexregisters

De indexregisters X en Y zijn identiek in het gebruik. Het zijn registers die handig toegepast kunnen worden bij het aanwijzen van data in het geheugen. Ze worden derhalve ook wel datapointers genoemd. Om data aan te kunnen wijzen, moeten deze registers een adres kunnen bevatten. Ze zijn derhalve 16 bits breed. De adresbus van de MC 6809 bevat immers 16 adreslijnen.

Stack pointers

De registers U en S zijn stack pointers. Ook deze registers wijzen naar data en kunnen derhalve eveneens worden aangeduid als datapointers. De stack pointer wijst naar data die in de stack staat. De stack is een deel van het datageheugen, maar wordt op een speciale manier gebruikt. De stack wordt uitgebreid behandeld in hoofdstuk 7.

Daar U en S een adres bevatten (namelijk een adres van een stacklocatie), zijn U en S 16-bits registers.

U en S zijn ook te gebruiken als indexregisters, terwijl de MC6809 instructies kent waarmee X en Y als stack pointers toe te passen zijn.

Program counter

De functie van de program counter hebt u in paragraaf 3.5 uitgebreid leren kennen. U hebt gezien dat de program counter precies bijhoudt, hoever de uitvoering van het programma is gevorderd. Telkens nadat er een byte van een instructie is opgehaald uit het geheugen, wordt de program counter met 1 verhoogd. Zodoende wijst hij telkens naar de volgende instructiebyte, die moet worden opgehaald. De program counter zouden we derhalve ook kunnen aanduiden met instructiepointer. Daar ook de program counter een adres bevat (namelijk het adres van de volgende op te halen instructiebyte), is ook dit register 16 bits breed.

Accumulators

Accumulator A hebt u in paragraaf 3.5 al leren kennen. De MC6809 heeft echter twee accumulators, namelijk A en B. Dit zijn registers voor de opslag van data. Denk aan de accu van een auto die voor de opslag van lading dient. Aangezien de databus 8 datalijnen bevat, ligt het voor de hand dat de accumulators 8 bits breed zijn. In de accumulators worden met name de uitkomsten opgeslagen van rekenkundige bewerkingen (bijvoorbeeld optellen en aftrekken) en logisch bewerkingen (zoals setten en inverteren).

Om inwendig met 16-bits getallen te kunnen rekenen heeft de MC6809 een 16-bits accumulator D. D is geen extra register, maar bestaat uit de accumulators A en B samen. Hierbij staan in B de bits 0 t/m 7 en in A de bits 8 t/m 15. A is dus de high-byte van D en B is de low-byte.

A en B zijn in het gebruik gelijk. Toch zijn ze niet geheel identiek, omdat er enkele instructies zijn, die wel voor de ene, maar niet voor de andere accumulator mogelijk zijn. U zult dit zien in hoofdstuk 6.

Direct page register

Het direct page register DP kan worden gebruikt bij het adresseren van data. We kunnen het geheugen opgedeeld denken in een aantal pagina's. We kunnen elke pagina adresseren met

behulp van de high-byte van het adres. Op deze wijze kunnen we 256 verschillende pagina's onderscheiden, immers met 8 bits kunnen 256 verschillende combinaties van enen en nullen worden gemaakt. Het adres van een pagina kan in het direct page register worden gezet. Bij het adresseren van een locatie op die pagina kunnen we vervolgens volstaan met het opgeven van de low-byte van het adres. Deze byte wijst dan de gezochte data op die pagina aan. We kunnen per pagina wederom 256 locaties adresseren. In totaal levert dit $256 \cdot 256 = 65.536$ adressen op. Het direct page register is op te vatten als geheugenpagina-pointer.

Condition code register

Tenslotte is er nog het condition code register, ook wel flagregister of statusregister genoemd (zie fig. 3.12). Ook dit register kent u al van de twee vorige paragrafen. Het condition code register bevat een aantal bits (ook wel flags genoemd), die iets zeggen over de eigenschappen van de uitkomst van een bewerking.

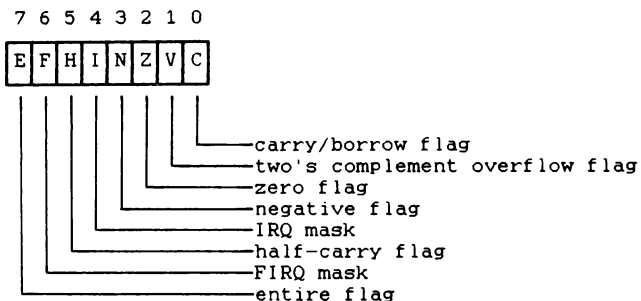


Fig. 3.12 De bits van het condition code register

De carry/borrow bit C is de bit die bij een bewerking niet in het register past. Bij een optelling van bijvoorbeeld twee 8-bits getallen is C de bit die van b7 naar de niet-bestaande b8 van de accumulator (A of B) wordt getransporteerd. Bij aftrekken van twee 16-bits getallen is C de bit die bij de niet-bestaande b16 van de accumulator (D) moet worden geleend. In paragraaf 2.11 is reeds opgemerkt, dat de carry in feite een unsigned overflow is. In hoofdstuk 6 zal blijken, dat de C-bit een iets algemenere betekenis heeft, hetgeen ook al uit de hierboven gegeven definitie valt af te leiden.

De overflow bit V is de two's complement overflow (zie ook paragraaf 2.11). We hebben gezien dat V gelijk aan 1 wordt, als er een tekenfout optreedt bij het optellen of aftrekken van two's complement getallen. In dat geval past de uitkomst niet meer in het register, waarin gewerkt wordt.

De microprocessor weet niet met welk soort getallen we werken, unsigned of two's complement. Hij houdt daarom steeds rekening met beide mogelijkheden. Hiertoe krijgt bit C de waarde die hoort bij de bewerking van unsigned numbers en tegelijkertijd krijgt bit V de waarde behorende bij dezelfde bewerking van two's complement numbers. Bij gebruik van unsigned numbers moet de programmeur bit C bij het resultaat betrekken en bit V buiten beschouwing laten, terwijl hij bij gebruik van two's complement numbers juist op bit V moet letten, waarbij bit C geen betekenis heeft.

De zero flag Z geeft aan of de uitkomst van een bewerking al dan niet gelijk is aan 0. Indien het resultaat 0 is, wordt Z=1 (het nul-zijn is waar). Is het resultaat daarentegen ongelijk aan 0, dan wordt Z=0 (het nul-zijn van de uitkomst is niet waar).

Bit Z is handig bij het nagaan of de uitkomst van een bewerking 0 is. We hoeven immers alleen deze bit (in het condition code register) te testen in plaats van elke bit van de uitkomst (in de accumulator) afzonderlijk.

De negative flag N wordt geset als het resultaat van een bewerking negatief is. In het andere geval wordt N gereset. In feite is N een kopie van de tekenbit van het resultaat (dus van bit 7 van een 8-bits getal, respectievelijk van bit 15 van een 16-bits getal). Bit N zal bij een aantal tests handig kunnen worden gebruikt.

De IRQ-mask I en de FIRQ-mask F, alsmede de entire flag E, hebben te maken met interrupts en zullen derhalve pas in hoofdstuk 9 aan de orde komen.

De half-carry flag krijgt de waarde van de transportbit van b3 naar b4 (dus halverwege een 8-bits getal). Dit is dus de "1-onthouden" tussen de rechter en de linker nibble van een byte. Deze bit wordt alleen aangepast bij het optellen in A of in B, dus niet bij een optelling in D of bij een andere bewerking!!! De half-carry zal belangrijk blijken te zijn bij een BCD-optelling.

3.7 Samenvatting

Een computer bestaat uit:

input/output:	teneinde data binnen te halen en te verzenden
processor:	voor het rekenen en het nemen van beslissingen
geheugen:	om programma's en gegevens op te slaan

Deze onderdelen zijn door bussen met elkaar verbonden:

databus:	voor het vervoer van de eigenlijke gegevens
adresbus:	om geheugen en I/O te adresseren

controlbus: teneinde geheugen en I/O van de juiste
 stuursignalen te voorzien

De processor is opgebouwd uit:

 rekenorgaan: om de gewenste bewerkingen uit te voeren
 registers: voor de tijdelijke opslag van adressen en data
 besturing: teneinde de gewenste acties op het juiste
 moment te laten plaatsvinden

Een instructiecyclus wordt onderverdeeld in:

 fetch cyclus: het ophalen en decoderen van de opcode
 executiecyclus: het (eventuele) ophalen van de data en het
 uitvoeren van de bewerking

De programmeerbare registers van de MC6809 (programmeermodel)
zijn:

 indexregisters X en Y
 stack pointers U en S
 program counter PC
 accumulators A en B (samen D)
 direct page register DP
 condition code register CC

Het condition code register bevat de volgende flags:

E = entire flag
F = FIRQ-mask
H = half-carry
I = IRQ-mask
N = negative bit
Z = zero bit
V = two's complement overflow
C = carry/borrow

3.8 Opgaven

- 1 De MC68000 is een 16/32-bits microprocessor. Wat kunt u van deze microprocessor zeggen met betrekking tot zijn bussen?
- 2 Bepaal de inhoud van het condition code register in de onderstaande gevallen. Ga er telkens van uit dat (CC)=0, voordat de bewerking plaatsvindt.
 - a. $\%1011\ 1100 + \%1110\ 0100$
 - b. $\%0110\ 1100 + \%1010\ 1000$
 - c. $\%0101\ 0101 + \%0110\ 1001$
 - d. $\%0001\ 1110 + \%0101\ 1011$
 - e. $\%1101\ 0001 + \%1001\ 0111$
 - f. $\%0111\ 0010 - \%1000\ 1001$
 - g. $\%1010\ 1100 - \%0100\ 0011$
 - h. $\$DB + \$C9$
 - i. $\$17 - \AF

4 ADRESSERINGS- METHODEN

4.1 Specifieke doelstelling

Nadat u hoofdstuk 4 hebt bestudeerd, zult u tot het volgende in staat zijn:

- * Weten wat de betekenis is van opcode, postbyte en operand.
- * De diverse instructieformaten kennen.
- * Weten wat 1-, 2-, 3- en 4-adres-machines zijn.
- * De adresseringsmethoden van de MC6809 kennen.
- * Weten wat de essentiële verschillen tussen de diverse adresseringsmethoden zijn.
- * De diverse methoden van adresseren toepassen.
- * De postbyte bepalen aan de hand van de documentatie van de fabrikant (appendix B).
- * Uit de postbyte afleiden om welke adresseringsmethode het gaat.
- * De operand bepalen als het effectieve adres is gegeven.
- * Uit de operand het effectieve adres berekenen.

4.2 Instructieformaten

In paragraaf 3.5 hebt u al gezien dat een instructie in principe bestaat uit een opcode en een operand, dus

<instructie> = <opcode> <operand>

De opcode geeft aan wat voor bewerking (operatie) moet worden uitgevoerd. De opcode kan uit een of uit meerdere bytes bestaan. In principe kunnen het een willekeurig aantal bits zijn.

De operand geeft nadere informatie betreffende de bewerking. De operand kan zijn:

- een constant gegeven
(waarmee de bewerking moet worden uitgevoerd)
- een adres
(van het gegeven dat nodig is bij het uitvoeren van de bewerking)
- een code
(die aanvullende informatie bevat voor de instructie)

Ook de operand bestaat uit een aantal bits, meestal een of meerdere bytes.

Er zijn zelfs instructies, waarbij de operand helemaal ontbreekt. We spreken in dit geval van 0-adres-instructies. Hierbij wordt met adres een geheugenadres of een I/O-adres bedoeld. Een register van een microprocessor heeft eigenlijk ook een adres. Dit "adres" maakt deel uit van de opcode en wordt als zodanig door de programmeur niet apart opgegeven. De 0-adres-instructies bestaan alleen uit een opcode. Deze moet in dit geval niet alleen informatie bevatten omtrent wat er moet gebeuren, maar ook waar of waarmee de operatie moet worden uitgevoerd. We komen hier in de volgende paragraaf nog op terug.

Bij de instructies die u tot nu toe hebt gezien, is er slechts één operand. We spreken hier van een 1-adres-machine. Om twee getallen ($G1$ en $G2$), die in het geheugen staan, op te tellen en de uitkomst (S) in het geheugen op te bergen, zijn er bij dit type machines drie instructies nodig. Eerst moet $G1$ worden binnengehaald in de accumulator van de processor. Vervolgens wordt $G2$ opgehaald en opgeteld bij de inhoud van de accumulator (dus bij $G1$). Tenslotte wordt S (vanuit de accumulator) in het geheugen opgeborgen.

Computers die instructies kennen met 2 operands (2 adressen), noemen we 2-adres-machines. Voor het berekenen van $S=G1+G2$ zijn nu slechts twee instructies nodig: één voor het ophalen van $G1$ en $G2$ uit het geheugen en het in de accumulator optellen van deze getallen, en één voor het opbergen van de som S in het geheugen.

Voor dezelfde berekening hebben we slechts één instructie nodig als we een 3-adres-machine gebruiken. Deze machine heeft instructies met 3 operands. De twee eerste operands zijn de adressen van de twee gegevens ($G1$ en $G2$), waarmee de bewerking moet plaatsvinden en de derde operand is het adres van de locatie van het resultaat (S) van de bewerking.

Er zijn ook nog 4-adres-machines. De instructies van zulke machines kunnen 4 adressen bevatten. Het vierde adres is het adres van de volgende instructie. Aparte spronginstructies zijn bij deze machines derhalve niet nodig. Bij een sprong wijst het vierde adres naar de instructie, die als volgende aan de beurt is om uitgevoerd te worden.

Het spreekt vanzelf dat de architectuur van de processor ingewikkelder wordt, naarmate de instructies meer operands kunnen bevatten.

Microprocessors kennen veelal slechts 1-adres-instructies. Strict genomen zouden we moeten zeggen dat ze ook 2-adres-instructies kennen, immers er is een adres nodig om het register van de processor aan te duiden en een adres om de gewenste geheugenlocatie aan te geven. Het adres van de microprocessorregisters is echter opgenomen in de opcode. We

zullen derhalve met een adres de aanduiding van een geheugenlocatie of van een input/output apparaat bedoelen. De MC6809 microprocessor kent alleen 1-adres-instructies. De microprocessors uit de MC68000-serie bezitten ook 2-adres-instructies.

Fig. 4.1 geeft een overzicht van de diverse instructieformaten.

type instructie	formaat
0-adres-instructie	<opcode>
1-adres-instructie	<opcode> <adres>
2-adres-instructie	<opcode> <adres 1> <adres 2>
3-adres-instructie	<opcode> <adres 1> <adres 2> <adres 3>
4-adres-instructie	<opcode><adres 1><adres 2><adres 3><adres 4>

Fig. 4.1 Instructieformaten

De meeste opcodes van de MC6809-instructies bestaan uit één byte. De volgende instructies hebben 2-bytes opcodes, zoals u in appendix B kunt zien: CMPD, CMPS, CMPU, CMPY, LDS, LDY, STS, STY, SWI2, SWI3 en alle long branch instructies, uitgezonderd LBRA en LBSR.

De instructies van de MC6809 bestaan uit 1 tot 5 bytes. U zult deze formaten tegenkomen bij het bestuderen van de adresseringsmethoden van deze microprocessor.

Met adresseren wordt bedoeld

- het aanwijzen van de data, die nodig zijn voor het uitvoeren van een instructie,
- het aanwijzen van een geheugen- of I/O-locatie voor het opbergen van data of
- het aanwijzen van het adres van de volgende instructie (bij sprongen).

De MC6809 kent de volgende adresseringsmethoden:

- inherent of implied
- immediate
- extended (absolute)
- direct
- indexed
 - constant offset indexed
 - accumulator offset indexed
 - auto-increment/decrement
 - program counter relative
- indirect
 - extended indirect
 - indexed indirect
- relative
 - short branch relative
 - long branch relative

4.3 Inherent addressing

De inherent addressing instructie ziet er als volgt uit:
<instructie> = <opcode>

Bij de inherent addressing methode ontbreekt de operand van de instructie. Dit betekent dat de opcode niet alleen zegt wat er moet gebeuren, maar ook waar de bewerking dient te worden uitgevoerd of met welk gegeven. Dit houdt in dat het adres in feite verweven is met de opcode. Vandaar dat deze adresseringsmethode inherent of ook wel implied addressing wordt genoemd. Eigenlijk is er niet van echte adressering sprake, immers er wordt in de instructie niet expliciet een adres opgegeven.

Inherent addressing wordt in hoofdzaak toegepast voor het bewerken van registerinhouden.

Voorbeeld 4.1

CLRA clear A
 maak de inhoud van accumulator A gelijk aan 0

DECB decrement B
 verlaag de inhoud van accumulator B met 1

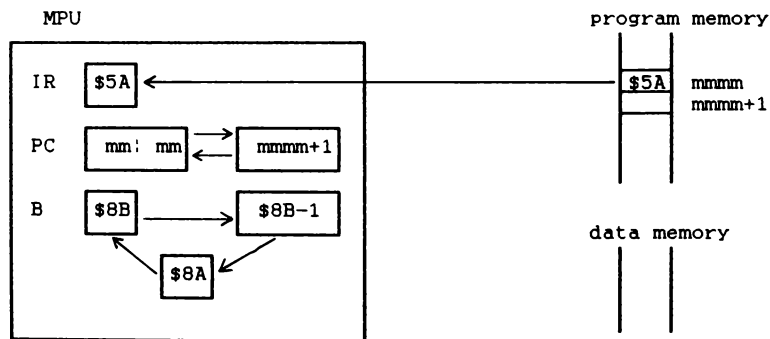


Fig. 4.2 Afhandeling van de inherent addressing instructie DECB

Fig. 4.2 laat zien hoe een inherent addressing instructie wordt uitgevoerd.

Hierin zijn de registers van de microprocessing unit (MPU) aangegeven voor zover zij voor de voorbeeldinstructie van belang zijn. Het condition code register is buiten beschouwing gelaten. \$5A is de opcode van de instructie DECB. In het voorbeeld is aangenomen dat de inhoud van accumulator B aanvankelijk gelijk is aan \$8B. Nadat de instructie DECB is uitgevoerd, zal de inhoud van B gelijk zijn aan \$8A. De program

counter, die telkens naar het volgende op te halen byte uit het programmeergeheugen wijst, is intussen met 1 verhoogd.

Aangezien de inherent addressing instructies meestal betrekking hebben op een van de registers van de microprocessor, spreekt men ook wel van register addressing.

De opcode, en derhalve ook de instructie, bestaat bij de inherent addressing mode uit 1 of 2 bytes.

4.4 Immediate addressing

Bij immediate addressing ziet een instructie er als volgt uit:
<instructie> = <opcode> <operand>

De operand is bij deze adressering

- het gegeven waarmee de bewerking moet worden uitgevoerd, of
- een aanvullende code ten behoeve van de bewerking zelf.

Er is hierbij geen sprake van echte adressering, omdat in de instructie geen adres aanwezig is. De instructie bevat zelf de gegevens, die voor de bewerking nodig zijn. Dit betekent dat de gegevens onmiddellijk door de processor voor gebruik beschikbaar zijn, zodra de instructie is binnengehaald. Men spreekt derhalve wel van een immediate operand.

Is de operand de data, nodig om de operatie mee uit te voeren, dan kan deze 1 of 2 bytes groot zijn. Een register dat één byte kan bevatten, zoals A en B, zal worden gevuld met een 1-byte gegeven. Een register dat 2 bytes groot is, zoals X en Y, zal met een 2-bytes gegeven worden geladen.

Immediate addressing wordt hoofdzakelijk toegepast voor het werken met constanten.

Voorbeeld 4.2

LDA	#\$3C	load A immediate laad A met het getal \$3C (A is een 8-bits register)
LDX	#\$A57B	load X immediate laad X met het getal \$A57B (X is een 16-bits register)
ADDB	#\$FA	add B immediate tel op bij de inhoud van B het getal \$FA (B is een 8-bits register)
SUBD	#\$2284	subtract D immediate verminder de inhoud van D met het getal \$2284 (D is een 16-bits register)
SUBA	##1011	verminder de inhoud van A met het getal 1011
ADDA	#27	tel op bij de inhoud van A het getal 27

CMPU # $\$AA30$ compare U immediate
 vergelijk de inhoud van register U met het
 getal $\$AA30$

Het symbool # (hekje genoemd) is nodig om aan te geven, dat de operand moet worden opgevat als de data, waarmee de instructie wordt uitgevoerd. Het gaat hier dus niet om het adres van de data, waarmee de instructie moet worden uitgevoerd. Het symbool # komt alleen voor bij immediate addressing instructies. Fig. 4.3 laat de afhandeling van een immediate addressing instructie zien.

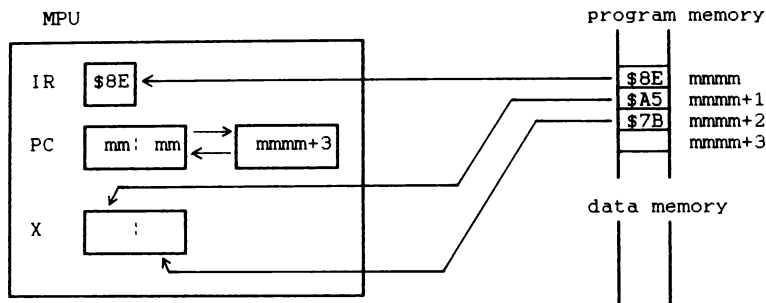


Fig. 4.3 Afhandeling van de immediate addressing instructie
 LDX $\$A57B$

Is de operand de data, die nodig is om de bewerking uit te voeren, dan kan de opcode bij de MC6809 1 of 2 bytes groot zijn. De meeste instructies hebben een opcode van één byte. Er zijn een aantal instructies, die een 2-bytes opcode bezitten (zie paragraaf 4.2).

Daar de operand (die bij immediate addressing de data voorstelt), afhankelijk van de grootte van het betreffende register, uit 1 of 2 bytes bestaat, hebben we te doen met een 2-, 3- of 4-byte instructie.

De operand is een constante. Dit impliceert dat de data in een immediate addressing instructie een constante is en geen variabele.

Moeten we bewerkingen uitvoeren met constante gegevens, dan passen we immediate addressing toe. De constanten plaatsen we direct in de instructie, dus in het programmeergeheugen. Het heeft in het algemeen geen zin om constanten in het datageheugen te zetten, omdat dit alleen extra geheugenruimte kost en extra executietijd. We moeten dan in de instructie immers het adres van de constante opnemen en vervolgens die constante uit het datageheugen ophalen. Bij immediate addressing is de constante meteen beschikbaar, zodra de instructie is opgehaald. Er hoeft niet eerst nog in het datageheugen worden gelezen, hetgeen extra tijd zou kosten.

Het gaat bij data in het programmeergeheugen om constanten, omdat het programma zichzelf niet wijzigt. Het programma is, na het uitvoeren ervan, niet veranderd. Wordt een programma bijvoorbeeld twee keer achter elkaar uitgevoerd, dan is het resultaat in beide gevallen gelijk. Dit zou niet het geval zijn, als de instructies van het programma andere instructies in dat programma zouden wijzigen. Het is zeer onverstandig om programma's te ontwerpen, die zichzelf tijdens het uitvoeren modificeren. Zulke programma's zijn zeer moeilijk te begrijpen en zeer moeilijk aan te passen of uit te breiden in het geval van nieuwe wensen.

Als een programma onveranderd blijft, veranderen de getallen, die in het programma voorkomen, evenmin. Getallen in een programma zijn derhalve steeds hetzelfde, dus constant.

Is de operand een aanvullende code, die nodig is voor het uitvoeren van een bewerking, dan is deze bij de MC6809 altijd één byte groot en de opcode bestaat nu eveneens uit één byte. Het gaat hier altijd om een 2-bytes instructie. Wat moet u zich nu bij deze code voorstellen? Er is bijvoorbeeld een instructie, die de inhoud van een register kopieert in een ander register. De processor hoort te weten, welk het zogenaamde source register is (de bron) en welk register de destination is (de bestemming). Dit wordt aangegeven in de aanvullende code ten behoeve van de opcode. Deze aanvullende code wordt postbyte genoemd (post = na).

Voorbeeld 4.3

TFR A,CC	transfer A to CC verplaats de inhoud van A naar het condition code register; de inhoud van A zelf blijft hierbij ongewijzigd
PSHS A,X,Y,U	push registers on hardware stack zet de inhoud van A, X, Y en U op de hardware stack. (De stack wordt behandeld in hoofdstuk 7.)

De informatie om welke registers het bij deze instructies gaat, vinden we in de postbyte. We komen hier nog op terug in de hoofdstukken 6 en 7.

Samenvattend kunnen we zeggen dat een immediate addressing instructie er als volgt uit kan zien:

```
<instructie> = <opcode> <data>
               (beide 1 of 2 bytes)
               (de instructie telt dus 2, 3 of 4 bytes)
<instructie> = <opcode> <postbyte>
               (1 byte) (1 byte)
               (de instructie telt dus 2 bytes)
```

4.5 Extended (absolute) addressing

Een extended (of absolute) addressing instructie heeft de volgende vorm:

<instructie> = <opcode> <operand>

Bij extended (of absolute) addressing is de operand het absolute of effectieve adres. Dit is het adres van de geheugenlocatie, die nodig is bij het uitvoeren van de betreffende instructie.

Enerzijds kan dit adres nodig zijn om naar toe te springen, teneinde vandaar het programma te vervolgen. Denk hierbij aan de herhaling, die we hebben leren kennen bij het voorbeeld van de CV-regeling in paragraaf 1.5.

Anderzijds kan er een adres nodig zijn om data op te bergen of om de locatie aan te geven waar de benodigde data staat. Er wordt dan door de operand een plaats aangewezen in het (data)geheugen. De informatie die op deze plaats staat, kunnen we ophalen, bijvoorbeeld met behulp van een extended addressing instructie "load" ("laad"). We kunnen echter de inhoud van die geheugenlocatie ook wijzigen met behulp van een "store"-instructie ("berg op"). Dit betekent dat de data bij een extended addressing instructie een variabele is. De data kan immers veranderd worden. De operand (zelf een constante) wijst de data in het geheugen aan en werkt hier als datapointer. De operand is het (absolute of effectieve) adres van de data. Een absoluut adres is te vergelijken met het adres van een persoon, dus straatnaam, huisnummer en woonplaats. Zo'n adres is constant. De bewoner kan variabel zijn.

Een geheugenadres bij een computer bestaat slechts uit een getal. Daar een adres uit 16 bits bestaat, zal de operand een 2-bytes getal zijn. De opcode kan uit 1 of 2 bytes bestaan, zodat de extended addressing instructie uit 3 of 4 bytes bestaat.

Extended addressing wordt gebruikt bij spronginstructies en voor het verwerken van variabelen die zich op vaste geheugenadressen bevinden.

Voorbeeld 4.4

LDA	\$20E6	laad A met de inhoud van de geheugenlocatie met adres \$20E6
STB	\$CC38	zet de inhoud van B op de geheugenlocatie met adres \$CC38
ADDA	\$1F09	tel op bij (A) de inhoud van de geheugenlocatie met adres \$1F09
LDX	\$B064	laad X met de data die staat op de adressen \$B064 en \$B065; bedenk dat X een 16-bits register is en derhalve moet worden gevuld met 2 bytes uit het geheugen; voor de high-byte van X geldt: (XH) := (\$B064); voor de low-byte geldt van X geldt: (XL) := (\$B065)

JMP \$7ACB jump to \$7ACB
 spring naar adres \$7ACB en vervolg daar de
 uitvoering van het programma

Fig. 4.4 demonstreert de afhandeling van een extended (of absolute) addressing instructie.

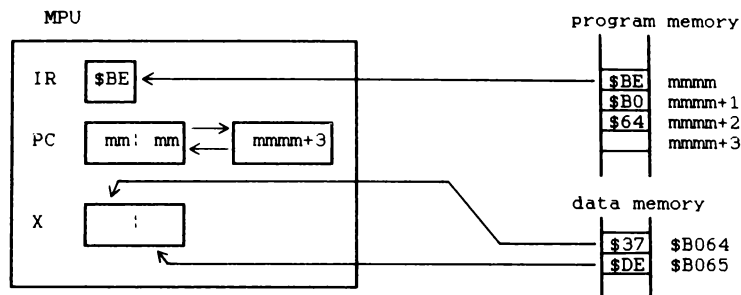


Fig. 4.4 Afhandeling van de extended (absolute) addressing instructie
 LDX \$B064

4.6 Direct (page) addressing

Een direct (page) addressing instructie ziet er als volgt uit:
 <instructie> = <opcode> <operand>

De operand is hier de low-byte van het effectieve adres. De high-byte van het effectieve adres is de inhoud van het direct page register DP. Het direct page register adresseert als het ware de gewenste pagina van het geheugen en de operand geeft aan welke locatie op die pagina bedoeld wordt.

De operand bestaat bij deze adresseringsmethode altijd uit slechts één byte. Samen met de 1- of 2-bytes opcode, hebben we hier te maken met een 2- of 3-bytes instructie.

Direct addressing is bedoeld om op een snelle wijze data te kunnen adresseren.

Direct addressing is een verkorte versie van extended (absolute) addressing. Bij de laatste moet het gehele adres worden gespecificeerd in de instructie, terwijl we bij direct addressing alleen de low-byte in de instructie hoeven op te geven. Een direct addressing instructie heeft dus één locatie minder nodig in het programmeergeheugen en heeft daardoor ook een kortere executietijd. Er wordt immers bij het ophalen van de instructie één keer minder informatie uit het geheugen gehaald. We moeten van tevoren wel zorgen dat het direct page register naar de juiste pagina wijst. Dit kost extra instructies, zodat

direct addressing alleen zin heeft als in het programma een aantal malen achter elkaar dezelfde pagina moet worden geadresseerd. Dan wordt het programma korter en sneller.

Voorbeeld 4.5

ADDD <\$55 LDX <\$AD STB <\$9B JMP <\$83	tel op bij (D) de inhoud van de geheugen-locaties \$55 en \$56 op de pagina die wordt aangewezen door het direct page register laad X met de inhoud van de geheugenplaatsen \$AD en \$AE op de pagina die wordt aangewezen door het direct page register berg op de inhoud van B op locatie \$9B van de pagina die wordt aangewezen door het direct page register spring naar adres \$83 van de pagina, aangewezen door het direct page register en ga daar verder met de uitvoering van het programma
---	---

Het teken "<" wordt gebruikt om aan te geven dat het om direct addressing gaat. Dit is nodig om onderscheid te maken tussen het extended adres, bijvoorbeeld \$BA (= \$00BA) en het direct page adres \$BA. In het laatste geval gaat het om het adres \$BA op de geheugenpagina, die wordt aangewezen door het direct page register.

Fig. 4.5 demonstreert de afhandeling van een direct addressing instructie.

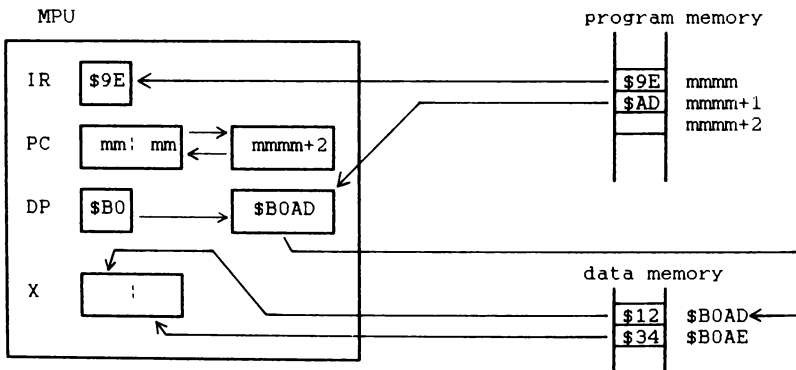


Fig. 4.5 Afhandeling van de direct (page) addressing instructie LDX <\$AD

Vooraf bij programma's die zeer snel moeten zijn en/of die een minimale geheugenruimte mogen gebruiken, is het nuttig zoveel mogelijk direct addressing instructies toe te passen.

Opmerking:

Bij de gegeven voorbeelden zijn steeds hexadecimale getallen gebruikt voor data en voor adressen. Dit is echter niet noodzakelijk. We mogen in de instructies ook decimale, binaire en ASCII waarden toepassen.

4.7 Indexed addressing

4.7.1 Algemeen

De MC6809 kent de volgende indexed addressing modes:

- constant offset indexed
- accumulator offset indexed
- auto-increment/decrement
- program counter relative
- indexed indirect

Indexed addressing wordt gebruikt bij het bewerken van tabellen met data. De tabellen bevinden zich op variabele geheugenlocaties.

Indexed addressing vertoont enige overeenkomst met extended addressing. Bij extended (of absolute) addressing wijst de operand naar de gewenste data in het geheugen. De operand is dus een datapointer. Bij indexed addressing hebben we ook te maken met een datapointer, echter deze is nu een register (een indexregister). Waar bij de extended addressing de operand het effectieve adres bevat (dat derhalve een constante is), zo staat bij indexed addressing het effectieve adres in een indexregister. Daar de inhoud van een register (in tegenstelling tot een operand) veranderd kan worden, is het adres bij indexed addressing een variabele.

Het voordeel van extended addressing is dat we geen register nodig hebben voor de opslag van het adres van de data. Een nadeel is dat het adres een constante is. Dit laatste kunnen we echter ondervangen door gebruik te maken van extended indirect addressing (zie paragraaf 4.8).

Het nadeel van indexed addressing is dat er een register verloren gaat ten behoeve van het adresseren van de data. Vooral als we veel registers nodig hebben voor de opslag van gegevens, is dit erg lastig. Het volgende voorbeeld laat zien dat indexed addressing zeer nuttig is bij het doorlopen van programmalussen.

Voorbeeld 4.6 (variabel adres met indexed addressing)

Stel dat we 200 bytes, die bijvoorbeeld van een disk unit komen, achter elkaar in het geheugen moeten zetten, te beginnen vanaf adres \$1000. Laten we aannemen dat de informatie van de disk unit bereikbaar is via adres \$FF00. Zou nu een adres

alleen een constante kunnen zijn, dan zouden we een erg lang programma nodig hebben om de 200 bytes in het geheugen te zetten. Dit programma zou er ongeveer als volgt uit komen te zien:

```
LDA $FF00 lees byte van disk
STA $1000 berg gelezen data op in datageheugen
LDA $FF00 lees byte van disk
STA $1001 berg gelezen data op in datageheugen
LDA $FF00 lees byte van disk
STA $1002 berg gelezen data op in datageheugen
enzovoorts.
```

Natuurlijk moet er eerst nagegaan worden of de volgende byte van de disk unit al gereed is, voordat deze door de microprocessor gelezen kan worden. Dit betekent dat er aan elke LDA-instructie nog enkele instructies vooraf moeten gaan, zodat het programma nog veel langer wordt.

Door gebruik te maken van indexed addressing kan het programma veel korter worden:

```
LDX #$1000 maak de adrespointer X gelijk aan het
* beginadres ($1000) van het datageheugen
VERDER LDA $FF00 lees byte van disk
STA ,X+ berg data op in datageheugen en verhoog
* de inhoud van de adrespointer met 1 (zodat
* de adrespointer naar de volgende locatie in
* het datageheugen wijst)
JMP VERDER vervolg het programma bij VERDER
```

Het teken "*" aan het begin van een regel geeft aan, dat de betreffende regel geen instructie, maar alleen commentaar bevat.

Natuurlijk moet ook hier gewacht worden op het gereed zijn van de volgende byte van de disk unit. Bovendien moeten we nog een test inbouwen om te stoppen met het lezen van de volgende byte. Als we 200 bytes gelezen hebben, moeten we niet meer terugspringen. Deze beide tests hoeven echter slechts één keer in de programmalus te worden uitgevoerd, zodat het programma tot een tiental instructies beperkt zal blijven.

De registers die we bij indexed addressing kunnen gebruiken, zijn natuurlijk de indexregisters X en Y, maar ook de stack pointers U en S (zoals reeds in paragraaf 3.6 werd opgemerkt) en soms zelfs de program counter PC.

Het algemene formaat van een indexed addressing instructie is als volgt:

<instructie> = <opcode> <postbyte> <operand>

De postbyte zegt onder andere welk register als indexregister

wordt gebruikt bij de instructie. We komen hier nog uitgebreider op terug. De postbyte is altijd aanwezig. De operand daarentegen komt niet altijd voor. Hij bevat een offset welke een two's complement number is. We zullen hierop nog nader ingaan bij de behandeling van de betreffende adresseringsmethode.

4.7.2 Constant offset indexed addressing

De operand is in dit geval een two's complement offset van 0, 5, 8 of 16 bits. De vraag rijst nu onmiddellijk wat de processor hiermee doet. Hij heeft in feite een adres nodig van een geheugenplaats teneinde de instructie, bijvoorbeeld optellen of springen, te kunnen uitvoeren. Er moet immers bekend zijn wat er moet worden opgeteld of waar naartoe moet worden gesprongen. Dit gaat als volgt in zijn werk. Het effectieve adres bij een constant offset indexed addressing instructie is de som van de offset en de inhoud van het aangewezen indexregister. Dit geeft ons de mogelijkheid om een willekeurige byte in een tabel aan te wijzen. Het indexregister bevat het beginadres van de tabel en de offset bepaalt de hoeveelste byte we willen hebben. De offset geeft hier de verplaatsing aan ten opzichte van de eerste byte in de tabel. De offset wordt derhalve ook wel displacement genoemd. De displacement is een constante bij de constant offset indexed addressing instructies.

In de assembly-instructie wordt na de opcode de operand genoteerd, gevolgd door een komma en de naam van het gewenste indexregister (X, Y, U of S).

Voorbeeld 4.7

ADDA	\$25,X	tel op bij (A) de inhoud van de geheugenlocatie met het adres $\$25+(X)$; (X) betekent de inhoud van X)
STB	-25,U	berg op de inhoud van B op de geheugenlocatie met het adres $-25+(U)$
INC	\$A68F,S	verhoog de inhoud van de geheugenlocatie met het adres $\$A68F+(S)$ met 1
LDX	\$AA22,Y	laad X met de inhoud van de geheugenlocaties met de adressen $\$AA22+(Y)$ en $\$AA22+(Y)+1$; bedenkt dat X een 16-bits register is
SUBB	0,U	verminder de inhoud van B met de inhoud van de geheugenlocatie met het adres $0+(U)$
STA	,Y	berg op de inhoud van A op de geheugenlocatie met het adres $0+(Y)$; we mogen 0,Y ook schrijven als ,Y; evenzo mag de offset 0 bij de andere indexregisters worden weggelaten

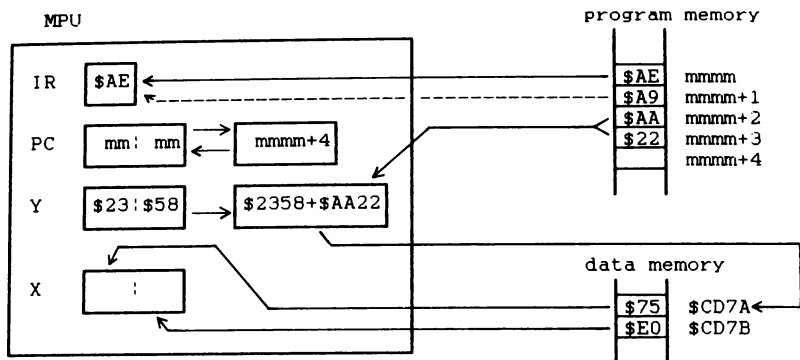


Fig. 4.6 Afhandeling van de constant offset indexed addressing instructie
LDX \$AA22,Y

Fig. 4.6 laat zien hoe een constant offset addressing instructie wordt uitgevoerd. Merk op dat de inhoud van het indexregister Y hierbij zelf niet verandert!

Bij het optellen (en aftrekken) van getallen moeten de tekenbits onder elkaar staan. Hebben de getallen een verschillende lengte, dan moet het kleinste getal naar links worden uitgebreid met de tekenbit, totdat de tekenbits onder elkaar staan. Hierbij moet er eerst voor worden gezorgd, dat de komma's onder elkaar staan, opdat de cijfers met dezelfde gewichten worden opgeteld en niet bijvoorbeeld tweetallen bij achttallen. In paragraaf 2.11 is reeds aangetoond dat we positieve getallen (tekenbit is 0) naar links mogen uitbreiden met een onbeperkt aantal nullen, terwijl we een negatief getal (tekenbit is 1) naar links mogen uitbreiden met enen. Het zal u duidelijk zijn dat een tekenbit niet mag worden opgeteld bij een bit met een bepaald gewicht. Bij het optellen van een 1-byte (2's complement) offset bij een 2-bytes indexregister, wordt de tekenuitbreiding door de microprocessor verzorgd. Dit betekent dat bij de offset \$7B de inhoud van het indexregister wordt verhoogd met \$007B en bij de offset \$93 met \$FF93. \$7B is immers een positief getal, \$93 is negatief.

De operand is een 2-bytes getal als de offset 16 bits bevat en een 1-byte getal als de offset 8 bits bevat. De operand ontbreekt bij een 0-bits of 5-bits offset. In deze twee gevallen is de offset opgenomen in de postbyte. Een constant offset indexed addressing instructie bevat dan ook 2, 3, 4 of 5 bytes. Hoe minder bytes de operand bevat, hoe minder geheugenruimte er nodig is voor de instructie en hoe kleiner de executietijd van de instructie is. Als er geen operand is, behoeft er ook geen operand uit het (programma)geheugen te

worden opgehaald. Bestaat de operand daarentegen uit bijvoorbeeld twee bytes, dan moeten deze twee bytes na elkaar via de databus uit het geheugen worden opgehaald alvorens de processor met het uitvoeren van de instructie kan beginnen. Het ophalen van die 2-bytes operand kost extra tijd.

De postbyte geeft aan welk register als indexregister functioneert en hoeveel bytes de operand bevat. Fig. 4.7 toont de postbyte van deze adresseringsmethode.

mode	postbyte
0-bits offset	1RR0 0100
5-bits offset	0RRn nnnn
8 bits offset	1RR0 1000
16-bits offset	1RR0 1001

nnnnn is de 5-bits offset

RR duidt indexregister aan:

00=X 01=Y

10=U 11=S

Fig. 4.7 Postbyte bij constant offset indexed addressing

We zien hier dat de postbyte uitsluitsel geeft omtrent het aantal bits dat de offset bevat en omtrent welk register als indexregister functioneert.

4.7.3 Accumulator offset indexed addressing

Hier ontbreekt de operand. De (wederom two's complement) offset is hier de inhoud van een van de accumulatoren A, B of D. Welke van deze accumulatoren hiervoor gekozen is, alsmede welk register als indexregister functioneert, wordt vastgelegd in de postbyte (zie fig. 4.8). De inhoud van de accumulatoren kunnen we eenvoudig wijzigen. Dit houdt in dat de offset in dit geval een variabele is.

mode	postbyte
acc. A offset	1RR0 0110
acc. B offset	1RR0 0101
acc. D offset	1RR0 1011

RR duidt indexregister aan:

00=X 01=Y

10=U 11=S

Fig. 4.8 Postbyte bij accumulator offset indexed addressing

Een accumulator offset indexed addressing instructie telt 2 of 3 bytes.

De microprocessor bepaalt het effectieve adres door de inhoud van de aangeduide accumulator op te tellen bij de inhoud van het aangeduide indexregister. De inhoud van accumulator en indexregister veranderen hierbij niet! Ook hier wordt de offset uitgebreid tot een 16-bits getal.

Voorbeeld 4.8

STA D,Y	berg op de inhoud van A op het adres (D)+(Y)
LDX A,U	laad X met de inhoud van de geheugenplaatsen met de adressen (A)+(U) en (A)+(U)+1
ADDB A,X	verhoog de inhoud van B met de inhoud van de geheugenplaats met het adres (A)+(X)

Fig. 4.9 toont hoe de uitvoering van een accumulator offset indexed addressing instructie in zijn werk gaat.

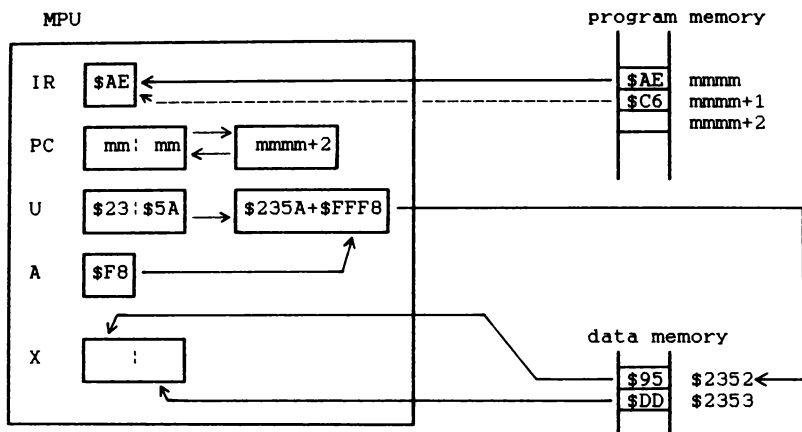


Fig. 4.9 Afhandeling van de accumulator offset indexed addressing instructie
LDX A,U

4.7.4 Auto-increment/decrement indexed addressing

Bij deze vorm van indexed addressing is de offset altijd 0! We hebben wederom geen operand, zodat de instructie 2 of 3 bytes omvat.

Het verschil met de gewone 0-bits offset indexed addressing is dat daar de inhoud van het indexregister niet verandert. Bij auto-increment/decrement addressing wordt het indexregister na de operatie met 1 of met 2 verhoogd, respectievelijk voor de operatie met 1 of met 2 verlaagd. Omdat eerst de bewerking wordt uitgevoerd en pas daarna het indexregister wordt verhoogd, spreekt men van post-increment. Zo spreekt men van pre-decrement, omdat eerst de inhoud van het indexregister wordt verlaagd, alvorens de bewerking wordt uitgevoerd.

Voorbeeld 4.9

LDX ,U++	laad X met de inhoud van de geheugenplaatsen met adres (U) en (U)+1; verhoog daarna (U) met 2
----------	---

STB	,Y+	berg op de inhoud van B op de geheugenplaats met adres (Y) en verhoog daarna (Y) met 1
ADDA	,-X	verlaag de inhoud van X met 1 en tel vervolgens op bij (A) de inhoud van de geheugenplaats met adres (X); let wel, (X) is de verlaagde inhoud van X
SUBD	--S	verlaag (S) met 2 en verminder (D) met de inhoud van de geheugenplaatsen met adres (S) en (S)+1, waarbij (S) de inhoud is van de verlaagde S

De postbyte geeft hier aan welk register als indexregister gekozen wordt, of het om een auto-increment, dan wel om een auto-decrement instructie gaat en of het om een verhoging of verlagings met 1 of met 2 gaat (zie fig. 4.10). Het effectieve adres is gelijk aan de inhoud van het indexregister.

mode	postbyte
increment met 1	1RR0 0000
increment met 2	1RR0 0001
decrement met 1	1RR0 0010
decrement met 2	1RR0 0011

RR duidt indexregister aan:
 00=X 01=Y
 10=U 11=S

Fig. 4.10 Postbyte bij auto-increment/decrement indexed addressing

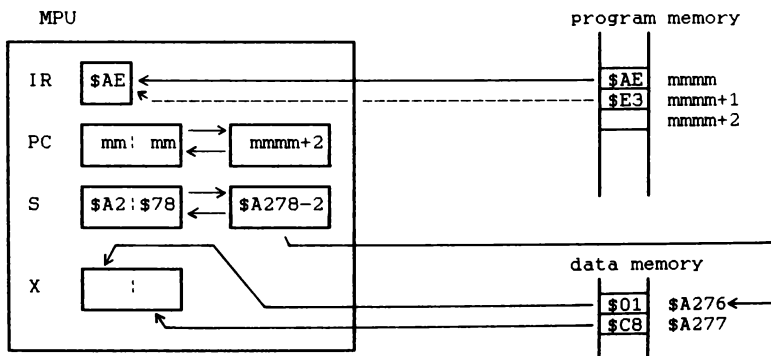


Fig. 4.11 Afhandeling van de auto-decrement indexed addressing instructie
`LDX,--S`

In voorbeeld 4.6 (paragraaf 4.7.1) is reeds gebleken dat auto-increment indexed addressing handig kan worden toegepast bij het maken van programmalussen. Telkens na het opbergen van data in het geheugen, wordt de pointer (het indexregister) verhoogd. Zodoende wijst de pointer naar de volgende locatie in het datageheugen. Vervolgens wordt teruggesprongen in het programma

en wordt opnieuw data binnengehaald van de disk unit. Deze data wordt wederom opgeborgen op de plaats, aangewezen door de pointer. Hierna wordt de pointer weer verhoogd, enzovoorts.

Fig. 4.11 laat een voorbeeld zien van de afhandeling van een auto-decrement (pre-decrement) instructie.

We kunnen de auto-increment/decrement instructies ook handig toepassen als we bijvoorbeeld een tabel van beneden naar boven, dan wel van boven naar beneden moeten doorzoeken. Dit kunnen we doen met behulp van de instructie LDA ,X+ respectievelijk LDA ,-Y in het geval dat we het zoeken byte voor byte doen. X bevat het beginadres van de tabel, Y bevat het eindadres + 1. Doorzoeken we een tabel die bestaat uit getallen van 2 bytes, dan kiezen we bijvoorbeeld de instructies LDD ,X++ respectievelijk LDD ,--Y. Nu is X weer het beginadres van de tabel, echter Y is het eindadres + 2.

4.7.5 Program counter relative addressing

Program counter relative addressing is hetzelfde als constant offset indexed addressing, waarbij de program counter optreedt als indexregister. De offset kan in dit geval echter alleen uit 8 of uit 16 bits bestaan. Let op dat er met de program counter als indexregister geen accumulator offset kan worden gebruikt, noch de auto-increment/decrement addressing!

Voorbeeld 4.10

LDB	55,PC	laad B met de inhoud van de geheugenplaats met adres 55+(PC)
STD	-690,PC	berg op de inhoud van D in de geheugenlocaties met adres -690+(PC) en -690+(PC)+1
SUBA	<\$F9,PC	verminder de inhoud van A met het getal dat staat op adres \$FFF9+(PC)
LDY	\$AFF3,PC	laad Y met de inhoud van de geheugenplaatsen \$AFF3+(PC) en \$AFF3+(PC)+1

Het teken "<" wordt bij indexed addressing gebruikt om aan te geven dat de offset uit één byte bestaat.

Een instructie kan pas worden uitgevoerd, nadat de hele instructie uit het programmeergeheugen is opgehaald. De program counter wijst dan naar de volgende instructie. Let er dus op dat bij program counter relative addressing gerefereerd wordt aan het adres van de volgende instructie!

De program counter relative addressing instructies bestaan uit 3, 4 of 5 bytes. Zie fig. 4.12 voor de postbyte.

Fig. 4.13 laat de afhandeling van een program counter relative addressing instructie zien.

mode	postbyte
8-bits offset	1XX0 1100
16-bits offset	1XX0 1101

X = don't care

Fig. 4.12 Postbyte bij program counter relative addressing

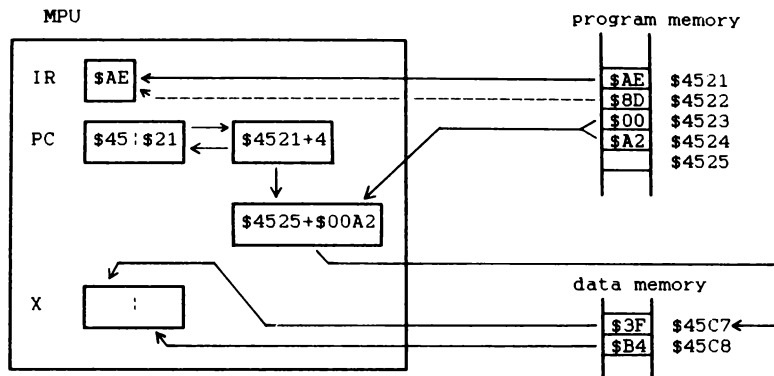


Fig. 4.13 Afhandeling van de program counter relative addressing instructie
LDX \$002A,PC

De program counter relative addressing stelt ons in staat om data op te halen of weg te schrijven op geheugenplaatsen met een bepaalde verplaatsing ten opzichte van de inhoud van de program counter. Met andere woorden we kunnen data bereiken op een plaats met een bepaalde (vaste) afstand tot de instructie in het programma. Dit is nodig bij het schrijven van position independent code.

Met position independent code wordt een programma in machinetaal bedoeld, dat goed functioneert, onafhankelijk van de positie in het geheugen waar het programma geladen wordt. We zullen dit hier nader toelichten.

Fig. 4.14 toont een deel van een programma, dat alleen goed functioneert, indien het geladen is vanaf geheugenlocatie \$2000. De waarde van de variabele TEMP (temperatuur) moet worden vergeleken met een ingestelde waarde REF (bijvoorbeeld 20 graden). Voor de waarden van TEMP en REF zijn twee plaatsen gereserveerd in het geheugen en wel direct achter het programma.

Een label stelt een symbolisch adres voor. Dit wil zeggen dat we een adres weergeven met een naam in plaats van met het echte fysische adres van de geheugenlocatie.

De instructies LDA \$226D en CMPA \$226E (compare A) zijn beide 3-bytes instructies.

adres	label	opcode	operand	commentaar
\$2000		--	--	
.				
\$21A3		LDA	\$226D	vergelijk (\$226D)=TEMP
\$21A6		CMPA	\$226E	met (\$226E)=REF
\$21A9		--	--	
.				
\$226D	TEMP	--		locatie voor temperatuur
\$226E	REF	--		locatie voor referentie

Fig. 4.14 Positie-afhankelijk programma

Dat het programma van fig. 4.14 inderdaad positie-afhankelijk is, dus alleen correct functioneert als het programma is geladen vanaf geheugenplaats \$2000, kunnen we gemakkelijk inzien door het programma vanaf een andere geheugenplaats te laden, bijvoorbeeld vanaf adres \$3000 (zie fig. 4.15). Het kan zijn dat we het programma in een andere computer willen gebruiken, waarbij het geheugen pas begint op adres \$3000.

adres	label	opcode	operand	commentaar
\$3000	--	--		
.				
\$31A3		LDA	\$226D	vergelijk (\$226D)
\$31A6		CMPA	\$226E	met (\$226E)
\$31A9	--	--		
.				
\$326D	TEMP	--		locatie voor temperatuur
\$326E	REF	--		locatie voor referentie

Fig. 4.15 Op verkeerde plaats geladen positie-afhankelijk programma

U ziet dat een positie-afhankelijk programma niet meer goed werkt, als het is geladen op andere geheugenlocaties. In het voorbeeldprogramma, geladen vanaf \$3000, wordt de inhoud van \$226D vergeleken met de inhoud van \$226E. In de plaats daarvan had TEMP (nu de inhoud van \$326D) moeten worden vergeleken met REF (nu de inhoud van \$326E).

Door gebruik te maken van program counter relative addressing kunnen we positie-onafhankelijk programmeren. Immers dan relateren we de adressen aan de program counter, oftewel aan de momentane positie in het programmeergeheugen. Fig. 4.16 toont de positie-onafhankelijke versie van het programma van fig. 4.14. De instructies LDA \$00C6,PC en CMPA \$00C3,PC bevatten elk vier bytes.

Verplaatsen we het positie-onafhankelijke programma van fig. 4.16 naar het geheugengebied, beginnend op adres \$3000, dan blijft het programma correct functioneren (zie fig. 4.17).

adres	label	opcode	operand	commentaar
\$2000		--	--	
.				
\$21A3		LDA	\$00C6.PC	vergelijk de inhoud van
\$21A7		CMPA	\$00C3.PC	\$00C6+(PC) = \$00C6+\$21A7 =
	*			\$226D = TEMP met de inhoud
	*			van \$00C3+(PC) =
	*			\$00C3+\$21AB = \$226E = REF
\$21AB		--	--	
.				
\$226D	TEMP	--		locatie voor temperatuur
\$226E	REF	--		locatie voor referentie

Fig. 4.16 Positie-onafhankelijk programma

adres	label	opcode	operand	commentaar
\$3000		--	--	
.				
\$31A3		LDA	\$00C6.PC	vergelijk de inhoud van
\$31A7		CMPA	\$00C3.PC	\$00C6+(PC) = \$00C6+\$31A7 =
	*			\$326D = TEMP met de inhoud
	*			van \$00C3+(PC) =
	*			\$00C3+\$31AB = \$326E = REF
\$31AB		--	--	
.				
\$326D	TEMP	--		locatie voor temperatuur
\$326E	REF	--		locatie voor referentie

Fig. 4.17 Positie-onafhankelijk programma werkt ook correct in een ander geheugengebied

We zullen hier niet verder op het positie-onafhankelijk programmeren ingaan. Dit komt uitgebreid terug in hoofdstuk 12.

De indexed indirect addressing mode wordt behandeld in de volgende paragraaf. Deze paragraaf handelt over de verschillende vormen van indirecte adressering.

4.8 Indirect addressing

4.8.1 Algemeen

Het formaat van een indirect addressing instructie ziet er als volgt uit:

<instructie> = <opcode> <postbyte> <operand>

Indirecte adressering kan worden toegepast bij het bewerken van gegevens in tabellen. In tegenstelling tot bij indexed addressing zijn hier geen indexregisters nodig.

Indirecte adressering wordt gebruikt voor toegang tot locaties, waarvan de adressen variabel zijn. Zo zijn adressen van randapparatuur veranderlijk. Ze zijn afhankelijk van de hardwareconfiguratie van het computersysteem. Worden deze adressen op vaste geheugenlocaties opgeslagen, dan hoeft de programmeur niet te weten op welke adressen de randapparaten zijn aangesloten. Hij hoeft slechts te weten waar de adressen zijn opgeborgen. Indien dit voor elke configuratie op dezelfde plaats geschiedt, hoeven de programma's niet te worden aangepast, indien deze op een andere machine (met een andere indeling van de hardware) moeten worden uitgevoerd.

U zou verwachten dat de instructies bij deze adresseringsmethode wederom een aparte opcode hebben, teneinde zich te onderscheiden van dezelfde instructies met een andere addressing mode. Dit is bij de MC6809 echter niet het geval. De oorzaak hiervan ligt in het feit dat de 8 bits van de opcode ontoereikend zijn voor het aanduiden van alle combinaties van operatie en adressering. Daar er in de postbyte van de indexed addressing instructies nog ruimte over is, heeft de fabrikant de postbyte niet alleen gebruikt voor het aanduiden van het type indexed addressing, maar ook voor het aangeven van de indirecte adresseringsmethoden. In het andere geval zouden alle indirect addressing instructies twee bytes voor de opcode moeten hebben en bovendien een postbyte. Deze laatste is nodig voor het aangeven van het type indirect addressing. De indirect addressing instructies zouden dan een byte meer bevatten en derhalve ook meer executietijd vergen. Het is dus efficiënter om gebruik te maken van de ruimte, die de postbyte van de indexed addressing instructies biedt. Vandaar dat de opcodes van de indirect addressing instructies van de MC6809 in de data sheets te vinden zijn onder de indexed addressing (zie appendix B). Dit betekent dat de opcode van een instructie met indirecte adressering gelijk is aan die bij dezelfde instructie met indexed addressing.

De postbyte geeft aan of het gaat om indexed, dan wel om indirect addressing en om welk type (in beide gevallen).

Indirecte adressering wil zeggen dat het adres van de gewenste geheugenlocatie niet direct in de instructie voorhanden is (zoals dit wel het geval is bij absolute addressing, waar de operand het adres is).

Absolute adressering geeft een adres aan op een manier, die onafhankelijk is van de plaats waar men zich bevindt. Stel dat de familie Processor woont op het adres Binairlaan 101 in Digitaalstad. Als u aan de genoemde familie een brief stuurt met het genoemde (absolute) adres, maakt het niet uit waar u deze brief post. Hij zal zonder meer bezorgd kunnen worden. De enige voorwaarde hierbij is, dat het adres slechts een keer bestaat. Als u direct het absolute adres van de gewenste locatie opgeeft, spreekt men van directe adressering. Brieven

worden dus op directe wijze geadresseerd.
Extended en direct page addressing behoren tot de directe adresseringsmethoden.

Stel dat u de familie Processor wilt bezoeken, maar u bent de naam van de straat vergeten, als u in Digitaalstad bent aangekomen. U kunt dan ter plaatse aan iemand het adres vragen. De betreffende persoon kent de familie echter niet en verwijst u naar het postkantoor, Decimaalstraat 58, teneinde daar het adres te vragen. In dit geval spreken we van indirecte adressering. Het gewenste adres (van de familie Processor) is te vinden op een ander adres (dat van het postkantoor).

Bij microprocessoren worden twee soorten indirecte adressering onderscheiden:

- register-indirect
Het gewenste adres bevindt zich in een register.
- geheugen-indirect
Het gewenste adres bevindt zich op een geheugenlocatie.

Indexed addressing, zoals u die heeft leren kennen in paragraaf 4.7, valt onder de register-indirecte adressering. Het gewenste adres wordt niet direct opgegeven in de instructie, maar er wordt gezegd in welk (index)register het gewenste adres zich bevindt.

In deze paragraaf wordt de geheugen-indirecte adressering behandeld. We onderscheiden hierbij twee typen:

- extended indirect addressing
- indexed indirect addressing

4.8.2 Extended indirect addressing

Bij extended indirect addressing wijst de operand (het indirecte adres) naar het adres van de data (of naar het adres van de gewenste geheugenlocatie). De operand is bij deze adresseringsmethode een adrespointer. Daar een adres uit 2 bytes bestaat, zullen er achtereenvolgens 2 bytes uit het geheugen moeten worden opgehaald om het absolute adres van de gewenste geheugenplaats te krijgen. De operand (de adrespointer) wijst naar de high-byte van het adres (net zoals een datapointer naar de high-byte van 16-bits data wijst).

Daar de operand bij deze vorm van adresseren een adres is, bestaat deze uit 2 bytes. De postbyte is altijd één byte en de opcode kan weer uit 1 of 2 bytes opgebouwd zijn. Dit betekent dat een extended indirect addressing instructie uit 4 of 5 bytes bestaat.

Er is slechts één soort extended indirect addressing. De postbyte heeft derhalve altijd dezelfde waarde, namelijk \$9F.

Voorbeeld 4.11

```
LDA  [$3421]  laad A met de data die staat op het adres  
              dat wordt aangewezen door $3421  
SUBD  [$AB12]  verminder D met de data die staat op het  
              adres dat wordt aangewezen door $AB12  
LDX   [$4533]  laad X met de data die staat op het adres  
              dat wordt aangewezen door $4533
```

De rechte haken [en] om de operand worden gebruikt om aan te geven dat het hier om indirecte adressering gaat (en dat de operand dus een adrespointer is).

Fig. 4.18 demonstreert de afhandeling van een extended indirect addressing instructie.

Zoals bij absolute addressing de operand naar de data wijst en deze data derhalve als een variabele beschouwd kan worden, zo wijst hier de operand naar een adres, hetgeen betekent dat we het (absolute of effectieve) adres kunnen opvatten als een variabele. We hebben de mogelijkheid het effectieve adres te veranderen.

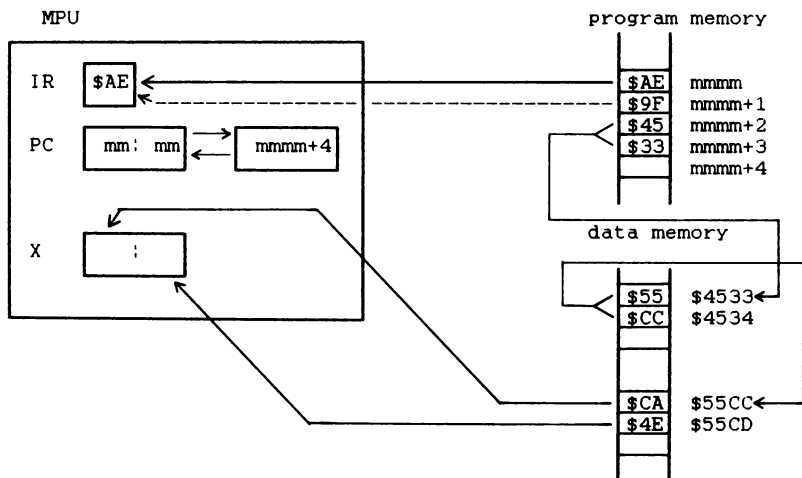


Fig. 4.18 Afhandeling van de extended indirect addressing instructie:
LDX [\$4533]

Indirecte adressering is nuttig in het geval dat er pointers worden gebruikt. Denk aan voorbeeld 4.6 (paragraaf 4.7.1), waar een pointer (indexregister X) werd gebruikt om telkens de juiste geheugenlocatie aan te wijzen voor het opbergen van data. In plaats van een register als pointer te gebruiken, kan ook een geheugenplaats worden toegepast, dankzij de indirecte adressering. In het volgende voorbeeld wordt dit toegelicht.

Voorbeeld 4.12 (Variabel adres met extended indirect addressing)
 We lossen het probleem van voorbeeld 4.6 nu op met behulp van extended indirect addressing:

	LDX	#\$1000	maak de adrepointer (\$0000) gelijk
	STX	\$0000	aan het beginadres (\$1000) van het
			datageheugen
*			
VERDER	LDA	\$FF00	lees byte van disk
	STA	[\$0000]	berg data op in datageheugen
	INC	\$0001	verhoog de inhoud van de low-byte
			van de adrepointer met 1 (zodat
*			de adrepointer wijst naar het
*			volgende adres in het
*			datageheugen)
	JMP	VERDER	vervolg het programma bij VERDER

Bedenk dat een adres uit twee bytes bestaat en dat derhalve een pointer twee geheugenlocaties in beslag neemt. Bij verhogen van de pointer, moet de low-byte worden verhoogd. Deze staat op de locatie na de high-byte in het geheugen, dus op een hoger adres.

Natuurlijk moet ook hier gewacht worden op het gereed zijn van de volgende byte van de disk unit. Bovendien moeten we nog een test inbouwen om te stoppen na het opbergen van de laatste byte. Nadat er 200 bytes gelezen zijn, moet er niet meer worden teruggesprongen.

Door extended indirect addressing toe te passen in plaats van indexed addressing, hebben we een register gespaard. Dit register kan voor andere zaken worden gebruikt.

Indirecte adressering wordt ook vaak toegepast om informatie voor meerdere gebruikers of programma's toegankelijk te maken. De gebruikers hoeven alleen te weten waar zich het adres van de informatie (of van het informatieblok) bevindt. Dit moet dan een vaste geheugenlocatie zijn. Het informatieblok zelf kan op een veranderlijke plaats in het geheugen staan. Bijvoorbeeld kan de locatie afhangen van de grootte van het blok. De grootte van het blok is vaak variabel, omdat het aantal gegevens in vele toepassingen regelmatig verandert. Denk bijvoorbeeld aan een voorraadadministratie of een adressenbestand. Indien het adres verandert, hoeven de programma's niet te worden aangepast. Alleen de inhoud van de pointer (het geheugenadres) moet worden gewijzigd.

4.8.3 Indexed indirect addressing

Net als bij de extended indirect addressing bepaalt de operand hier niet het effectieve adres, maar het adres van het effectieve adres. De operand geeft aan op welke (twee) geheugenlocaties het effectieve adres te vinden is. De operand wijst hierbij (zoals altijd) naar de high-byte van het adres.

De low-byte bevindt zich op de volgende geheugenplaats. Het indexregister bevat nu niet het adres van de data, maar wijst de geheugenlocaties aan, waar het effectieve adres van de data te vinden is. Het indexregister is hier op te vatten als een adrespointer (net als de operand bij een extended indirect addressing instructie). De inhoud van het indexregister kan gewijzigd worden, hetgeen betekent dat de adrespointer een variabele is. Dit is in tegenstelling tot bij de extended indirect addressing, waar de adrespointer een constante is. Het effectieve adres zelf is in beide gevallen variabel, omdat het zich in het datageheugen bevindt.

In principe komen alle indexed adresseringsmethoden ook als indexed indirect addressing voor. Er zijn slechts drie uitzonderingen en wel de 5-bits constant offset, de auto-increment met 1 en de auto-decrement met 1. In de postbyte van de 5-bits offset indexed addressing is geen ruimte om de indirectie aan te geven (zie fig. 4.7 van paragraaf 4.7.2).

De indirect auto-increment/decrement addressing met 1 is onzinnig, omdat bij indirect addressing altijd een adres wordt opgehaald uit het geheugen. Een adres bestaat uit twee bytes, hetgeen betekent dat we altijd twee plaatsen verder of twee plaatsen terug moeten om het volgende, respectievelijk het voorgaande adres op te halen.

Indexed indirect addressing instructies bevatten evenveel bytes als de overeenkomstige niet-indirecte indexed addressing instructies.

In de assembly-instructie wordt de indirectie (net als bij de extended indirect addressing) aangegeven door de operand tussen rechte haken te plaatsen. In de machinetaalinstructie wordt de indirectie in de postbyte aangegeven.

Voorbeeld 4.13

LDB	[-83,X]	laad B met de inhoud van de geheugenplaats, waarvan het adres staat op $-83+(X)$ en $-83+(X)+1$
STA	[D,Y]	berg op de inhoud van A op de geheugenplaats, waarvan het adres staat op $(D)+(Y)$ en $(D)+(Y)+1$
SUBD	[.U++]	verminder D met de data die staat op het adres, dat wordt aangewezen door (U) en vermeerder (U) daarna met 2
LDX	[\$0244,PC]	laad X met de data die staat op het adres, dat wordt aangewezen door $\$0244+(PC)$

Fig. 4.19 laat de afhandeling van een extended indirect addressing instructie zien.

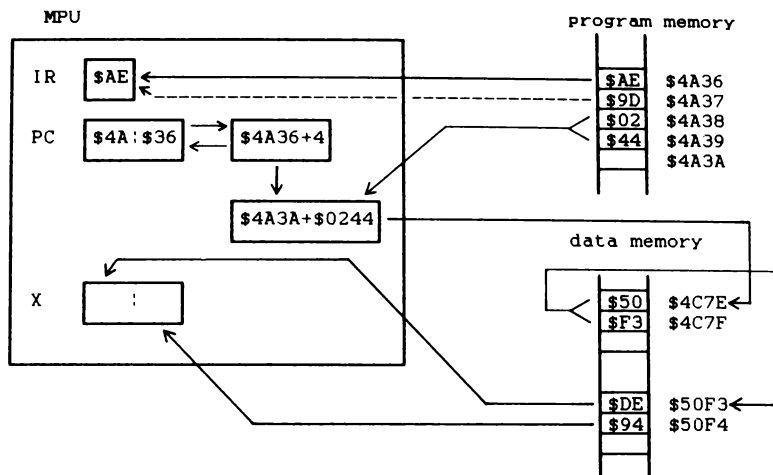


Fig. 4.19 Afhandeling van de indexed indirect addressing instructie (en wel een program counter relative indirect instructie)
LDX [\$0244,PC]

Met behulp van indexed indirect addressing kunnen we data bereiken, waarvan de adressen in een tabel staan. Laden we het indexregister met het beginadres van de tabel, dan bepaalt de offset het hoeveelste adres in de tabel wordt aangewezen. Door middel van de indirectie wordt vervolgens doorverwezen naar de hierbij behorende geheugenplaats.

De postbyte (zie fig. 4.20) geeft behalve de indirectie ook aan om welk type indexed addressing het gaat.

mode	postbyte
0-bits offset	1RR1 0100
8-bits offset	1RR1 1000
16-bits offset	1RR1 1001
acc. A offset	1RR1 0110
acc. B offset	1RR1 0101
acc. D offset	1RR1 1011
auto-incr. met 2	1RR1 0001
auto-decr. met 2	1RR1 0011
8-bits offset van PC	1XX1 1100
16-bits offset van PC	1XX1 1101

RR duidt indexregister aan:

00=X 01=Y
10=U 11=S

X = don't care

Fig. 4.20 Postbyte bij indexed indirect addressing

Vergelijken we de postbytes van de indexed indirect addressing (fig. 4.20) met de postbytes van de niet-indirecte indexed addressing (zie de figuren 4.7, 4.8, 4.10 en 4.12), dan ontdekken we dat bit 4 van de postbyte aangeeft of we al dan niet met indirectie te maken hebben. Bit 4 = 0 betekent dat er geen indirectie is, bit 4 = 1 geeft aan dat er wel indirectie is.

4.9 Relative addressing

De relative addressing instructies hebben het volgende formaat:
<instructie> = <opcode> <operand>

De operand stelt hier een relatief adres voor. Relatieve adressering wordt gebruikt om positie-onafhankelijke sprongen te kunnen uitvoeren (zie ook hoofdstuk 12).

Vraagt u in Digitaalstad waar de familie Processor woont en u krijgt als antwoord "Binairlaan 101", dan is er sprake van absolute adressering. U krijgt steeds hetzelfde antwoord, ongeacht de plaats waar u zich bevindt. Indien u als antwoord krijgt "aan het einde van deze straat links, dan de tweede straat links en daar het zesde huis aan de rechterkant", dan is hier sprake van relatieve adressering. Het adres wordt nu opgegeven in relatie tot de plek waar u staat. Dit houdt in dat het relatieve adres anders zou zijn, als u elders in Digitaalstad gevraagd zou hebben naar de familie Processor.

Het relatieve adres van een geheugenlocatie wordt opgegeven als een verplaatsing ten opzichte van de inhoud van de program counter. Deze adresseringsmethode is derhalve te vergelijken met de program counter relative addressing. De "gewone" relative addressing wordt echter alleen gebruikt bij spronginstructies en wordt daarom ook wel aangeduid als branch relative instructie. De program counter relative instructie kan worden gebruikt voor spronginstructies en voor het aanwijzen van data in het geheugen.

Branch relative instructies stellen ons in staat om op een positie-onafhankelijke wijze te springen naar een ander deel van het programma, teneinde daar met de uitvoering van het programma verder te gaan. We zullen hiervan verderop in deze paragraaf een voorbeeld laten zien (fig. 4.23 en 4.24).

Het effectieve adres van de geheugenlocatie waar naartoe wordt gesprongen, is gelijk aan de som van het relatieve adres (de operand) en de inhoud van de program counter. U moet zich realiseren, dat de inhoud van de program counter gelijk is aan het adres van de volgende instructie. Dit is de instructie die achter de spronginstructie in het geheugen staat. Immers als de

spronginstructie is opgehaald uit het programmeergeheugen, wijst de program counter al weer naar de volgende instructie.

Het relatieve adres is een two's complement getal van 1 of 2 bytes. De opcode bestaat eveneens uit 1 of 2 bytes, zodat een relative addressing instructie uit 2, 3 of 4 bytes bestaat. Let op dat bij het optellen van een 1-byte offset bij de (2-bytes) inhoud van de program counter de offset eerst wordt uitgebreid tot 16 bits, opdat de tekenbit van de offset geheel links komt te staan.

Bij een positieve offset wordt er vooruitgesprongen in het programma, bij een negatieve offset wordt er teruggesprongen.

Bij de MC6809-microprocessor zijn voorwaardelijke sprongen alleen uit te voeren door middel van relative addressing. Een voorwaardelijke sprong is een sprong die alleen wordt uitgevoerd, indien aan een of meerdere voorwaarden is voldaan. Is niet aan de gestelde voorwaarde(n) voldaan, dan wordt de sprong niet uitgevoerd en gaat de uitvoering van het programma verder met de volgende instructie. Bij zo'n spronginstructie zijn er dus twee mogelijkheden:

1. er is aan de conditie(s) voldaan en de sprong wordt derhalve uitgevoerd, of
2. er is niet aan de conditie(s) voldaan en het programma wordt gewoon vervolgd, zonder dat de sprong wordt uitgevoerd.

Dit betekent dat er in het programma als het ware een aftakking ontstaat, zoals duidelijk te zien is in het flowdiagram van fig. 4.21.

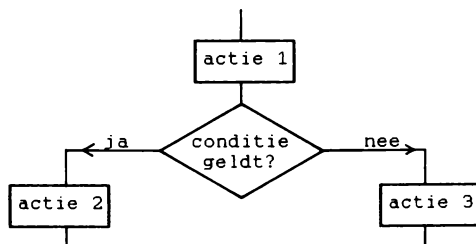


Fig. 4.21 Een conditionele sprong veroorzaakt een aftakking.

Vanwege het ontstaan van een aftakking in de flow van het programma ten gevolge van een voorwaardelijke sprong, wordt zo'n sprong een branch genoemd. Branch betekent tak of zijtak. Elke branch-instructie kan een operand hebben van 1 of van 2 bytes. In het laatste geval spreken we van een long branch, omdat er nu veel verder kan worden gesprongen en wel door het gehele 64k-geheugen. Bedenk hierbij dat op adres \$FFFF adres \$0000 volgt, zodat het hele geheugen kan worden bereikt, hetzij door 32.768 plaatsen terug te springen, hetzij door 32.767 plaatsen vooruit te springen. Met een relatief adres van één byte kunnen er maximaal 128 plaatsen terug of 127 plaatsen vooruit worden gesprongen.

Voorbeeld 4.14

BPL	24	Branch if plus spring naar de geheugenplaats met het adres $24+(PC)$, mits het resultaat (van de vorige bewerking) positief is
BCC	-53	Branch if carry clear spring naar de geheugenlocatie met het adres $-53+(PC)$, indien de carry bit (in het condition code register) 0 is
LBRA	\$5AC2	Long branch always spring naar de geheugenlocatie met het adres $\$5AC2+(PC)$

Fig. 4.22 demonstreert de afhandeling van een relative addressing instructie.

Als de sprong wordt uitgevoerd, wordt de inhoud van de program counter overschreven met de berekende effectieve waarde van het nieuwe adres. Dit heeft tot gevolg dat het programma op dat adres wordt voortgezet. De program counter wijst immers telkens naar de volgende uit te voeren instructie.

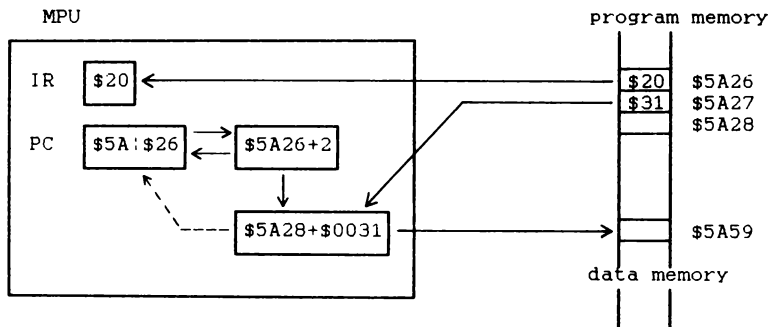


Fig. 4.22 Afhandeling van de relative addressing instructie BRA \$31

adres	label	opcode	operand	commentaar
\$4020	--	--		
...				
\$42B3	*	BRA	\$21	ga naar adres $(PC)+\$21 =$
	*			$\$42B5+\$0021 = \$42D6 =$
\$42B5	--	--		VERDER
...				
\$42D6	VERDER	--	--	
...				

Fig. 4.23 Positie-onafhankelijk springen

Fig. 4.23 laat zien dat een relative addressing (sprong) instructie positie-onafhankelijk is. Wordt hetzelfde programma vanaf adres \$A410 geladen, dan blijft het correct werken (zie fig. 4.24).

adres	label	opcode	operand	commentaar
\$A410		--	--	
.				
\$A6A3		BRA	\$21	ga naar adres (PC)+\$21 =
.	*			\$A6A5+\$0021 = \$A6C6 =
\$A6A5	*	--	--	VERDER
.				
\$A6C6	VERDER	--	--	
.				

Fig. 4.24 Positie-onafhankelijke sprong blijft op andere geheugenlocaties correct werken

Een spronginstructie, die wordt uitgevoerd met extended addressing, is niet positie-onafhankelijk. Hier is de operand het effectieve adres zelf. Verplaatsen we het programma, waardoor ook de instructie waar naartoe moet worden gesprongen, verplaatst wordt, dan wordt er naar het absolute adres gesprongen, dat hoort bij de oorspronkelijke positie van het programma. De te bereiken instructie staat daar na het verplaatsen van het programma niet meer.

We hebben nu alle adresseringsmethoden van de MC6809-microprocessor behandeld. De microprocessor ziet aan de opcode om welke operatie het gaat en welke adresseringsmethode dient te worden toegepast. Hiervoor is er bij sommige instructies een postbyte nodig. De opcode geeft in dat geval aan, dat er nog een postbyte volgt.

U hebt in dit hoofdstuk gezien, dat bijvoorbeeld de instructie LDX verschillende opcodes heeft, afhankelijk van de gekozen addressing mode. De fabrikant heeft twee bits van de opcode gebruikt om de adresseringsmethode aan te geven. Dit impliceert dat er vier verschillende manieren van adresseren kunnen worden weergegeven. Dit zijn bij de MC6809 immediate, direct, indexed en extended (zie appendix B). We missen dan de indirecte adresseringen. Er is al eerder opgemerkt (zie paragraaf 4.8.1), dat de opcode van een indirect addressing instructie gelijk is aan de opcode van de overeenkomstige instructie met indexed addressing. Het onderscheid tussen de diverse soorten wordt hierbij verder gemaakt door de postbyte.

4.10 Samenvatting

Overzicht van de instructieformaten van de MC6809:

```
<opcode>
<opcode><postbyte>
<opcode><operand>
<opcode><postbyte><operand>
```

Overzicht van de adresseringsmethoden van de MC6809:

inherent addressing
 <opcode>

immediate addressing
 <opcode><postbyte> of
 <opcode><data>
 <operand> = <data>
 <data> = constante

extended (absolute) addressing
 <opcode><effectieve adres>
 <operand> = <effectieve adres> = datapointer
 <data> = variabele
 <effectieve adres> = constante

direct (page) addressing
 <opcode><low-address>
 <operand> = <low-address>
 <high-address> = (DP) = variabele
 <low-address> = constante
 <data> = variabele

indexed addressing

1. constant offset
 <opcode><postbyte> (0- en 5-bits offset)
 <opcode><postbyte><offset> (8- en 16-bits offset)
2. accumulator offset (8- en 16-bits offset)
 <opcode><postbyte>
3. auto-increment/decrement (0-bits offset)
 <opcode><postbyte>
4. program counter relative (8- en 16-bits offset)
 <opcode><postbyte><offset>

IX = X, Y, U of S bij 1, 2 en 3 en IX = PC bij 4
offset+(IX) = datapointer = variabele
<effectieve adres>=offset+(IX)
<offset> = (two's complement) displacement
 ten opzichte van (IX)
<offset> = constante bij 1, 3 en 4
 = variabele bij 2
<data> = variabele
<effectieve adres> = variabele

```

indirect addressing
1. extended indirect
    <opcode><postbyte><adres van effectieve adres>
    <operand> = adrepointer = constante
    <data> = variabele
    <effectieve adres> = variabele
2. indexed indirect
    zie bij indexed addressing
    met dien verstande dat
    offset+(IX) = adrepointer = variabele
    <effectieve adres>=(offset+(IX))

relative addressing
    <opcode><relatieve adres>
    <operand> = (two's complement) displacement
               ten opzichte van (PC)
               = relatieve adres = constante
    <effectieve adres> = <relatieve adres>+(PC)

```

4.11 Opgaven

- Geef achter elke van de onderstaande instructies aan om welke adressering het gaat.
 - STA <\$F2
 - LDB #\$44
 - DECA
 - ADDD \$59F4
 - SUBD #\$AB67
 - TFR CC,A
 - STD [\$19DC]
 - JMP <\$C7
 - LDA [\$20FF]
- Bepaal de hexadecimale waarde van de opcode van de onderstaande instructies aan de hand van appendix B.
 - LDA (zowel voor immediate, direct, extended absolute en extended indirect addressing)
 - STD (zowel voor direct, extended absolute en indexed addressing)
 - DECB
 - TFR
- Bepaal het effectieve adres van de constant offset indexed addressing instructies voor de onderstaande gevallen. Ga er telkens van uit dat de inhoud van X gelijk is aan \$A48C.
 - LDA ,X
 - LDA -9,X
 - LDA 20,X
 - LDA \$7A,X
 - LDA \$90,X

- f. LDA \$7104,X
g. LDA \$F300,X
- 4 Bepaal telkens de mode, het indexregister en zo mogelijk de offset van de constant offset indexed addressing instructies, die de volgende postbytes hebben:
- 1110 1000
 - 1000 0100
 - 0011 0111
 - 1110 1001
 - 0100 0110
 - 1000 1001
- 5 Neem telkens aan dat de onderstaande instructies beginnen op adres \$3A42. Verder is gegeven dat (X)=\$2F58, (Y)=\$2064, (U)=\$5C47, (S)=\$8001 en (D)=\$5820. Bereken voor elke instructie het effectieve adres en geef nauwkeurig aan om welke adresseringsmethode het gaat.
- LDA #\$F4 (2-bytes instructie)
 - STD \$EC09 (3-bytes instructie)
 - CMPX <\$A6,Y (3-bytes instructie)
 - LDB ,X+ (2-bytes instructie)
 - ADDD ,--S (2-bytes instructie)
 - BRA \$49 (2-bytes instructie)
 - JMP \$282D,PC (4-bytes instructie)
 - SUBA B,U (2-bytes instructie)
 - LBCC \$8FA2 (4-bytes instructie)
- 6 Gegeven is dat (Y)=\$AA48 en dat de inhoud van elke geheugenlocatie gelijk is aan de low-byte van het adres, dus (\$80A3)=\$A3, (\$CD4F)=\$4F, enzovoorts. Bereken het effectieve adres van de instructies
- LDA [,-Y]
 - STD [,Y++]
 - ADDB [\$25,Y]
 - SUBA [-22,Y]

5 ASSEMBLEREN

5.1 Specifieke doelstelling

Na het bestuderen van deze paragraaf moet u het volgende kunnen:

- * Weten wat assembly is.
- * De functie van assemblers kennen.
- * De betekenis van de vier velden van een assembly-instructie kennen.
- * Weten hoe de operand in assembly moet worden geschreven voor de diverse adresseringsmethoden.
- * De betekenis van de behandelde assembler directives kennen.
- * Deze assembler directives op de juiste manier kunnen gebruiken.
- * De eigenschappen van de verschillende soorten assemblers kennen.

5.2 De functie van assemblers

Nu de diverse adresseringstechnieken, die bij de instructies worden toegepast, bekend zijn, zou u zich kunnen gaan verdiepen in de instructies zelf. Het is echter aan te bevelen om eerst iets meer te weten over assemblers. Assemblers maken het programmeren aanzienlijk eenvoudiger.

Zonder assembler zou u de volgende extra acties zelf moeten ondernemen bij het schrijven van een programma:

- Van elke instructie de juiste opcode opzoeken (in de lijst van instructies, zie appendix B).
- De gevonden opcodes in hexadecimale notatie invoeren in de computer, indien u in het bezit bent van een hexadecimal loader. Bij het ontbreken van een hexadecimal loader, zou u het programma zelfs in binaire code moeten invoeren (zie ook paragraaf 1.6).
- (Aan de hand van appendix B) de juiste postbytes bepalen in geval van indexed addressing en indirect addressing.
- De postbytes van enkele immediate addressing instructies, zoals TFR en PULL, berekenen aan de hand van de documentatie van de microprocessor (zie appendix B).
- Precies bijhouden op welk geheugenadres elke instructie komt te staan, teneinde de adressen te bepalen van de plaats, waar het programma moet worden voortgezet in geval van spronginstructies.
- De relatieve sprongadressen berekenen, zijnde de displacements ten opzichte van de inhoud van de program counter.

- Alle volgende adressen na het toevoegen van instructies aanpassen (als blijkt dat er instructies vergeten zijn).
- Na het tussenvoegen van instructies de nodige adressen van spronginstructies aanpassen.
- Na het tussenvoegen van instructies alle volgende instructies opnieuw intoetsen.

Het zal u duidelijk zijn, dat dit alles een zeer tijdrovende en weinig interessante aangelegenheid is.

Een assembler stelt u in staat om te programmeren in assembly. Dit is een symbolische taal, waarin de instructies niet meer in binaire of hexadecimale code worden ingevoerd, maar in mnemonics, geheugensteuntjes. Mnemonics zijn afkortingen van de omschrijvingen van instructies. We hebben er reeds een aantal leren kennen. Denk bijvoorbeeld aan BRA (BRanch Always, met als opcode \$20), LDA # (LoaD A immediate, met als opcode \$86), enzovoorts.

Deze mnemonics zijn veel gemakkelijker te onthouden dan hexadecimale of binaire codes. Dit betekent dat u niet meer in tabellen hoeft te zoeken bij het programmeren. Ook wordt de kans op vergissingen veel kleiner.

De assembler, een standaard vertaalprogramma, zorgt voor de vertaling van de mnemonics in binaire code. Hij neemt het zoekwerk in de instructietabellen in feite van u over. Het vertalen van assembly naar machinetaal is een vrij eenvoudige zaak. Assembly language is immers een symbolische machinetaal, waarbij elke assembly-instructie met precies één machine-instructie overeen komt.

De assembler stelt u verder in staat om symbolische sprongadressen te gebruiken, alsmede symbolische namen voor variabelen en eventueel constanten. Door naamkaartjes (labels) aan de betreffende instructies, respectievelijk geheugenplaatsen van variabelen te hangen, kan de assembler deze symbolische adressen vertalen in fysische geheugenadressen. U hoeft nu niet meer bij te houden op welke geheugenplaatsen de diverse instructies of variabelen staan. U kunt deze locaties op eenvoudige wijze aangeven met behulp van labels.

Voorbeeld 5.1

```

--      --
      JMP      VERDER
--      --
--      --
VERDER  --      --
--      --
```

VERDER komt als symbool een of meerdere malen voor in de operand van een instructie. De assembler kan dit symbool vertalen in het juiste fysische adres, indien dit symbool ergens in het programma ook als label voorkomt. Een label staat

precies op de plaats die overeenkomt met het fysische adres. Een symbool mag derhalve slechts één keer als label voorkomen, omdat een symbool slechts bij één fysisch adres kan behoren, vanwege de eenduidigheid. Komt een symbool meerdere malen als label voor, dan geeft de assembler een foutmelding, bijvoorbeeld "double defined symbol".

De assembler rekent zelfs de displacements voor u uit bij de relative branch-instructies. U kunt volstaan met het opgeven van een symbolisch adres in de operand, net als bij de JMP-instructie van voorbeeld 5.1. Uit de positie van de bijbehorende label, wordt het relatieve adres berekend. Tenslotte biedt een assembler nog een aantal handige hulpmiddelen bij het programmeren in assembly, zoals u verderop in dit hoofdstuk zult zien.

5.3 Assembly-instructies

5.3.1 Algemeen

De assembler moet tijdens het vertalen van assembly-instructies in machine-instructies kunnen detecteren wat opcodes, operands en labels zijn. Bovendien moet commentaar onderscheiden kunnen worden van instructies. In assemblyprogramma's is het toevoegen van commentaar toegestaan, maar hierover meer in paragraaf 5.3.5.

Om dit te bewerkstelligen is een assembly-instructie verdeeld in vier velden:

1. label
2. opcode
3. operand
4. commentaar

Dit is tevens de volgorde van de velden. De velden worden van elkaar gescheiden door een speciaal symbool, meestal een spatie (<SP>) of een tabulator symbool (<TAB>). Bij gebruik van de tabulator wordt naar de volgende (meestal vooringestelde) kolom gesprongen, zodat de velden keurig netjes recht onder elkaar op het video display verschijnen (zie fig. 5.1). Aan het einde van elke instructie wordt er een return gegeven. We geven dit aan met <CR>, hetgeen staat voor Carriage Return. Dit begrip is ingeburgerd bij het gebruik van typemachines, waarbij deze toets de wagen met het papier doet terugkeren naar zijn beginstand, het begin van een nieuwe regel. De symbolen "<" en ">" worden gebruikt om aan te geven dat het hier om een enkele toets gaat en niet om de twee toetsen C en R apart. Ook begrippen als <opcode> worden wel op deze wijze aangegeven om duidelijk te maken, dat het om een begrip gaat en niet om de opeenvolging van de letters o, p, c, o, d en e.

Een leeg veld wordt aangegeven met behulp van een spatie of door met een <TAB> naar het volgende veld te springen. Niet ieder veld zal bij elke instructie een waarde bevatten.

label	opcode	operand	commentaar
	--	--	
TERUG	ADDA	#1	verhoog teller met 1
	--	--	
	BRA	TERUG	vervolg het programma bij TERUG

Fig. 5.1 De veldindeling van assembly-instructies

5.3.2 Labels

Labels worden gebruikt om te verwijzen naar een bepaalde geheugenplaats. De programmeur hoeft niet te weten om welk fysisch adres het gaat. Dat zoekt de assembler later wel uit. Zoals u reeds gezien hebt, worden labels vaak gebruikt om te verwijzen naar (opcodes van) instructies, waar naartoe moet worden gesprongen vanaf een instructie elders in het programma. Labels kunnen ook worden gebruikt om data aan te wijzen. Zodoende hoeft de programmeur zich niet druk te maken omtrent de juiste fysische locaties waar die data staat of moet komen te staan.

Het gebruik van labels maakt een programma veel beter leesbaar, omdat snel kan worden gezien naar welke plaats een operand verwijst bij sprongen, of omdat de label aanduidt om wat voor soort data het gaat. Dit laatste geldt alleen als de naam voldoende suggestief is gekozen, bijvoorbeeld TKAMER voor de kamertemperatuur.

Bij de meeste assemblers is de lengte van een label begrensd tot bijvoorbeeld 6 karakters. Bovendien mogen niet alle karakters voorkomen in labels. Duidelijk zal zijn dat het symbool <SP> niet in een label mag voorkomen, omdat dit symbool als scheidingstekens van twee velden wordt gebruikt. <SP> geeft aan dat het volgende symbool bij het volgende veld (opcode) behoort en niet bij de label. Vaak wordt als eis gesteld dat een label begint met een letter.

Het verdient aanbeveling alleen letters en cijfers te gebruiken in labels.

Kies geen mnemonic als label. Dit kan verwarrend werken. Kies, ter vergroting van de leesbaarheid van uw programma, suggestieve namen (of afkortingen) voor labels. Tracht zodoende uw programma's zoveel mogelijk zelf-documenterend te maken. Het is duidelijk dat labels als TEMP en DRUK veel suggestiever zijn dan VAR1 en VAR2. TEMP en DRUK suggereren immers dat het om een temperatuur en een druk gaat. VAR1 en VAR2 duiden alleen aan dat het om variabelen gaat, maar geven totaal geen aanwijzing omtrent welke variabelen er bedoeld worden. De programmeur loopt zodoende een veel grotere kans zich te vergissen. Bovendien is het onverstandig labels te kiezen die veel op

elkaar lijken. Ten gevolge van een simpele typefout bij het intoetsen van uw programma zou VAR2 in plaats van VARI ingevoerd kunnen worden. Het programma verloopt dan natuurlijk verkeerd. Zulke fouten zijn vaak moeilijk op te sporen. De label TEMP zal niet gauw veranderen in DRUK ten gevolge van typefouten. Wel kunt u per ongeluk TENP intoetsen. Dit is echter geen probleem, omdat de assembler tijdens het vertalen het symbool TEMP nergens in het programma kan ontdekken als label. Dit betekent dat de assembler niet kan achterhalen door welk fysisch adres het symbool TEMP moet worden vertaald. Ditzelfde geldt als de label wel goed is ingevoerd, maar het symbool (voorkomend in de operand van een instructie) is foutief ingetoetst. De assembler kan een symbool niet vertalen als het symbool zelf of als de bijbehorende label verkeerd is ingetoetst en zal dan een foutmelding geven, bijvoorbeeld "undefined symbol". Gaat een label over in een andere label die eveneens bestaat, dan kan de vertaling wel geschieden en wordt er geen foutmelding gegeven. De label is nu echter vertaald door een ander fysisch adres dan door de programmeur bedoeld was. Het vertaalde programma zal dan niet correct zijn. De kans dat dit gebeurt is echter klein als u symbolen kiest, die weinig op elkaar lijken.

Voor het invoeren van een assemblyprogramma en voor het verbeteren van fouten hebben we een zogenaamde editor nodig. Een editor is een standaardprogramma, dat het mogelijk maakt teksten in de computer in te voeren en te wijzigen. Met behulp van een editor kunnen teksten op een vrij eenvoudige wijze worden veranderd. Er kunnen stukken tekst op willekeurige plaatsen worden toegevoegd en er kunnen willekeurige delen van de tekst worden verwijderd dan wel gewijzigd. Tussenvoegen of weghalen van instructies heeft geen effect op de labels. Denken we dat ons programma goed is ingevoerd, dat laten we het door de editor wegschrijven, bijvoorbeeld op een disk. We verlaten daarna de editor om de assembler aan te roepen, teneinde het programma te vertalen. Deze zorgt er dan voor, dat de symbolen in de juiste (eventueel veranderde) fysische adressen worden vertaald.

5.3.3 Opcode

De betekenis van het veld <opcode> zal duidelijk zijn. Hierin dient de mnemonic te staan van de opcode van de instructie. Deze mnemonics zijn helaas niet gestandaardiseerd. Dit betekent dat ze per type microprocessor kunnen verschillen. Deze verschillen komen vooral tot uiting bij microprocessoren van verschillende fabrikanten. Wat bij de ene fabrikant LD of L (load) heet, wordt bij een andere fabrikant aangeduid met MOV (move). Voor de juiste mnemonics van de MC6809 verwijzen we naar hoofdstuk 6 en appendix B (instructieset). Voeren we (bijvoorbeeld ten gevolge van een typefout) een niet-

bestaande mnemonic in, dan zal de assembler een foutmelding geven, bijvoorbeeld "mnemonic error".

5.3.4 Operand

Het operandveld bevat informatie omtrent de gegevens waarmee de bewerking moet worden uitgevoerd. Hier staan ook aanwijzingen omtrent de adresseringsmethode, die gebruikt dient te worden. Een MC6809-assembler kiest doorgaans voor extended addressing en niet voor direct page addressing, mits dit nadrukkelijk wordt gevraagd. Dit laatste geschiedt door de operand (het adres) te laten voorafgaan door het teken "<". Het teken ">" forceert de extended addressing mode.

Nu is de operand van een direct addressing machinetaalinstructie slechts één byte groot. Alleen de low-byte van het adres wordt opgegeven. De high-byte bevindt zich in het direct page register. Het is raadzaam om in de assembly-instructie het gehele adres op te geven. De assembler doet met de high-byte niets. U moet er zelf voor zorgen dat het direct page register met behulp van voorafgaande instructies wordt gevuld met de juiste waarde, zijnde de high-byte van het gewenste adres oftewel het gewenste page-adres. Het desondanks opgeven van de high-byte in de assembly-instructie verhoogt de leesbaarheid van het programma. U hoeft niet steeds terug te zoeken in het programma om welke geheugenpagina het gaat.

Voorbeeld 5.2

LDA <\$3B64

Hierin ziet u meteen dat A geladen wordt met de inhoud van locatie \$64 op geheugenpagina \$3B. Bij de assembly-instructie

LDA <\$64

ziet u niet om welke pagina het gaat. Hier moet u terugzoeken naar de laatste instructie die het direct page register van informatie voorziet.

De assembler vertaalt beide instructies in \$96 64.

addressing mode	operand
inherent	<operand> ontbreekt
immediate	#<expressie>
extended	<expressie> of ><expressie>
direct	<<expressie>
indexed	<expressie>.R met R = X, Y, U, S, PC of PCR
extended indirect	[<expressie>]
indexed indirect	[<expressie>].R
relative	<expressie>

Fig. 5.2 Operand van assembly-instructies

Fig. 5.2 toont de operand van assembly-instructies als functie van de adresseringsmethode.

De tekens "<" en ">" mogen ook voor de <expressie> van een indexed addressing instructie voorkomen. Het teken "<" forceert dan een 1-byte offset, terwijl het teken ">" een 2-bytes offset forceert.

De operand van een program counter relative instructie kunt u specificeren als <expressie>,PC of als <expressie>,PCR.

De <expressie> is een symbool, een constante of een opeenvolging van symbolen en/of constanten, gescheiden door rekenkundige of logische operatoren.

De toegestane operatoren zijn, ook weer afhankelijk van de assembler die gebruikt wordt:

+	optelling
-	afrekking
*	vermenigvuldiging (unsigned numbers)
/	deling (unsigned numbers)
~	inversie
!	OR
&	AND

Het berekenen van de waarde van de expressie geschiedt van links naar rechts, met dien verstande dat vermenigvuldigen en delen gaat voor optellen en aftrekken. In de expressie mogen haakjes voorkomen. Wat tussen haakjes staat, wordt het eerst uitgerekend.

Een expressie kan het programmeren in assembly vereenvoudigen.

Voorbeeld 5.3

```
LDA    DATA
LDB    DATA+1
```

Hier wordt A geladen met een getal (bijvoorbeeld uit een tabel) en B wordt geladen met het volgende getal (dus met het getal dat op het volgende adres staat). Op deze wijze hoeven niet alle getallen uit een tabel een aparte label te krijgen.

Bij de relative addressing dient het volgende goed in de gaten te worden gehouden. Zoals al gezegd is, berekent de assembler het juiste relatieve adres uit het opgegeven symbolische adres en de locatie van de label met hetzelfde symbool. Dit betekent dat de assembler het symbolische adres interpreteert als een absoluut adres, waar naartoe moet worden gesprongen. Uit de plaats van de spronginstructie en de label berekent de assembler de afstand, de displacement. Dit is het relatieve adres, dat in de operand van de machinetaalinstructie wordt ingevuld.

Geeft u als operand een getal op in een relative branch-instructie in assembly, dan wordt ook dit getal als een absoluut adres opgevat. Ook hier wordt de displacement

uitgerekend en ingevuld in de machinetaalinstructie, zodanig dat tijdens het uitvoeren van het programma inderdaad naar de fysische locatie wordt gesprongen, die aangegeven is door dat getal.

Iets soortgelijks gebeurt bij een program counter relative instructie. Ook hier rekent de assembler de displacement uit tussen het adres van de volgende instructie (zijnde de huidige inhoud van de program counter tijdens het uitvoeren van het programma) en de locatie van de opgegeven label, of dit nu een symbolische naam is of een getal. Wederom wordt de operand in beide gevallen opgevat als een absoluut adres. De assembler vervangt dit in de machinetaalinstructie door het bijbehorende relatieve adres. U hoeft zelf het relatieve adres niet meer uit te rekenen. Dit doet de assembler. In een assembly-instructie kunt u de displacement niet eens meer opgeven, omdat deze door de assembler zal worden opgevat als het absolute adres en vertaald in de bijbehorende displacement.

Voorbeeld 5.4

adres	assembly		machinetaal		
---	---	---	---	---	
\$0042	BRA	\$7A	\$20	\$36	spring naar het adres (PC)+\$36 = \$0044+\$0036 = \$007A
\$0044	---	---	---	---	
---	---	---	---	---	
---	---	---	---	---	
\$007A	---	---	---	---	
---	---	---	---	---	

Let op dat de instructie BRA \$7A twee bytes ruimte vergt, zodat de inhoud van de program counter na het ophalen van deze instructie gelijk is aan \$0044!

Voorbeeld 5.5

adres	assembly		machinetaal		
---	---	---	---	---	
\$2000	LDA	\$10,PC	\$A68D	\$E00C	(A) := (\$E00C+(PC)) = = (\$E00C+\$2004) = (\$0010)
\$2004	---	---	---	---	
---	---	---	---	---	

Hier wordt accumulator A geladen met het getal dat staat op geheugenlocatie \$0010 en niet met het getal op de locatie met het adres \$2014. De assembler berekent de displacement tussen de inhoud van de program counter en geheugenplaats \$0010. De instructie LDA \$10,PC kost vier bytes: opcode=\$A6, postbyte=\$8D en displacement=\$E00C.

Het is overigens, met het oog op de leesbaarheid van uw programma's, niet verstandig om fysische adressen op te geven.

Het gaat om de variabelen zelf en niet om hun plaats in het geheugen.

Het is verder belangrijk u te realiseren, dat een expressie in de operand van een assembly-instructie wordt uitgerekend tijdens het assembleren en niet tijdens het uitvoeren van het programma!

Voorbeeld 5.6

```
LDA    #20
STA    NUMMER
LDB    TABEL+NUMMER
```

U zou hier ten onrechte kunnen denken, dat B wordt geladen met het eenentwintigste getal uit een tabel, waarbij het eerste getal op adres TABEL staat. Dit is echter niet het geval, immers de assembler bepaalt tijdens het assembleren de waarde van de expressie. TABEL en NUMMER zijn beiden symbolische adressen. Via gelijknamige labels wordt er door de assembler een getalwaarde (van een fysisch adres) aan toegekend, bijvoorbeeld TABEL=\$2A01 en NUMMER=\$3000. Dit betekent dat TABEL+NUMMER de som is van de adressen TABEL en NUMMER (dus \$2A01+\$3000) en niet de som van het adres TABEL en de inhoud van de geheugenplaats met als adres NUMMER (zijnde \$2A01+\$0014). Bovendien is de inhoud van de geheugenplaats NUMMER tijdens het assembleren nog ongedefinieerd. De geheugenplaats met het adres NUMMER krijgt de waarde 20 pas nadat de instructies LDA #20 en STA \$3000 zijn uitgevoerd. Tijdens het assembleren wordt er alleen vertaald, er worden geen instructies uitgevoerd!

5.3.5 Commentaar

Het commentaarveld is een veld, waarin u willekeurige teksten kunt plaatsen. Het commentaarveld loopt tot de <CR>. De assembler slaat dit veld gewoon over. Het is bedoeld om opmerkingen te plaatsen ter verduidelijking van het programma. Voor de microprocessor is het volkomen nutteloos. Voor de programmeur is het echter van onschatbare waarde, mits op de juiste wijze gebruikt.

Er moet op de eerste plaats zinvol commentaar bij de instructies of groepjes van instructies worden gegeven. "Maak CONTROL2 gelijk aan 1" is geen zinvol commentaar. Wel zinvol is het te vermelden "Ontsteek de hoofdblander". Zo ook is de mededeling "Stop de waterpomp" zinvol commentaar in tegenstelling tot "Reset CONTROL1".

Naast het feit dat commentaar zinvol moet zijn, moet het zo kort mogelijk en ter zake zijn.

Gebruik geen duistere afkortingen en zorg voor begrijpelijk commentaar.

Commentaar is essentieel voor het zelf-documenterend maken van een programma. Hierdoor is een programma beter onderhoudbaar. In de praktijk blijken programma's nogal eens te moeten worden aangepast ten gevolge van gewijzigde omstandigheden, gewijzigde eisen of gebleken onvolkomenheden. Is een programma slecht gedocumenteerd, dan is dit wijzigen een haast onmogelijke zaak. Vaak kan er dan beter helemaal opnieuw worden begonnen, hetgeen vanzelfsprekend een dure aangelegenheid is. Goede documentatie is derhalve van wezenlijk belang.

Ook een goede modulaire opbouw van een programma is essentieel. Door de opbouw in modules hoeven vaak slechts een of enkele modules te worden gewijzigd, terwijl de rest van het programma onaangetast blijft. Men hoeft dan niet bij elke (soms kleine) wijziging het gehele programma te doorlopen om te controleren of die wijziging mogelijk nog consequenties heeft op andere plaatsen in het programma. Ook is het zaak om de modules zeer goed te documenteren, zodat bekend is wat de module precies doet, waar de ingangsgegevens voor de module moeten staan en waar de uitgangsgegevens komen te staan. Hiertoe moet er commentaar worden opgenomen aan het begin van elke module, alsook aan het begin van het totale programma. Zou dit in het commentaarveld moeten geschieden, dan gaat er veel ruimte verloren, immers de eerste drie velden moeten hier worden overgeslagen. Vandaar dat in assembly de gehele regel voor commentaar gebruikt kan worden.

Nu moet de assembler natuurlijk wel de mogelijkheid hebben om commentaar te onderscheiden van labels. Dit gebeurt door in de eerste kolom een speciaal teken op te nemen, dat aangeeft dat er commentaar volgt. Hiervoor wordt vaak het teken "*" gebruikt. Begint een regel met een "*", dan betekent dit, dat de gehele regel commentaar is en dat de assembler deze regel derhalve kan overslaan.

Zo kunnen we een programma en een module steeds beginnen met een aantal regels commentaar.

Aan het begin van een module (evenals aan het begin van een programma) moet het volgende commentaar zijn opgenomen:

1. naam van module of programma
2. naam van de programmeur
3. revisienummer van module of programma
4. datum van de laatste revisie
5. korte functiebeschrijving
6. locatie van de ingangsparameters
7. locatie van de uitgangsparameters
8. welke registerinhouden worden gewijzigd

We zullen in de voorbeelden, die volgen nadat de instructieset is behandeld, laten zien hoe we op de juiste manier commentaar kunnen toepassen.

5.4 Assembler directives

Om een assemblyprogramma op de juiste manier te vertalen, heeft de assembler een aantal aanwijzingen nodig. Deze worden assembler directives genoemd, of ook wel pseudo-instructies. Het zijn een soort instructies, die door de assembler bij het vertalen worden uitgevoerd. Deze pseudo-instructies worden echter zelf niet vertaald en zijn derhalve geen instructies, die door de microprocessor tijdens de executie van het programma worden uitgevoerd. Vandaar de toevoeging "pseudo".

Er zijn ook assembler directives, die niet strict noodzakelijk zijn, maar die wel meehelpen het programmeren eenvoudiger te maken of de listings (de afdrukken) van programma's overzichtelijker.

De mnemonics van de assembler directives behoren in het opcode-veld te staan.

U zult nu kennis maken met de belangrijkste assembler directives. De mnemonics, die hierbij worden gebruikt, zijn afhankelijk van de fabrikant van de microprocessor.

ORG <expressie>

"ORiGiN" vertelt de assembler vanaf welk geheugenadres de volgende gegevens of instructies moeten worden geladen. De waarde van <expressie> geeft aan op welk geheugenadres moet worden begonnen.

Voorbeeld 5.7

```
ORG     $3000
```

De nu volgende gegevens worden geladen te beginnen op adres \$3000.

Door in het programma alleen symbolische adressen toe te passen, kan een programma eenvoudig op een andere geheugenlocatie worden geladen door de ORG-directive te wijzigen en het assemblyprogramma vervolgens opnieuw te vertalen. De assembler zorgt voor de juiste vertaling van de symbolische adressen.

<label> EQU <expressie>

"EQUate" kent aan <label> de waarde van <expressie> toe. EQU-directives worden vooral toegepast om geheugenadressen toe te kennen aan (symbolische) variabelen.

Voorbeeld 5.8

```
SWITCH EQU     $FF00
```

Overal waar de assembler (tijdens het vertalen) het symbool SWITCH tegenkomt, wordt hiervoor het getal \$FF00 ingevuld. In dit voorbeeld betekent dit, dat de waarde van de variabele SWITCH op geheugenplaats \$FF00 is te vinden.

[<label>] FCB <expressie>[,<expressie>,...]

Wat hier tussen de rechte haken staat, mag worden weggelaten. De rechte haken hebben hier niets te maken met indirecte adressering.

"Form Constant Bytes" laat de assembler de waarden van de in de operand opgegeven expressies achtereenvolgens in het geheugen opnemen. Dit gebeurt op de eerstvolgende vrije geheugenlocaties. De assembler houdt tijdens het assembleren precies bij waar hij gebleven is met het invullen van geheugenruimte ten behoeve van programma en gegevens. Het adres van de eerste vrije geheugenlocatie ten behoeve van de reeks data, wordt toegekend aan <label>. FCB-directives worden vooral toegepast om tabellen met constanten te maken.

Voorbeeld 5.9

```
BYTES FCB 35,-20,$A6,$B,'c
      FCB 0,%101,$33
```

Zijn er na ORG \$3000 al 10 geheugenplaatsen gevuld (dus de plaatsen \$3000 t/m \$3009), dan zal de assembler aan de label BYTES de waarde \$300A toekennen. Voorts zal op locatie \$300A het getal 35 = \$23 worden gezet, op locatie \$300B komt -20 = \$EC te staan, enzovoorts, zodat:

```
($300A) = $23 = 35
($300B) = $EC = -20
($300C) = $A6
($300D) = $0B
($300E) = $63 = 'c = ASCII-c
($300F) = $00 = 0
($3010) = $05 = %101
($3011) = $33
```

In het programma kunnen bijvoorbeeld de volgende instructies (met bijbehorende betekenis) voorkomen:

```
LDX  #BYTES      laad X met het getal BYTES, dus
                  (X) := BYTES = $300A
LDA  BYTES+6      laad A met de inhoud van de
                  geheugenplaats met als adres
                  BYTES+6, dus (A) := (BYTES+6) =
                  ($300A+6) = ($3010) = $05
LDB  2,X          laad B met de inhoud van de
                  geheugenplaats met als adres
                  2+(X), dus (B) := (2+(X)) =
                  (2+$300A) = ($300C) = $A6
```

[<label>] FDB <expressie>[,<expressie>,...]

"Form Double Bytes" doet hetzelfde als FCB, echter met dit verschil dat nu de waarden van de expressies telkens twee achtereenvolgende geheugenlocaties in beslag nemen. FDB-directives worden veelal gebruikt om tabellen van constanten te maken, waarbij elke constante 2 bytes groot is. Op deze wijze kunnen we bijvoorbeeld tabellen met adressen maken. Adressen zijn immers 2-bytes getallen.

Voorbeeld 5.10

WORDS FDB \$FF04,%101101

Als WORDS tijdens het assembleren bijvoorbeeld de waarde \$3012 krijgt, dan zal na het assembleren gelden:

(\$3012) = \$FF
(\$3013) = \$04
(\$3014) = \$00
(\$3015) = \$2D

[<label>] FCC /<string>/

"Form Constant Characters" heeft een soortgelijk effect als FCB, met dien verstande dat nu achtereenvolgens de ASCII-waarden van de in <string> gespecificeerde tekst in het geheugen worden opgeslagen.

FCC-directives worden toegepast om allerlei boodschappen in het geheugen op te bergen, die vanuit het programma kunnen worden aangeroepen. Denk bijvoorbeeld aan de foutboodschap "Delen door 0", die moet worden geprint, zodra in een deelroutine door 0 zou moeten worden gedeeld. Dit betekent dat de berekening niet kan worden uitgevoerd.

Voorbeeld 5.11

FOUT FCC /Delen door 0/

Het zichtbaar maken van deze foutboodschap kan met het volgende stukje programma geschieden:

*	LDX	#FOUT	pointer X wijst naar het eerste
*			te tonen karakter van de
*			foutboodschap
NEXTCH	LDA	,X+	lees een karakter en laat X
*			naar het volgende karakter
*			wijzen
*	STA	VIDEO	toon het karakter op het video
*			display
*	CMPX	#FOUT+12	ga na of alle karakters zijn
*			getoond
*	BNE	NEXTCH	zo niet, lees het volgende
*			karakter (Branch if Not Equal
*			to NEXTCharacter)

We hebben hier voor de eenvoud aangenomen dat we een karakter op een videoscherm kunnen zetten door de bijbehorende ASCII-informatie op de locatie met het adres VIDEO op te bergen.

[<label>] RMB <expressie>

"Reserve Memory Bytes" zorgt ervoor, dat de assembler een aantal geheugenlocaties overslaat. Dit heeft tot gevolg, dat er als het ware een aantal geheugenplaatsen (bytes) worden gereserveerd voor bijvoorbeeld het opslaan van tussenuitkomsten en/of resultaten van bewerkingen. Het aantal bytes waar het om gaat, wordt bepaald door de waarde van de expressie.

Voorbeeld 5.12

RESULT RMB 15

Hier wordt een geheugenruimte van 15 bytes gereserveerd ten behoeve van het opbergen van het resultaat van een bewerking. De eerste byte heeft als (symbolisch) adres RESULT. Is de assembler bij de RMB-directive aangekomen op adres \$3022, dan wordt RESULT overal in het programma vertaald door \$3022. Na de RMB 15 zal de assembler de volgende gegevens of instructies laden vanaf adres \$3031.

[<label>] END [<expressie>]

"END" deelt de assembler mee, dat hij klaar is met vertalen. Na de END-directive volgen dus geen instructies of gegevens meer, die in het geheugen van de computer moeten worden opgenomen. END moet derhalve één keer in een programma voorkomen, en wel aan het einde.

END is geen instructie en heeft derhalve niet tot gevolg dat het programma stopt! Dit laatste moet op een andere wijze worden bewerkstelligd, maar hierover meer bij het behandelen van de instructies (in hoofdstuk 6).

Met behulp van <expressie> kunnen we aangeven op welk adres het programma moet worden gestart. Het operating system, dat zorg draagt voor het starten van het programma, moet weten op welke geheugenplaats het programma begint. Vandaar dat het handig is, dit adres op te geven in de END-directive. Doen we dit niet, dan zullen we dit adres bij het starten van het programma apart moeten opgeven, hetgeen betekent dat we moeten onthouden op welke adressen al onze programma's beginnen. Vermelden we bij elk programma in de END-directive waar het betreffende programma begint, dan kan elk programma na het laden door het operating system op het juiste beginadres worden gestart, zonder dat dit beginadres moet worden opgegeven bij het execute-commando.

Voorbeeld 5.13

END \$3031

END START

In het laatste voorbeeld moet START als label voorkomen bij de eerste instructie van het programma.

Tot slot behandelen we nog enkele assembler directives, die handig zijn voor het verkrijgen van een goede lay-out van de listing van het programma.

NAM <string>

"NAME" zorgt ervoor dat bij het afdrucken van het programma elke pagina begint met een hoofd, te beginnen bij die pagina die volgt op de directive NAM. In het hoofd wordt de tekst afgedrukt, die u vermeldt in het operandveld. <string> is een opeenvolging van karakters, die tijdens het afdrucken als hoofd op elke pagina wordt geprint.

Deze <string> hoeft niet gelijk te zijn aan de <label>, die het

begin van het programma aangeeft.
De naam waaronder het programma op een disk is weggeschreven,
kan anders zijn dan de hier bedoelde <string> en <label>.

Voorbeeld 5.14

NAM CV-REGELING

Op elke pagina van de listing verschijnt nu het hoofd "CV-REGELING", terwijl tevens automatisch de datum en een paginanummer worden afgedrukt.

PAGE

"PAGE" zorgt ervoor dat bij het afdrukken naar een nieuwe pagina wordt gesprongen op de plaats waar de assembler directive PAGE staat.

SPC <expressie>

"SPaCe" zorgt ervoor, dat bij het afdrukken een aantal regels (gelijk aan de waarde van <expressie>) wordt overgeslagen op de plaats van de assembler directive SPC.

Voorbeeld 5.15

SPC 3

Er worden 3 regels overgeslagen op de plaats waar deze directive in de tekst voorkomt.

Assembler directives mogen in principe overal in het programma voorkomen. Het zal echter duidelijk zijn dat de assembler directives die geheugenruimte invullen of reserveren, niet zomaar tussen instructies mogen voorkomen. Hierdoor zouden de gegevens die in deze geheugenruimte staan, als instructies worden opgevat tijdens het uitvoeren van het programma. Dit is het geval bij de directives ORG, FCB, FDB, FCC en RMB. De vier laatste assembler directives kunt u het beste, samen met de EQU-directives, aan het begin of aan het einde van het programma zetten. Dit vergroot de overzichtelijkheid.

We zullen hier nog een voorbeeld geven, dat aantoont hoe we handig gebruik kunnen maken van de faciliteiten die een assembler ons biedt.

Voorbeeld 5.16 (Symbolische adressen vergemakkelijken wijzigingen)

Stel we hebben een programma, waarin op diverse plaatsen het BTW-tarief (19 %) voorkomt, bijvoorbeeld in instructies als

LDA #19

Wanneer we dit programma ook nog gebruiken, nadat het BTW-tarief is gewijzigd (in bijvoorbeeld 20 %), dan moeten we instructie voor instructie doorzoeken op het voorkomen ervan. In elke instructie moet het percentage van 19 % worden gewijzigd in 20 %. Dit is vooral bij grotere programma's een enorme klus. Bovendien is het risico een instructie over het hoofd te zien, niet gering. Vandaar dat het verstandig is het

BTW-tarief aan het begin van het programma door middel van een assembler directive vast te leggen:

```
BTW      EQU      19
```

In het programma passen we dan immediate addressing instructies toe, zoals

```
LDA      #BTW
```

Verandert nu het BTW-tarief in 20%, dan veranderen we slechts de EQU-directive in:

```
BTW      EQU      20
```

en assembleren het programma opnieuw. De assembler zorgt er nu voor dat overal in het programma het symbool BTW wordt vertaald door 20.

Een andere oplossing is het BTW-tarief als constante in het geheugen op te laten nemen met behulp van de assembler directive

```
BTW      FCB      19
```

en extended addressing instructies te gebruiken, zoals

```
LDA      BTW
```

Men hoeft nu zelfs niet opnieuw te assembleren als men zelf de inhoud van de betreffende geheugenplaats wijzigt in 20. Gaat men later het programma opnieuw in het geheugen laden, dan moet het eerst opnieuw zijn vertaald met de gewijzigde FCB-directive. Tijdens het laden van het programma worden ook de gedefinieerde constanten geladen.

Het is raadzaam om zo weinig mogelijk constanten in de instructies op te nemen. Beter is het de constanten als groep te declareren met behulp van EQU-, FCB- of FDB-directives. Zodoende hoeven we bij wijzigingen alleen deze declaraties (aan het begin of einde) van het programma aan te passen en het programma opnieuw te assembleren.

Pas ook bij dit soort wijzigingen het revisienummer en de datum van het programma aan en omschrijf kort wat de revisie inhoudt. In ons voorbeeld kunnen we een commentaarregel opnemen als:

```
* Revisie 10: BTW gewijzigd van 19 % in 20 %.
```

Het programma wordt door het toepassen van symbolen veel beter leesbaar. BTW is immers beter te begrijpen en eenduidiger dan het getal 19! Het getal 19 kan bijvoorbeeld ook betrekking hebben op het aantal verkochte computers.

5.5 Soorten assemblers

Naar de werking van assemblers kunnen assemblers worden onderverdeeld in one-pass assemblers en two-pass assemblers.

Bij one-pass assemblers leest de assembler een instructie, vertaalt deze en leest de volgende instructie, enzovoorts. Er ontstaat een probleem, zodra een symbool naar voren wijst

(forward reference), dat wil zeggen dat het symbool verderop in het programma als label voorkomt. Dit betekent dat het symbool nog niet kan worden vertaald in een geheugenadres. Denk bijvoorbeeld aan een sprong naar een instructie die verderop in het programma staat. In zo'n geval zal de instructie niet worden vertaald, maar wordt deze in een tabel gezet. Aan het einde van het assembleren zijn alle symbolen bekend. Bij het laden van het programma (door de loader) worden de instructies uit de tabel op de juiste plaats aan het programma toegevoegd. Instructies die terugwijzen (backward reference) kunnen wel tijdens het assembleren worden vertaald, omdat de label, behorende bij het symbool in de operand, reeds gepasseerd is, zodat het geheugenadres van het betreffende symbool reeds bekend is.

De meeste assemblers zijn two-pass assemblers. Zo'n assembler leest eerst alle instructies zonder ze te vertalen, maar genereert tijdens deze eerste stap een zogenaamde symbol table. Hierin staan alle symbolen, die in het programma voorkomen en successievelijk worden de bijbehorende geheugenadressen ingevuld. Aan het einde van de eerste stap is de hele symbol table compleet. Dan leest de assembler voor de tweede keer alle instructies, maar nu wordt elke gelezen instructie telkens vertaald in machinecode. Hierbij wordt gebruik gemaakt van de symbol table, die tijdens de eerste stap is gemaakt. Het voordeel van een two-pass assembler is dat er tijdens de tweede stap een nette listing van het programma kan worden gemaakt, waarbij alle instructies in de juiste volgorde achter elkaar staan. Dit in tegenstelling tot bij de one-pass assembler.

Tot slot geven we de definities van enkele andere soorten assemblers, zonder daar diep op in te gaan.

Een conditional assembler is een assembler die het mogelijk maakt om delen van sourceprogramma's alleen te vertalen als er tijdens het assembleren aan bepaalde voorwaarden is voldaan. We kunnen op deze wijze extra variabelen al dan niet insluiten of instructies toevoegen die alleen ten behoeve van diagnostiek aanwezig zijn tijdens de testfase. Het voordeel van een conditional assembler is dat het sourceprogramma niet hoeft te worden gewijzigd, als een bepaalde groep instructies niet nodig is. We hoeven dan alleen de conditie te veranderen en het programma opnieuw te vertalen. Het nadeel is echter, dat de sourceprogramma's minder goed leesbaar zijn. Ze bevatten immers alle eventueel voorkomende groepen instructies.

Een macro-assembler laat toe om gebruik te maken van zogenaamde macro's. Een macro is een rijtje instructies. Een macro heeft verder een naam. Het gebruik van een macro is nuttig als een bepaald groepje instructies vaker voorkomt in een programma. We hoeven dan niet telkens al die instructies te herhalen, maar

kunnen volstaan met het opgeven van de naam van de macro in het opcodeveld. Tijdens het assembleren worden de bijbehorende machinetaalinstructies op de juiste plaatsen tussengevoegd. Door het gebruik van macro's wordt het sourceprogramma korter, beter leesbaar en gemakkelijker te wijzigen. Het objectprogramma wordt niet korter bij gebruik van macro's.

Een cross-assembler is een assembler die sourceprogramma's kan vertalen, die op een ander type microprocessor lopen dan de assembler zelf. Om op een computer met een 6502 microprocessor een sourceprogramma voor een 6809 microprocessor te assembleren, is een cross-assembler nodig.

Een assembler die assembly-programma's van een bepaald type processor vertaalt in machinetaalprogramma's voor een ander type processor, wordt ook cross-assembler genoemd. Het gaat hierbij doorgaans om aanverwante processoren, zoals bijvoorbeeld de MC6809 en de MC6802.

Een self-assembler of resident assembler is een assembler die op dezelfde microprocessor loopt als de sourceprogramma's, die hij moet assembleren.

Een meta-assembler kan verschillende instructiesets behandelen.

Een disassembler doet het omgekeerde als een assembler. Hij vertaalt een machinetaalprogramma in een assemblyprogramma. Het nadeel van een gedisassembleerd programma is dat dit geen commentaar en geen labels (symbolische adressen) bevat. Derhalve is zo'n assemblyprogramma vrijwel onleesbaar.

5.6 Samenvatting

Een assembler

- vertaalt een assemblyprogramma (een programma in mnemonics) in machinetaal,
- zoekt hiertoe de juiste opcodes in machinetaal op,
- berekent de bijbehorende postbytes,
- laat het gebruik van symbolische adressen toe,
- berekent de displacements bij relative branch instructies en bij program counter relative instructies,
- laat het gebruik van commentaar toe,
- geeft een aantal foutmeldingen, onder andere bij niet gedefinieerde symbolen, bij dubbel gedefinieerde labels en bij foutief ingetoetste instructies en
- kent een aantal directives, waarmee op eenvoudige wijze informatie in het geheugen kan worden opgeslagen en waarmee een overzichtelijke lay-out van een programmalisting kan worden verkregen.

Een assembly-instructie bestaat uit vier velden, die echter niet allen informatie hoeven te bevatten:

label opcode operand commentaar

De operand heeft de volgende vorm, afhankelijk van de addressing mode:

```

immediate      #<expressie>
extended       ><expressie> of <expressie>
direct         <<expressie>
indexed        <expressie>,R
               met R = X, Y, U, S, PC of PCR
extended indirect  [<expressie>]
indexed indirect  [<expressie>],R]
relative        <expressie>

```

<expressie> is een symbool, een constante of een opeenvolging van symbolen en/of constanten, gescheiden door rekenkundige of logische operatoren.

Assembler directives:

ORG	<expressie>	origin
<label> EQU	<expressie>	equate
[<label>] FCB	<expressie>[,<expressie>,...]	form constant bytes
[<label>] FDB	<expressie>[,<expressie>,...]	form double bytes
[<label>] FCC	/<string>/	form constant characters
[<label>] RMB	<expressie>	reserve memory bytes
[<label>] END	[<expressie>]	end of assembly file
NAM	<string>	name
PAGE		new pagina
SPC	<expressie>	space (lege regels)

5.7 Opgaven

- Geef de assemblynotatie van de onderstaande machinetaal-instructies. Gebruik hierbij appendix B.

a. \$4F	e. \$E6 9F 57 BC
b. \$B6 00 A8	f. \$A7 89 37 21
c. \$86 33	g. \$10 AF FD E3 09
d. \$D6 E4	h. \$AF A3
- Geef (in hexadecimale code) adres en inhoud van de geheugenlocaties ten gevolge van de onderstaande (achtereenvolgende) assembler directives.

```

                ORG      $2000
                FDB      $A56E,$33,$8DE5,$3480
MELD           FCC      /error/
                RMB      10
                FCB      25,-20,$11,'Z,%0110
                ORG      $3500
                FDB      12,%1111000100,$ED00
INPUT          EQU      $FF03
                FCB      $40,$10

```


6 INSTRUCTIESET

6.1 Specifieke doelstelling

Na hoofdstuk 6 te hebben bestudeerd, wordt u in staat geacht het volgende te kunnen:

- * De betekenis van de behandelde instructies van de MC6809 kennen.
- * De instructies op de juiste wijze toepassen.
- * Vermijden dat uw programma's onnodig lang worden of moeilijk te begrijpen zijn ten gevolge van een niet-optimale keuze van de instructies.
- * Weten wat er bij elke instructie met de bits van het condition code register gebeurt.
- * De inhoud van het condition code register bepalen na een gegeven instructie.
- * Nagaan (aan de hand van appendix B) hoeveel geheugenruimte en hoeveel tijd elke instructie kost. Dit is nuttig als u zeer tijd- of ruimtekritische programma's moet ontwerpen.
- * Een assembly-instructie vertalen in machinecode.
- * Een machinetaalinstructie disassembleren.
- * Structogrammen opstellen.
- * Vanuit structogrammen assemblyprogramma's ontwerpen.
- * Gegeven assemblyprogramma's begrijpen.

6.2 Inleiding

U bent al vertrouwd geraakt met de opbouw van een instructie. Zoals u weet kan deze bij de MC6809-microprocessor er als volgt uitzien:

```
<opcode>
<opcode><postbyte>
<opcode><operand>
<opcode><postbyte><operand>
```

De bytes van de instructies van een programma staan als een lange rij binaire getallen in het geheugen, nadat het assemblyprogramma door de assembler is vertaald en het vertaalde programma, het machinetaalprogramma, door de loader in het werkgeheugen van de computer is gezet. De vraag is nu hoe de processor uit die lange, aaneengesloten rij binaire getallen kan onderkennen, welke getallen hij als opcode moet behandelen en welke als postbyte of als operand. Welnu, dit gaat als volgt in zijn werk.

Starten van programma

Als een programma wordt gestart, doordat u het operating system

hiertoe de opdracht geeft, moet de processor allereerst weten op welke geheugenlocatie de eerste instructie staat. Hiervoor zorgt het operating system door de program counter van de microprocessor te laden met het adres van die geheugenlocatie. Dit adres moeten we natuurlijk aan het operating system meedelen, hetgeen op de volgende manier kan geschieden:

1. De binaire programmapfile bevat het startadres van het programma niet.

Laten we aannemen, dat we het (binaire) programma met de naam CV willen uitvoeren ten behoeve van de besturing van een CV-installatie. Het programma staat op een disk. De binaire file CV bevat informatie over de geheugenadressen waarop de binaire informatie moet worden geladen. Een file is een rij gegevens op een disk. De assembler heeft bij het vertalen gezorgd voor de informatie omtrent de te gebruiken geheugenruimte. De assembler heeft deze informatie op zijn beurt weer van de programmeur gekregen via de assembler directive ORG, bijvoorbeeld ORG \$2000.

We moeten het binaire programma CV eerst laden in het werkgeheugen van de computer. Dit wordt verzorgd door het operating system na het commando GET CV (get betekent haal). Als dat gebeurd is, geven we aan het operating system de opdracht het programma uit te voeren en we delen tevens mee op welk adres de (opcode van de) eerste instructie staat. Dit geschiedt bijvoorbeeld met de opdracht RUN \$2010, aangenomen dat de eerste instructie op adres \$2010 begint. Veronderstel dat er na de ORG \$2000 nog 16 geheugenplaatsen zijn gevuld met constanten middels een FCB-directive.

2. De binaire programmapfile bevat het startadres van het programma wel.

Hebben we in het sourceprogramma het startadres \$2010 opgegeven in de END-directive (END \$2010 of END BEGIN), dan is de assembler in staat geweest dit beginadres in de binaire file op te nemen, zodat het operating system weet op welk adres het programma gestart moet worden. In dit geval kunnen we volstaan met het commando EXE CV (execute CV). Deze opdracht heeft tot gevolg dat het operating system de binaire file CV in het geheugen laadt en vervolgens het programma op het juiste beginadres start. Het voordeel van deze laatste methode is, dat we niet het beginadres van elk programma hoeven te onthouden. De commando's GET, RUN en EXE gelden voor het operating system LABDOS. Bij een ander operating system kunnen deze commando's enigszins afwijken.

Uitvoeren van programma

Nu de program counter door het operating system is geladen met het adres van de eerste instructie, haalt de processor de inhoud van die geheugenplaats op en plaatst deze in het instructieregister. Elke instructie begint immers met een opcode. Een programma start zodoende altijd op de

geheugenlocatie waar de opcode van de eerste instructie is opgeborgen.

De opcode geeft vervolgens aan wat de aard van de instructie is. Hieruit blijkt tevens of voor het uitvoeren van deze instructie nog meer gegevens nodig zijn, bijvoorbeeld in de vorm van een tweede opcode-byte, een postbyte of een operand. Na het decoderen van de opcode weet de microprocessor of hij nog meer bytes uit het geheugen moet ophalen om de instructie te kunnen uitvoeren. In het geval dat er een postbyte volgt, blijkt uit de postbyte of er een 0-bytes, een 1-byte of een 2-bytes operand moet worden opgehaald (zie de postbytes van de indexed en van de indirect addressing modes in appendix B). Als op deze wijze de gehele instructie door de microprocessor uit het geheugen is opgehaald, wordt de betreffende instructie uitgevoerd. Daarna haalt de microprocessor de volgende byte uit het programmeergeheugen op en zet deze in het instructieregister. Deze byte wordt derhalve weer als opcode opgevat, hetgeen ook juist is. Immers na een instructie volgt weer een volgende instructie, die begint met een opcode.

Op deze wijze kan de microprocessor onderscheid maken tussen de verschillende instructies en tussen de diverse onderdelen van een instructie.

Onderscheiden van adresseringsmethode

De opcode, eventueel aangevuld met informatie uit de postbyte, maakt tevens onderscheid tussen de diverse adresseringsmethoden en geeft aan uit hoeveel bytes (1 of 2) de operand bestaat.

Voorbeeld 6.1

Voor de instructie SUBA zijn er verschillende opcodes, afhankelijk van de addressing mode (zie appendix B):

\$80 = SUBA immediate

Er volgt hier een 1-byte operand, zijnde de data, omdat A een 1-byte register is.

\$90 = SUBA direct

Hier volgt er een 1-byte operand, namelijk de low-byte van het effectieve adres.

\$A0 = SUBA indexed of indirect

Er volgt hier een postbyte, die aangeeft om welk type indexed of indirect addressing het gaat en hoeveel bytes er nog volgen ten behoeve van de operand.

\$B0 = SUBA extended absolute

Er volgen nu nog twee bytes voor de operand, namelijk de twee bytes van het effectieve adres.

Let op dat extended indirect addressing instructies dezelfde opcode hebben als de overeenkomstige indexed addressing instructies. Is de postbyte gelijk aan \$9F, dan hebben we te maken met extended indirect addressing, anders met een van de vormen van indexed addressing of indexed indirect addressing.

Grootte van instructies

Uit de gegevens van voorbeeld 6.1 kunnen we gemakkelijk afleiden hoeveel geheugenruimte elke instructie nodig heeft (zie ook appendix B: in de kolommen, aangegeven met het symbool "*" staan het aantal bytes dat elke instructie kost).

Voorbeeld 6.2

SUBA immediate kost 2 bytes:

1 byte voor de opcode en 1 byte voor de data

SUBA direct kost 2 bytes:

1 byte voor de opcode en 1 byte voor het low-address

SUBA indexed kost $2+n$ bytes:

1 byte voor de opcode, 1 voor de postbyte en n voor de operand, waarbij n afhangt van het type adressering dat gebruikt wordt. Dit laatste wordt bepaald door de postbyte. In de kolommen, aangegeven met "+*", vinden we de waarde van n (zie appendix B). Voor de 16-bits constant offset indexed addressing is n vanzelfsprekend gelijk aan 2 (2-bytes offset), zodat deze instructie in totaal $2+2=4$ bytes bevat.

SUBA extended absolute kost 3 bytes:

1 byte voor de opcode en 2 bytes voor het adres

Duur van instructies

De uitvoering van een instructie duurt langer naarmate de instructie uit meer bytes is opgebouwd. Het ophalen van een byte uit het geheugen kost één klokcyclus. Dit is de periodetijd van de klokpuls, waarmee de timing binnen de computer wordt verzorgd. Een instructie van 4 bytes kost 4 klokcycli om de instructie binnen te halen in de microprocessor. Daarna moet de instructie nog worden uitgevoerd. Of dit nog kan geschieden tijdens de klokcyclus waarin de laatste byte van de instructie wordt binnengehaald, hangt af van de bewerking die moet worden uitgevoerd. Zo nodig komen er nog een of meerdere klokcycli bij ten behoeve van de uitvoering van de instructie. In appendix B kunnen we in de kolommen met het symbool "~" de tijdsduur van elke instructie aflezen. Deze tijdsduur is uitgedrukt in aantallen klokcycli.

Voorbeeld 6.3

SUBA immediate duurt 2 klokcycli.

SUBA direct duurt 4 klokcycli.

Hier moet immers het effectieve adres van de data worden bepaald en vervolgens moet de data uit het datageheugen worden opgehaald, alvorens de bewerking kan worden uitgevoerd.

SUBA indexed duurt $4+m$ klokcycli, waarbij m afhangt van het type indexed addressing. We vinden de waarde van m in de kolom "+~" van appendix B. Zo kost de 16-bits constant offset indexed addressing instructie $4+4=8$ klokcycli.

Soorten instructies

De MC6809-microprocessor kent in totaal 1464 instructies. Hierin zijn inbegrepen de diverse adresseringsmethoden van de instructies. Afgezien van de adresseringsmethoden heeft de MC6809 120 instructies, waarbij we de 19 long branches niet apart meerekenen. Het aantal verschillende basisinstructies bedraagt 59.

Niet alle instructies zijn op elk register van toepassing. Evenmin zijn alle adresseringsmethoden voor elk type instructie geldig. Dit maakt het programmeren in assembly enigszins ingewikkeld. Gebruikt men een niet-bestaande instructie, dan zal de assembler een foutmelding geven. Met behulp van een overzicht van de bestaande instructies (zie appendix B) kan de programmeur zulke fouten vrij snel herstellen.

Vervelender is het feit dat niet alle instructies de bits van het condition code register op dezelfde manier beïnvloeden. Om programmeerfouten te voorkomen, moet de programmeur nauwgezet nagaan hoe de instructies deze bits beïnvloeden. Deze beïnvloeding is te vinden in appendix B. De assembler zal programmeerfouten niet signaleren.

Een assembler kan alleen syntaxfouten ontdekken. Syntaxfouten zijn taalfouten, zoals niet-bestaande opcodes en niet-gedefinieerde operands. Programmeerfouten (het verkeerd gebruik van instructies) kan de assembler niet waarnemen. De assembler weet immers niet wat de bedoeling van de programmeur is.

Met een zogenaamde orthogonale instructieset, waarbij elke instructie op dezelfde wijze van toepassing is op alle registers en op alle adresseringsmethoden, zou het programmeren eenvoudiger zijn. Er zouden echter veel meer bits nodig zijn voor het coderen van al die mogelijke combinaties van instructies, registers en adresseringsmethoden. Hierdoor zou de opcode uit meerdere bytes moeten bestaan, zodat de instructies meer geheugenruimte zouden innemen en een grotere executietijd zouden hebben. De programma's zouden dus minder efficiënt zijn.

De instructies kunnen naar hun functie worden verdeeld in vier groepen:

1. datatransportinstructies
2. logische instructies
3. rekenkundige instructies
4. spronginstructies

In de volgende paragrafen worden deze instructies uitgebreid behandeld. Hierbij zal telkens het betreffende deel van het overzicht van de instructies van appendix B worden besproken, opdat u weet, hoe u deze informatie op de juiste manier kunt gebruiken.

Indeling en gebruikte symbolen van appendix B:

- Kolom "instr" vermeldt de mnemonics van de instructies.
- Kolom "op" geeft de hexadecimale notatie van de opcodes.
- Onder "" vindt u de duur van de instructies, uitgedrukt in het aantal klokcycli.
- Bij "~+" moeten nog een aantal klokcycli worden bijgeteld om de executietijd van de instructies te vinden. Dit bij te tellen aantal is vermeld in kolom "+~" van de postbyte-tabel van de indexed en indirect addressing instructies.
- In kolom "*" is aangegeven uit hoeveel bytes de instructies bestaan.
- De indexed en indirect addressing instructies bevatten het aantal bytes dat vermeld is in kolom "*+", vermeerderd met het aantal van de postbyte-kolom "+*".
- Onder "beschrijving" staat een korte uitleg van de instructies.
- In de kolommen "H", "N", "Z", "V" en "C" wordt vermeld wat er met de betreffende bits van het condition code register gebeurt. Hierbij zijn de volgende symbolen gebruikt:
 - "-": bit verandert niet
 - "*": bit wordt aangepast aan het resultaat van de instructie
 - "0": bit wordt altijd gereset
 - "1": bit wordt altijd geset
 - "?": de waarde van de bit is niet bekend
- Een kleine letter verwijst naar een opmerking.
- Met (M) (M+1) wordt een 2-bytes getal aangeduid, waarbij de high-byte op adres M staat en de low-byte op adres M+1.

6.3 Datatransportinstructies

6.3.1 Algemeen

Datatransportinstructies (of data transfer-instructies) zijn de meest elementaire instructies. Denk hierbij aan het binnenhalen van de waarden van de kamertemperatuur en het wegsturen van de signalen om de hoofdbrander te ontsteken en de waterpomp te laten lopen bij het programma, dat de CV-installatie aanstuurt. Bij berekeningen moeten vaak tussenresultaten worden opgeborgen in het geheugen van de computer om ze later weer op te halen voor verdere verwerking.

We kunnen de datatransportinstructies in vier groepen onderverdelen:

1. transport tussen een register en het geheugen
2. transport tussen twee registers
3. transport tussen een register en een input/output device
4. transport tussen het geheugen en een input/output device

6.3.2 Transport tussen register en geheugen

Datatransport van register naar geheugen kan geschieden met behulp van store-instructies of met behulp van push-instructies.

Met behulp van de store-instructies kunnen we de inhoud van een register opbergen in het datageheugen.

De MC6809 kent de volgende STore-instructies:

```

STA   STS   STX
STB   STU   STY
STD

```

De inhouden van PC, DP en CC kunnen niet rechtstreeks in het datageheugen worden opgeborgen. Op welke wijze u deze registerinhouden toch in het datageheugen kunt opbergen, wordt behandeld in paragraaf 6.3.3.

Fig. 6.1 toont de mogelijke addressing modes met de bijbehorende opcodes van deze instructies, alsmede de beïnvloeding van de flags van het condition code register.

De store-instructies kunnen worden gebruikt met direct, indexed, indirect en extended addressing.

De instructies STS en STY hebben een 2-bytes opcode (respectievelijk \$10DF en \$109F).

Merkt verder op dat de half-carry en de carry niet veranderen bij een store-instructie en dat de overflow-flag altijd wordt gereset. De flags Z en N worden ingesteld overeenkomstig het resultaat van de instructie. Dat wil in dit geval zeggen, overeenkomstig de waarde van het opgeborgen getal. Is dit getal gelijk aan 0, dan wordt Z=1. In het andere geval wordt Z=0. Is het getal negatief, dan wordt N=1, anders wordt N=0. N wordt dus gelijk aan de tekenbit van het opgeborgen getal.

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~ + ##	extend op ~ #	inher op ~ #		
STA		97 4 2	A7 4+ 2+	B7 5 3		(M) := (A)	***0-
STB		D7 4 2	E7 4+ 2+	F7 5 3		(M) := (B)	***0-
STD		DD 5 2	ED 5+ 2+	FD 6 3		(M) (M+1) := (D)	***0-
STS		10 6 3	10 6+ 3+	10 7 4		(M) (M+1) := (S)	***0-
		DF	EF	FF			
STU		DF 5 2	EF 5+ 2+	FF 6 3		(M) (M+1) := (U)	***0-
STX		9F 5 2	AF 5+ 2+	BF 6 3		(M) (M+1) := (X)	***0-
STY		10 6 3	10 6+ 3+	10 7 4		(M) (M+1) := (Y)	***0-
		9	AF	BF			

Fig. 6.1 Overzicht van de STore-instructies

De operand bepaalt op welke geheugenlocatie de informatie moet worden opgeborgen. Bij een 2-bytes register wordt de high-byte opgeborgen op de geheugenlocatie, aangegeven door de operand, terwijl de low-byte wordt opgeborgen op de volgende geheugenplaats (met een hoger adres).

Voorbeeld 6.4

STA	\$40A2	(\$40A2) := (A)
STY	>\$33	(\$0033) := (YH) en (\$0034) := (YL)
STD	SOM	(SOM) := (DH) en (SOM+1) := (DL)
STB	\$48,X	(\$48+(X)) := (B)
STA	[B,U]	(A) wordt opgeborgen op de locatie met als adres ((B)+(U)) ((B)+(U)+1)

Bij de store-instructies gaat de oorspronkelijke informatie van de betreffende geheugenplaats(en) verloren. De inhoud van de registers blijft echter ongewijzigd!

Met behulp van de push-instructies kunnen we de inhoud van de registers op de stack plaatsen. Hierop komen we terug in hoofdstuk 7.

Datatransport van geheugen naar register kan geschieden met behulp van load-instructies of met behulp van pull-instructies. De pull-instructies worden behandeld in hoofdstuk 7. Bij een load-instructie wordt een register gevuld met de inhoud van een geheugenplaats. Net als bij de store-instructies kan voor het aangeven van het adres gebruik worden gemaakt van direct, indexed, indirect en extended addressing. Bovendien komt hier de immediate addressing voor, immers we kunnen een register ook laden met een getal, dat als constante in de operand van de instructie staat, dus in het programmeergeheugen.

De MC6809 kent de volgende Load-instructies (zie ook fig. 6.2):

LDA	LDS	LDX
LDB	LDU	LDY
LDD		

We constateren dat de registers PC, DP en CC niet rechtstreeks vanuit het datageheugen kunnen worden geladen. We komen hierop nog terug in paragraaf 6.3.3.

Bij 2-bytes registers wordt eerst de high-byte gevuld met de inhoud van de locatie, aangewezen door de operand van de instructie, waarna de low-byte wordt gevuld met de inhoud van de volgende geheugenplaats.

Voorbeeld 6.5

LDA	TEMP	(A) := (TEMP)
LDS	\$3A87	(SH) := (\$3A87) en (SL) := (\$3A88)

LDB [\$A400] (B) wordt gelijk aan de inhoud
 van de locatie met als adres
 (\$A400) en (\$A401)

LDX # \$BB51 (X) := \$BB51

LDY # TABEL (Y) := TABEL

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
LDA	86 2 2	96 4 2	A6 4+ 2+	B6 5 3		(A) := (M)	---0-
LDB	C6 2 2	D6 4 2	E6 4+ 2+	F6 5 3		(B) := (M)	---0-
LDD	CC 3 3	DC 5 2	EC 5+ 2+	FC 6 3		(D) := (M) (M+1)	---0-
LDS	10 4 4	10 6 3	10 6+ 3+	10 7 4		(S) := (M) (M+1)	---0-
	CE	DE	EE	FE			
LDU	CE 3 3	DE 5 2	EE 5+ 2+	FE 6 3		(U) := (M) (M+1)	---0-
LDX	8E 3 3	9E 5 2	AE 5+ 2+	BE 6 3		(X) := (M) (M+1)	---0-
LDY	10 4 4	10 6 3	10 6+ 3+	10 7 4		(Y) := (M) (M+1)	---0-
	8E	9E	AE	BE			

Fig. 6.2 Overzicht van de Load-instructies

Bij de load-instructies gaat de inhoud van de geheugenplaatsen niet verloren! Natuurlijk worden wel de betreffende registers overschreven, zodat de oude informatie in die registers verloren is gegaan.

6.3.3 Transport tussen registers

Datatransport tussen registers kan plaatsvinden met behulp van de immediate addressing instructies TransFeR en EXchanGe (zie fig. 6.3).

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
TFR	1F 6 2					(R2) := (R1)	----
EXG	1E 8 2					wissel(R1), (R2)	----

Fig. 6.3 De instructies TransFeR en EXchanGe

De transfer-instructie wordt als volgt geschreven:

TFR R1, R2

waarbij R1 het sourceregister is en R2 het destinationregister. Dit wil zeggen dat de inhoud van register R1 wordt overgebracht in register R2. Hierdoor wordt de oude informatie in R2 natuurlijk overschreven. De informatie in R1 blijft echter

ongewijzigd. We kunnen concluderen dat de inhoud van R1 wordt gekopieerd in R2. Na de transfer-instructie zijn de inhoud van R1 en R2 aan elkaar gelijk.

De volgorde van de namen van de registers in het operandveld van de instructie is hier van groot belang, immers het eerste register is de source (bron) en het tweede is de destination (bestemming)!

Voorbeeld 6.6

TFR	A,B	(B):=(A)
TFR	U,X	(X):=(U)
TFR	X,U	(U):=(X)

De transfer-instructie stelt ons in staat om (via een kleine omweg) de inhoud van PC, DP en CC in het datageheugen te zetten en omgekeerd om deze registers met informatie te vullen.

Voorbeeld 6.7

TFR	CC,A	zet de inhoud van CC in
STA	\$27D4	geheugenlocatie \$27D4
TFR	PC,D	(HULP):=(PCH) en
STD	HULP	(HULP+1):=(PCL)
LDB	\$FA49	vul DP met de inhoud van
TFR	B,DP	geheugenlocatie \$FA49
LDY	#NEXT	(PC):=NEXT
TFR	Y,PC	
LDA	#\$C3	(CC):=\$C3
TFR	A,CC	

De exchange-instructie heeft de volgende structuur (syntax):

EXG R1,R2

Hier worden de inhoud van de registers R1 en R2 met elkaar verwisseld. Dit betekent dat verwisselen van de namen van de registers in het operandveld van de instructie EXG geen effect heeft. Immers EXG R1,R2 is hetzelfde als EXG R2,R1.

Beide registers veranderen hier van inhoud, want de inhoud worden met elkaar verwisseld.

Voorbeeld 6.8

EXG	DP,A	(A):=(DP)vorig en (DP):=(A)vorig
EXG	S,Y	(Y):=(S)vorig en (S):=(Y)vorig

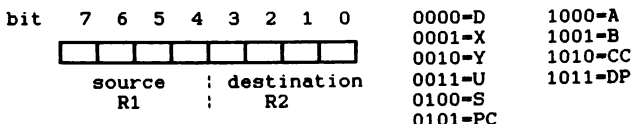


Fig. 6.4 Postbyte van TFR en EXG

De assembler zorgt ervoor dat de namen van de registers in het operandveld van de transfer- en de exchange-instructie worden

vertaald in de juiste postbyte. Fig. 6.4 toont de postbyte in geval van TFR en EXG.

De beide registers, die in het operandveld voorkomen, moeten dezelfde lengte hebben!

De instructies TFR en EXG beïnvloeden geen van de condition code bits. Een uitzondering hierop vormen vanzelfsprekend de instructies

TFR	R1,CC
EXG	R1,CC
EXG	CC,R2

Bij de branch-instructies (zie paragraaf 6.6) wordt uitgebreid ingegaan op het grote belang van de diverse bits van het condition code register.

6.3.4 Transport tussen register en I/O

Bij datatransport tussen een register en een input/output-apparaat kunnen we twee methoden onderscheiden:

1. memory-mapped I/O
2. I/O-mapped I/O

Memory-mapped I/O

De meeste microprocessors (ook de MC6809) maken gebruik van de methode van memory-mapped I/O. Bij deze methode zijn er geen speciale voorzieningen voor de uitwisseling van gegevens met I/O-devices. De I/O-devices worden met behulp van de gewone load- en store-instructies geadresseerd via de adresbus. De microprocessor ziet als het ware geen verschil tussen geheugenschakelingen en I/O-devices. Of er bij een bepaald adres een geheugen dan wel I/O wordt geselecteerd, hangt uitsluitend af van de adresdecoders. Een adresdecoder is de hardware buiten de microprocessor waarmee, afhankelijk van het aangeboden adres, een bepaalde geheugen- of I/O-schakeling wordt geactiveerd. Het zal duidelijk zijn dat er bij elk adres slechts één schakeling mag worden geactiveerd. Anders zouden er meerdere apparaten tegelijkertijd verschillende data op de databus kunnen zetten, hetgeen tot ongedefinieerde data en kortsluitingen leidt.

Voorbeeld 6.9

Veronderstel dat een CV-installatie bereikbaar is via de adressen \$FF22 voor het lezen van de kamertemperatuur en \$FF23 voor het aansturen van de gasklep van de hoofdbrander en de waterpomp. We kunnen de waarde van de kamertemperatuur lezen met behulp van de instructie

LDA \$FF22 (A):=kamertemperatuur

terwijl we de hoofdbrander kunnen ontsteken en de waterpomp laten lopen (aannemende dat deze aangesloten zijn op de bits 0

en 1) met de instructies

LDB	#\$00000011	ontsteek de hoofdbrander en
STB	\$\$F23	zet de waterpomp aan

I/O-mapped I/O

Bij I/O-mapped I/O is er tenminste één extra controllijn, die aangeeft of er sprake is van geheugenadressering of van I/O-adressering. De adressering zelf geschiedt in beide gevallen meestal via dezelfde adresbus. Bij het adresseren van de I/O zijn niet altijd alle adreslijnen gedefinieerd.

Bij I/O-mapped I/O zijn er aparte instructies voor het communiceren met I/O-devices. Zo heeft de Z80-microprocessor voor de communicatie met I/O-apparaten de aparte instructies IN en OUT, waarbij het adres van het I/O-apparaat over de low-byte van de adresbus wordt verzonden.

6.3.5 Transport tussen geheugen en I/O

Bij datatransport tussen het geheugen en een input/output-apparaat zijn in feite geen instructies nodig. Het datatransport gaat in dit geval immers helemaal buiten de microprocessor om. Vandaar dan men hier spreekt van direct memory access (DMA). De randapparaten hebben bij deze methode van datatransport rechtstreeks toegang tot het geheugen. Dat dit niet vanzelf gaat, spreekt voor zich. DMA wordt behandeld in hoofdstuk 10.

6.3.6 Voorbeelden van datatransport

Gestructureerd programmeren

In het algemeen zijn er meer oplossingen mogelijk voor een bepaald probleem. Welke oplossing de beste is, hangt af van de eisen die men stelt aan het programma. Is het belangrijk dat het objectprogramma zeer weinig geheugenruimte in beslag neemt, moet de executietijd van het programma minimaal zijn of moet het programma juist modulair van opbouw zijn, gemakkelijk te begrijpen en eenvoudig te wijzigen? Meestal zijn geheugenruimte en executietijd niet van belang. In dit boek ligt de nadruk op het ontwerpen van overzichtelijke en goed-gedocumenteerde programma's.

Om tot overzichtelijke programma's te komen, ontwerpen we eerst een structogram. Als het structogram eenmaal gereed is, zal blijken dat het vertalen hiervan in een programma eenvoudig is. Het opzetten van een structogram kost de meeste tijd, omdat hierbij het probleem moet worden geanalyseerd en uiteengerafeld in kleinere problemen, die gemakkelijk op te lossen zijn. Het overslaan of achteraf maken van een structogram levert geen

tijdwinst op, maar tijdverlies. Dit wordt veroorzaakt, doordat de ontwerper nu twee dingen tegelijk moet doen.

1. Hij moet het probleem structureren, dat wil zeggen hij moet nagaan welke acties er achtereenvolgens gedaan moeten worden om tot de oplossing van het gestelde probleem te komen. Hierbij zullen een aantal beslissingen moeten worden genomen en er zullen herhalingen voorkomen.
2. De ontwerper moet bepalen welke instructies gebruikt moeten worden.

Gebeuren deze zaken tegelijk, dan is de ontwerper al heel gauw het spoor in zijn programma bijster en zal hij vrij veel fouten maken. Bovendien is achteraf erg moeilijk na te gaan of het programma correct is. Het debuggen (testen en corrigeren) en het documenteren van het programma zal dan erg tijdrovend zijn.

Het analyseren van het probleem zal vlugger gaan als men zich niet hoeft te bekommeren om de keuze van de instructies (in welke programmeertaal dan ook). De ontwerper kan zich volledig concentreren op de te nemen acties, beslissingen en herhalingen om deze op overzichtelijke wijze in kaart te brengen met behulp van een structogram. Het programma kan daarna snel, met weinig inspanning en met weinig fouten worden gemaakt door het verkregen structogram nauwgezet te volgen en hierbij elk blokje te vertalen in een of meerdere instructies. De programmeur hoeft alleen te doen wat er staat, zonder nog te hoeven nadenken over de oplossing van het probleem.

Natuurlijk zal een structogram voor zeer kleine programma's niet nodig zijn. Echter bij programma's die meer dan ongeveer 10 instructies bevatten, blijkt het werken met een structogram al tijdwinst op te leveren. Het maken van de documentatie gaat aan de hand van een structogram ook veel beter. Een structogram biedt zelf al een behoorlijke documentatie.

Bij het ontwerpen van programma's met behulp van structogrammen zullen we steeds de volgende strategie aanhouden voor wat betreft de keuzestructuur en de herhalingsstructuur.

Keuzestructuur

Bij de keuzestructuur (if-then-else) volgen de instructies, behorende bij de linker actie (in fig. 6.5 actie T) direct na de spronginstructie. De instructies die horen bij de rechter actie (actie F), worden bereikt door de sprong uit te voeren. In fig. 6.5 wordt er niet gesprongen als de conditie true is. Actie T wordt nu uitgevoerd. Hierna wordt de vervolgactie uitgevoerd.

Is de conditie false, dan wordt er wel gesprongen en wordt actie F uitgevoerd. Nadat actie F is uitgevoerd, moet er over actie T worden heengesprongen naar de vervolgactie, die in beide gevallen (true en false) moet worden uitgevoerd, dus onafhankelijk van de voorgaande beslissing.

In fig. 6.5 ziet u het structogram van een keuze met de

bijbehorende structuur van het programma.

De keuzestructuur kan worden gelezen als "if conditie is true, then actie T, else actie F".

De fantasie-instructie "TC" (Test Condition) wordt gebruikt om de conditie te testen, die bepaalt of actie T dan wel actie F moet worden uitgevoerd.

De fantasie-instructie "BF" (Branch if False) is nodig om aan te geven, dat de betreffende sprong alleen wordt uitgevoerd, indien de beslissing genomen wordt om actie F uit te voeren. Wordt er niet gesprongen (conditie is true), dan wordt het programma vervolgd met de instructies die bij actie T horen. In echte programma's worden in plaats van de labels IF, THEN en ELSE suggestievere namen gekozen, zoals PMPAAN en GASUIT voor de acties die de pomp moeten starten, respectievelijk de gasklep sluiten.

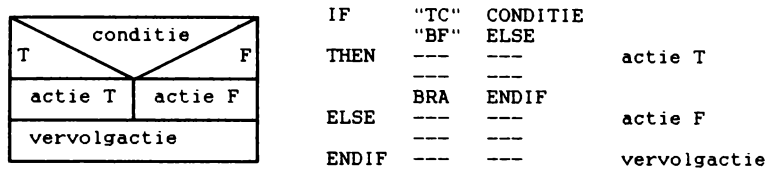


Fig. 6.5 Programma bij een keuzestructuur

Herhalingsstructuur

Fig. 6.6 laat zien hoe de aanpak bij een do-while-structuur is. Het programma kan worden gelezen als "while conditie is true, do actie T". Is de conditie niet meer true, dan wordt de vervolgactie uitgevoerd.

Fig. 6.7 toont het overeenkomstige programma bij een repeat-until-structuur. Dit kan worden gelezen als "repeat actie F until conditie is true". Zodra de conditie true is, wordt het programma voortgezet met de vervolgactie.

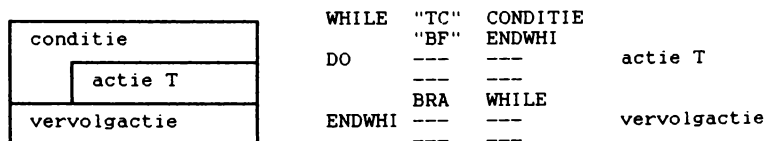


Fig. 6.6 Programma bij een do-while-structuur

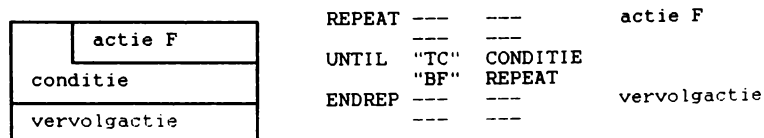


Fig. 6.7 Programma bij een repeat-until-structuur

Om optimaal te programmeren, moet u alle instructies van de microprocessor kennen. Dit betekent dat de voorbeelden aanvankelijk vrij eenvoudig zullen zijn. Naarmate er echter meer instructies behandeld zijn, zullen de gegeven problemen ingewikkelder worden. Dan komen de diverse instructies ook beter tot hun recht.

De instructies en de assembler directives zullen worden voorzien van een regelnummer, uitsluitend met het doel gemakkelijk naar de diverse (pseudo)instructies te kunnen verwijzen voor het maken van kanttekeningen. Bij het invoeren van programma's in de computer, mogen deze regelnummers niet worden ingevoerd!

De programma's laten we eindigen met SWI. Dit is een software interrupt, waarop we hier niet verder zullen ingaan. We nemen aan dat er naar een routine wordt gesprongen, die de besturing teruggeeft aan het operating system. Dit betekent dat het lopende programma wordt gestopt en dat we het operating system een nieuwe opdracht kunnen geven, bijvoorbeeld om een ander programma op te starten.

Hier volgen een aantal voorbeelden van toepassingen van datatransportinstructies.

Voorbeeld 6.10 (Initialiseer het direct page register)

Veronderstel dat er een tabel van constanten is, die op een willekeurige pagina in het geheugen kan staan. Het nummer van de pagina, waarop de tabel wordt geplaatst (bijvoorbeeld door het operating system) staat op adres TABADR (= \$7FFF). Schrijf een programma dat dit paginanummer in het direct page register zet.

Fig. 6.8 toont het structogram van het te ontwerpen programma.

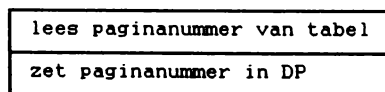


Fig. 6.8 Structogram bij het initialiseren van DP

```

10          NAM      INITIALISEER DP
20 *Progr: J.G. Rouland
30 *Revisie 0   Datum: 02-03-1986
40 *
50 *Dit programma laadt DP met de inhoud van $7FFF.
60 *
70 *
80          ORG      $3000          beginadres van objectprogramma
90 TABADR EQU      $7FFF
100 *
110 INIDP   LDA      TABADR          zet het paginanummer
120          TFR      A,DP           in DP

```

130	SWI	stop
140	END	INIDP

De lege commentaarregels (40, 60, 70 en 100) verhogen de leesbaarheid. Het is zinvol om het gebruik van numerieke adressen te vermijden. Indien er achteraf andere geheugenadressen worden gekozen, moet het hele programma nauwgezet worden doorlopen om al deze adressen aan te passen. Dit leidt gemakkelijk tot fouten. Gebruikt u alleen symbolische adressen, dan kunt u bij wijzigingen volstaan met het aanpassen van de EQU-directives, die bij elkaar staan aan het begin of aan het einde van de source file.

Iets soortgelijks geldt ook voor constanten. Ook deze kunnen worden vervangen door symbolen, waaraan de juiste waarden kunnen worden toegekend door middel van EQU-, FCB- of FDB-directives.

Voorbeeld 6.11 (Initialiseer geheugenbuffer)

EPROM's (Erasable Programmable Read Only Memories) zijn geheugenelementen, die met behulp van een EPROM-programmer kunnen worden geprogrammeerd. Ze zijn geheel met enen gevuld, als ze door de fabrikant worden afgeleverd. Programmeren betekent het nul maken, dan wel het ongewijzigd laten van locaties binnen de EPROM. Een 0 kan niet meer veranderd worden in een 1, hetzij door de gehele EPROM te wissen (met behulp van ultraviolet licht). Dit wissen duurt ongeveer een half uur. Het is verstandig om alleen dat deel van de EPROM te programmeren, dat voor het programma nodig is, en de rest met enen gevuld te laten. Dan kunt u eventueel later altijd nog een volgend programma in dezelfde EPROM bijladen, zonder eerst de EPROM te moeten wissen om daarna beide programma's achter elkaar te laden.

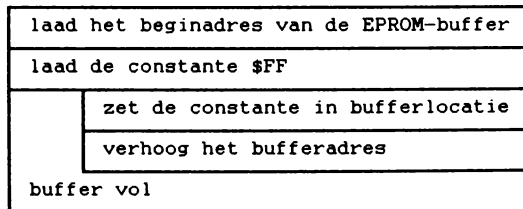


Fig. 6.9 Structogram bij het vullen van een EPROM-buffer met enen

Bij het programmeren van een EPROM wordt de informatie in een bepaald deel van het geheugen, de zogenaamde EPROM-buffer, in de EPROM gekopieerd. Stel dat de EPROM-buffer zich bevindt vanaf adres \$1000 tot en met adres \$17FF. Het gaat hierbij kennelijk om een 2k8-EPROM, immers de geheugenbuffer bevat 2048 bytes. 2k8 duidt op 2000 (in feite 2048) geheugenlocaties met een inhoud van 8 bits. Deze buffer moet nu worden gevuld met

enen. Dit betekent dat op elke locatie \$FF moet komen te staan. Dit impliceert een herhalingsstructuur, die 2048 keer moet worden doorlopen. Het structogram, behorende bij het geschetste probleem, wordt getoond in fig. 6.9. Het programma volgt hierna.

```

10          NAM      INIFF
20 *Progr.:   J.G. Rouland
30 *Revisie 0   Datum: 02-03-1986
40 *
50 *INIFF laadt de EPROM-buffer ($1000 t/m $17FF) met enen.
60 *
70 *
80          ORG      $3A00          beginadres van objectprogramma
90 EBUF      EQU      $1000          beginadres EPROM-buffer
100 UITBUF    EQU      $1800          eerste locatie na EPROM-buffer
110 DATA     EQU      $FF           weg te schrijven data
120 *
130 INIFF     LDX      #EBUF          initialiseer bufferpointer
140          LDA      #DATA          zet data $FF in bufferlocatie en
150 VERDER    STA      ,X+           verhoog bufferpointer
160          CMPX     #UITBUF        buffer vol?
170          BNE      VERDER          zo niet, schrijf nog een byte
180          SWI      stop
190          END      INIFF

```

U ziet dat wederom in het sourceprogramma geen enkel getal voorkomt, zodat eventuele veranderingen in de waarden eenvoudig aan te brengen zijn door alleen de EQU-directives te wijzigen. Let op dat de instructies 130, 140 en 160 immediate addressing instructies zijn (!), omdat hier de operand de data is en niet het adres van de data. Zo moet X geladen worden met het getal (het adres) EBUF (= \$1000) en niet met de inhoud van die geheugenplaats.

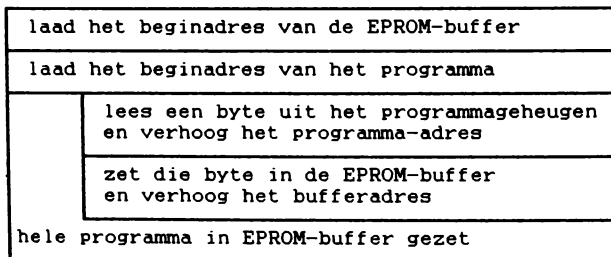


Fig. 6.10 Structogram bij het plaatsen van een programma in een EPROM-buffer

Voorbeeld 6.12 (Block transfer)

We willen nu een objectprogramma laden in een EPROM. Hiertoe

moeten we de objectfile in de EPROM-buffer zetten, nadat deze buffer met behulp van het programma INIBUF is gevuld met enen. Stel het objectprogramma staat op de adressen \$4C00 tot en met \$5261. Dit programma past dus ruimschoots in de EPROM-buffer. Ook hier onderscheiden we duidelijk een herhalingsstructuur, immers er moeten een aantal bytes worden verplaatst van het programmeergeheugen naar de EPROM-buffer. Fig. 6.10 toont het structogram.

```

10          NAM      BLOCK MOVE
20 *Progr.:  J.G. Rouland
30 *Revisie 0      Datum: 02-03-1986
40 *
50 *Dit programma verplaatst de inhoud van een buffer ($4C00
60 *t/m $5261) naar de EPROM-buffer (beginnend op $1000).
70 *
80 *
90          ORG      $3B00          beginadres objectcode
100 FIRST    EQU      $4C00          beginadres programmapointer
110 LAST     EQU      $5261          eindadres programmapointer
120 EBUF     EQU      $1000          beginadres EPROM-buffer
130 *
140 BLMOVE   LDX      #EBUF          initialiseer EPROM-bufferpointer
150         LDY      #FIRST          initialiseer programmapointer
160 NEXT     LDA      ,Y+            zet programmapointer in EPROM-buffer
170         STA      ,X+            en verhoog beide pointers
180         CMPX     #LAST+1         hele file verplaatst?
190         BNE      NEXT           zo niet, verplaats volgende byte
200         SWI      stop
210         END      BLMOVE

```

Merk op dat er twee pointerregisters worden gebruikt bij dit programma. We zullen ook een oplossing geven waarbij er slechts één pointer nodig is:

```

10          NAM      BLOCK MOVE
20 *Progr.:  J.G. Rouland
30 *Revisie 1      Datum: 02-03-1986
40 *
50 *Revisie 1 gebruikt één pointerregister minder dan revisie 0.
60 *
70 *Dit programma verplaatst de inhoud van een buffer ($4C00
80 *t/m $5261) naar de EPROM-buffer (beginnend op $1000).
90 *
100 *
110         ORG      $3B00          beginadres object file
120 EBUF     EQU      $1000          beginadres EPROM-buffer
130 LAATST   EQU      $1661          laatste te vullen locatie van
140 *         EPROM-buffer
150 DISPL    EQU      $3C00          afstand van EPROM-buffer tot
160 *         programma-buffer
170 *

```

180	BLMOVE	LDX	#EBUF	initialiseer EPROM-bufferpointer
190	NEXT	LDA	DISPL,X	lees programmabyte, zet deze in
200		STA	,X+	EPROM-buffer en verhoog pointer
210		CMPX	#LAATST+1	hele programma verplaatst?
220		BNE	NEXT	zo niet, verplaats volgende byte
230		SWI		stop
240		END	BLMOVE	

De laatste locatie in de EPROM-buffer die moet worden gevuld, heeft als adres $\$1000 + (\$5261 - \$4C00) = \$1000 + \$0661 = \1661 .

6.4 Logische instructies

6.4.1 Algemeen

Logische instructies worden veel gebruikt bij besturingen. Denk hierbij onder andere aan de besturing van een CV-installatie, een verkeersregeling, enzovoorts. Bij besturingen moeten kleppen open of dicht worden gezet, motoren gestart en gestopt, lampen aangestoken of gedoofd, enzovoorts. Deze besturingen vinden plaats door middel van signalen. Zo zal de rode lamp branden als het signaal ROOD gelijk aan 1 is. Voor ROOD=0 zal de rode lamp uit zijn. Elk signaal kan door de microprocessor worden ingesteld en veranderd. Dit gebeurt door de bits, die met de verschillende signalen overeenkomen, de juiste waarden te geven. Hiervoor kan een load-instructie worden gebruikt om de gewenste signaalwaarden in een accumulator te zetten en vervolgens de inhoud van deze accumulator met een store-instructie naar het betreffende input/output-apparaat te sturen. Na deze store-instructie nemen de signalen de ingestelde waarden aan. Willen we de waarden van een of meerdere signalen later wijzigen, dan kan dit eenvoudig gebeuren door met behulp van de logische instructies een of meerdere bits te veranderen.

Bij besturingen moeten ook signalen worden getest. Bij een lift kunt u op elke verdieping een knop indrukken om de lift aan te vragen. Verder bevinden zich in de lift drukknoppen om de gewenste etage op te geven. De signalen van al deze drukknoppen moeten door de liftbesturing getest worden, opdat de besturing de lift naar de juiste etages dirigeert. Deze signalen kunnen weer via de input/output door de microprocessor worden gelezen met behulp van een load-instructie. Vervolgens kan de waarde van elke bit (en dus van elk signaal) getest worden met behulp van logische instructies.

We kunnen de logische instructies onderverdelen in vijf groepen:

1. reset-instructies
2. set-instructies

3. inverteerinstructies
4. schuifinstructies
5. bittest-instructies

6.4.2 Reset-instructies

Met behulp van reset-instructies kunt u signalen 0 maken. Dit kan toegepast worden om lampen te doven, kleppen te sluiten, motoren te stoppen, enzovoorts.

De MC6809 kent de volgende reset-instructies:

```
CLR
CLRA    ANDA
CLRB    ANDB
        ANDCC
```

Een CLear-instructie heeft tot gevolg dat alle bits van de geadresseerde geheugenlocatie, respectievelijk van de bedoelde accumulator, 0 worden gemaakt (zie ook het overzicht van fig. 6.11).

De clear-instructie reset de overflow flag en de carry flag van het condition code register altijd. Vanzelfsprekend wordt de zero bit geset, immers het resultaat van de bewerking is 0, omdat alle bits gereset worden. De negative flag wordt gereset, omdat het resultaat positief is, immers bit 7 van het resultaat is 0. De half carry bit wordt niet veranderd.

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
CLRA CLRB CLR		0F 6 2	6F 6+ 2+	7F 7 3	4F 2 1 5F 2 1	(A) :=0 (B) :=0 (M) :=0	*0100 *0100 *0100

Fig. 6.11 Overzicht van de CLear-instructies

Voorbeeld 6.13

```
CLRB          (B) :=0
CLR    $FF01  ($FF01) :=0
CLR    <$A6,Y ($FFA6+(Y)) :=0
CLR    [$1000] maak alle bits 0 op de geheugen-
                  locatie waarvan het adres staat
                  op de locaties $1000 en $1001
```

Voorbeeld 6.14

Stel dat we acht lampen willen doven en dat deze lampen aangestuurd worden door de bits die zich bevinden op de

geheugenlocatie (die dienst doet als input/output) met het symbolische adres LAMPEN. Middels een EQU-directive kunt u de assembler meedelen door welk fysisch adres LAMPEN moet worden vertaald. Deze lampen kunnen gedoofd worden met de instructie

CLR LAMPEN

We hoeven dus niet achtereenvolgens de instructies

LDA #0

STA LAMPEN

te gebruiken. Dit kost alleen extra geheugenruimte en extra executietijd. Bovendien kan LDA #0 altijd worden vervangen door de kortere en snellere instructie CLRA.

Natuurlijk wilt u niet steeds alle bits resetten. Vaak moeten er slechts een of enkele signalen gereset worden. In dat geval moet u de AND-instructie gebruiken. Deze instructie voert de logische and-functie uit op (acht) paren van bits. De instructie

ANDA M

heeft tot gevolg dat elke bit van A "ge-and" wordt met elke overeenkomstige bit van de geheugenlocatie met het adres M. Het resultaat van deze bewerking wordt in A geplaatst:

$(A_i) := (A_i) \text{ AND } (M_i)$ voor $i = 0 \text{ t/m } 7$.

De inhoud van de geheugenlocatie M verandert bij de and-instructie niet.

De and-operatie komt overeen met de functie van een logische AND-gate. De uitgang van een AND is alleen dan 1 als alle ingangen 1 zijn.

Voorbeeld 6.15

Nemen we als voorbeeld dat $(A) = \$AA$ en $(M) = \$F0$, dan zal de instructie ANDA M het volgende resultaat hebben:

$(A) = \%1010\ 1010$

$(M) = \%1111\ 0000$ AND

$(A) := \%1010\ 0000$

Uit voorbeeld 6.15 blijkt, dat door te "anden" met een 0 de overeenkomstige bit altijd gereset wordt, of deze nu 0 of 1 was. Er geldt immers $0 \text{ AND } 0 = 0$ en $1 \text{ AND } 0 = 0$; het resultaat is in beide gevallen 0. Verder ziet u dat "anden" met 1 de overeenkomstige bit niet verandert. Ook dit klopt, immers $0 \text{ AND } 1 = 0$ en $1 \text{ AND } 1 = 1$; de eerste bit blijft ongewijzigd.

Samenvattend kunnen we zeggen:

$q \text{ AND } 0 = 0$

$q \text{ AND } 1 = q$

Het getal waarmee de and-operatie wordt uitgevoerd (in voorbeeld 6.15 de inhoud van M), wordt het masker genoemd. De waarde van het masker bepaalt welke bits worden gereset bij een and-instructie en welke onaangetast blijven. Een 0 in het masker reset de overeenkomstige bit, een 1 laat de betreffende bit onveranderd.

We hebben nu de mogelijkheid om een of meerdere bits te resetten.

Voorbeeld 6.16

Willen we de lampen 2, 5 en 7 doven en de toestand van de overige lampen ongewijzigd laten, dan kan dit geschieden met behulp van de volgende instructies:

```
LDA  LAMPEN      lees de toestand van de lampen
ANDA  #01011011  reset A2, A5 en A7
STA  LAMPEN      doof de lampen 2, 5 en 7
```

Vergeet de store-instructie niet. De and-instructie verandert alleen de inhoud van accumulator A, niet de inhoud van de geheugenplaats LAMPEN.

Zie fig. 6.12 voor het overzicht van de and-instructies.

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ #+	extend op ~ #	inher op ~ #		
ANDA	84 2 2	94 4 2	A4 4+ 2+	B4 5 3		(A) := (A) AND (M)	***0-
ANDB	C4 2 2	D4 4 2	E4 4+ 2+	F4 5 3		(B) := (B) AND (M)	***0-
ANDCC	1C 3 2					(CC) := (CC) AND M	bbbbbb

Opm.: b. Flags worden gereset of veranderen niet, afhankelijk van de waarde van het masker.

Fig. 6.12 Overzicht van de AND-instructies

Bij de and-instructie wordt de N-bit geset, indien het resultaat van de bewerking negatief is, met andere woorden als bit 7 van het resultaat 1 is. In het andere geval wordt N gereset. Z wordt geset als alle bits van de uitkomst van de and-operatie 0 zijn. Is dit niet zo, dan wordt Z gereset. De overflow bit wordt altijd gereset bij een and-instructie. De bits H en C worden niet beïnvloed.

Merk op dat de instructie ANDCC alleen kan optreden met immediate addressing. Vandaar dat in de beschrijving staat

(CC) := (CC) AND M

in plaats van (M). M is hier de immediate operand, de data, niet het adres van de data. ANDCC is bedoeld om bepaalde flags in het condition code register te resetten.

Voorbeeld 6.17

```
ANDCC  #MASKER      (CCi) := (CCi) AND MASKERi voor
                    i = 0 t/m 7
ANDCC  #%01110100   reset de bits E, N, V en C van
                    het condition code register
ANDA   [B,Y]         "and" (A) met de inhoud van de
                    geheugenlocatie met het adres
                    ((B)+(Y)) en ((B)+(Y)+1)
```

ANDB >\$A6,X

"and" (B) met de inhoud van de geheugenplaats met het adres \$00A6+(X); offset ">\$A6" betekent offset=\$00A6; niet-significante nullen zijn weggelaten en ">"-teken duidt op 2-bytes offset

6.4.3 Set-instructies

Met set-instructies kunnen we bepaalde bits 1 maken. Dit kan nuttig zijn als we bijvoorbeeld lampen willen ontsteken, kleppen open zetten of motoren starten.

De MC6809 kent de volgende set-instructies:

ORA
ORB
ORCC

De OR-instructie voert een or-functie uit op (acht) paren van bits. Hij heeft zodoende dezelfde werking als acht OR-gates (elk met twee ingangen). De uitgang van een OR-gate is 1 als minstens een van de ingangen 1 is. De uitgang is alleen dan 0 als alle ingangen 0 zijn. Passen we dit toe op de bitparen bij de instructie

ORA M

dan vinden we

(Ai) := (Ai) OR (Mi) voor i = 0 t/m 7.

Voorbeeld 6.18

Voor (A)=\$AA en (M)=\$F0 kunt u het resultaat van de or-instructie berekenen:

(A) = %1010 1010
(M) = %1111 0000
(A) := %1111 1010

Uit voorbeeld 6.18 blijkt dat die bits worden geset, waarvoor in het masker een 1 staat. De bits waarvoor de overeenkomstige bits in het masker 0 zijn, worden niet veranderd. Op deze wijze hebben we de mogelijkheid om een of meerdere bits te setten.

Er geldt:

q OR 0 = q
q OR 1 = 1

Voorbeeld 6.19

Veronderstel dat u de lampen 2, 3 en 7 moet ontsteken, zonder de toestand (het aan of uit zijn) van de andere lampen te beïnvloeden. U kunt dit bereiken met behulp van de instructies

LDB	LAMPEN	lees de toestand van de lampen
ORB	%10001100	set B2, B3 en B7
STB	LAMPEN	ontsteek de lampen 2, 3 en 7

Fig. 6.13 geeft de or-instructies met hun addressing modes.

instr	addressing modes					beschrijving	(CC) bit 53210 HNZVC
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
ORA	8A 2 2	9A 4 2	AA 4+ 2+	BA 5 3		(A) := (A) OR (M)	---0-
ORB	CA 2 2	DA 4 2	EA 4+ 2+	FA 5 3		(B) := (B) OR (M)	---0-
ORCC	1A 3 2					(CC) := (CC) OR M	ffff

Opm.: f. Flags worden geset of veranderen niet, afhankelijk van de waarde van het masker.

Fig. 6.13 Overzicht van de OR-instructies

Bij de instructies ORA en ORB worden de flags N en Z op de bekende wijze aangepast aan het resultaat van de bewerking. V wordt altijd gereset, terwijl de andere bits niet worden gewijzigd.

Bij de ORCC-instructie worden die flags geset, waarvoor op de overeenkomstige posities in het masker een 1 staat. De overige flags worden niet beïnvloed.

ORCC bestaat (bij de MC6809) alleen als immediate addressing instructie. Deze instructie heeft als doel bepaalde flags van het condition code register te setten.

Voorbeeld 6.20

ORCC	#MASKER	(Bi) := (Bi) OR MASKERi voor i = 0 t/m 7
ORCC	#\$01010001	set de flags F, I en C van het condition code register
ORA	PATR	(Ai) := (Ai) OR (PATRi) voor i = 0 t/m 7
ORB	[\$1000]	"or" (B) met de inhoud van de geheugenlocatie waarvan het adres staat op \$1000 en \$1001

6.4.4 Inverteer instructies

Er kunnen zich situaties voordoen, waarbij het handig is als we de mogelijkheid hebben om bits te inverteren. Denk bijvoorbeeld aan de alarmlichten bij een spoorwagengang. Deze moeten gaan knipperen als er een trein aankomt. Dit kan worden bereikt door de lampen achtereenvolgens (met een bepaalde tussenpoos) aan en uit te schakelen, ofwel door het stuursignaal van deze lampen telkens te inverteren. Ook moeten we soms een reeks pulsen (een zogenaamde pulstrein) verzenden, bijvoorbeeld om een bepaald apparaat te testen. Hiertoe kunnen we het betreffende signaal telkens inverteren.

De MC6809 kent de volgende inverteerinstructies:

COM
COMA EORA
COMB EORB

De (one's) Complement-instructie

COM M

heeft tot gevolg dat elke bit van de geheugenplaats met adres M wordt geïnverteerd:

(Mi) := NOT (Mi) voor i = 0 t/m 7.

Voorbeeld 6.21

Nemen we (M) = %1100 1010, dan geldt na de instructie COM M dat (M) = %0011 0101. U ziet dat elk bit geïnverteerd is.

Fig. 6.14 laat de COM-instructies zien.

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ #+	extend op ~ #	inher op ~ #		
COMA					43 2 1	(A) := NOT (A)	---01
COMB					53 2 1	(B) := NOT (B)	---01
COM		03 6 2	63 6+ 2+	73 7 3		(M) := NOT (M)	---01

Fig. 6.14 Overzicht van de Complement-instructies

Bij de complement-instructies worden de bits N en Z van het condition code register op de bekende wijze aangepast aan het resultaat van de bewerking. V wordt altijd gereset en C wordt altijd geset. De overige flags worden niet aangetast.

Voorbeeld 6.22

COMA (Ai) := NOT (Ai) voor i = 0 t/m 7
COM BYTE (BYTEi) := NOT (BYTEi) voor
i = 0 t/m 7
COM <\$C3 inverteer elke bit op de geheugen-
plaats waarvan het high-adres in
DP staat en het low-adres de
waarde \$C3 heeft

Natuurlijk wilt u niet altijd alle bits invertieren. Vaak moeten er slechts een of enkele bits worden geïnverteerd. In die gevallen dient u de exclusive-or-instructie toe te passen.

De Exclusive-OR-instructie

EORA M

bepaalt het resultaat van de exclusive-or-functie van de (acht) bitparen van (A) en (M). Dit resultaat wordt in accumulator A gezet, waardoor evenwel de oorspronkelijke inhoud van

accumulator A verloren gaat. De inhoud van geheugenplaats M (het masker) blijft ongewijzigd.

(Ai) := (Ai) EOR (Mi) voor $i = 0$ t/m 7

De exclusive-or-instructie heeft een soortgelijk effect als een exclusive-OR-gate. De uitgang van een exclusive-OR-gate is 1 als slechts een van de ingangen 1 is. De uitgang is 0 als geen van de ingangen 1 is of als er meerdere ingangen 1 zijn.

Voorbeeld 6.23

Voor (A)=\$AA en (M)=\$F0, geldt:

(A) = %1010 1010

(M) = %1111 0000 EOR

(A) := %0101 1010

Uit voorbeeld 6.23 blijkt dat een 1 in het masker de overeenkomstige bit in de accumulator inverteert, immers $0 \text{ EOR } 1 = 1$ en $1 \text{ EOR } 1 = 0$. Een 0 in het masker laat de overeenkomstige bit in de accumulator ongewijzigd, immers $0 \text{ EOR } 0 = 0$ en $1 \text{ EOR } 0 = 1$.

Samenvatend kan gesteld worden:

$q \text{ EOR } 0 = q$

$q \text{ EOR } 1 = \text{NOT } q$

Voorbeeld 6.24

Stel een verkeerslichtenregeling wordt op nachtbedrijf geschakeld. Alle lampen moeten worden gedoofd. De oranje lampen moeten echter gaan knipperen. Stel dat deze lampen worden gestuurd door de bits 1 en 4 van locatie HOOFD voor de hoofdstraat en door de bits 1 en 4 van locatie ZIJ voor de zijstraat. De rode lampen worden gestuurd door de bits 2 en 5 en de groene door de bits 0 en 3 van de betreffende locaties. De bits 6 en 7 moeten ongewijzigd worden gelaten. Het programma voor nachtbedrijf kan er globaal als volgt uit zien:

10	LDA	HOOFD	lees toestand van hoofdstraat
20	ANDA	%%11000000	bereid doven van lichten voor
30	LDB	ZIJ	lees toestand van zijstraat
40	ANDB	%%11000000	bereid doven van lichten voor
50	AANUIT	EORA	laat oranje lichten van
60	EORB	%%00010010	hoofdstraat en zijstraat
70	STA	HOOFD	knipperen (en doof de
80	STB	ZIJ	overige lichten)
90	BRA	AANUIT	

Daar de instructies van de microprocessor in enkele microseconden worden uitgevoerd, moet er een wachttijd worden ingebouwd in de programmalus. Dit kan geschieden tussen de instructies 80 en 90. De lichten moeten knipperen met een frequentie van ongeveer 1 Herz. Dit betekent dat de lampen telkens een halve seconde aan en een halve seconde uit moeten zijn. Verder moet het nachtbedrijf natuurlijk onderbroken kunnen worden. Hiertoe moet er een test worden opgenomen (vlak

voor instructie 90). Verder moet de onvoorwaardelijke spronginstructie BRA worden vervangen door een voorwaardelijke sprong, zodat alleen wordt teruggesprongen naar AANUIT, als het nachtbedrijf gehandhaafd moet blijven. Over de voorwaardelijke spronginstructies vindt u meer in paragraaf 6.6.

Voor het overzicht van de exclusive-or-instructies raadplegen we fig. 6.15.

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ #+	extend op ~ #	inher op ~ #		
EORA	88 2 2	98 4 2	A8 4+ 2+	B8 5 3		(A) := (A) EOR (M)	***0-
EORB	C8 2 2	D8 4 2	E8 4+ 2+	F8 5 3		(B) := (B) EOR (M)	***0-

Fig. 6.15 Overzicht van de Exclusiv-OR-instructies

Bij de exclusive-or-instructies worden de flags N en Z aangepast aan de waarde van de uitkomst van de bewerking. Het V-bit wordt altijd gereset en de overige bits blijven onveranderd.

Voorbeeld 6.25

EORB	OFFSET,S	inverteer de bits van B overeenkomstig het masker dat staat op de geheugenlocatie met het adres OFFSET+(S)
EORA	##00010110	inverteer de bits A1, A2 en A4

6.4.3 Schuifinstructies

De schuifinstructies hebben tot gevolg dat de informatie in een register of geheugenplaats één positie naar links of naar rechts schuift. Met behulp van deze instructies kunnen bitpatronen worden gereorganiseerd.

De MC6809 kent de volgende schuifinstructies:

LSL	LSR	ROL	ROR
LSLA	LSRA	ROLA	RORA
LSLB	LSRB	ROLB	RORB

Fig. 6.16 geeft een overzicht van de werking van de verschillende schuifinstructies, toegepast op de inhoud van een geheugenplaats met adres M.

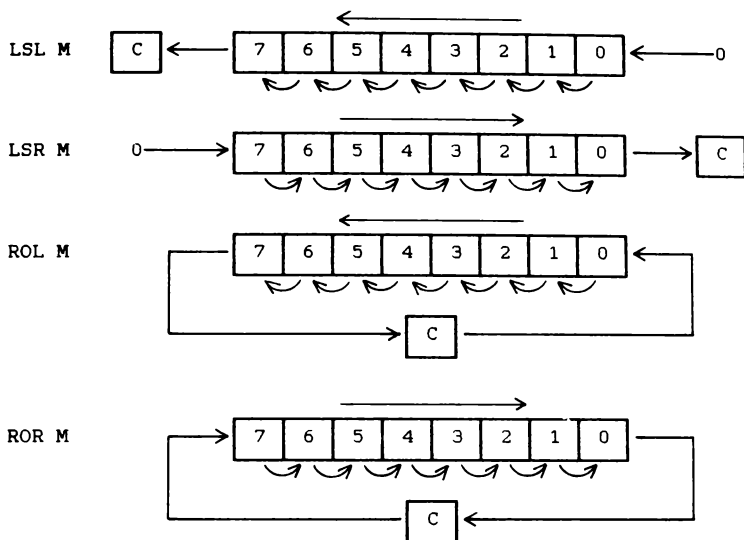


Fig. 6.16 Overzicht van de werking van de schuifinstructies

Bij de Logic Shift Left instructies worden tegelijk alle bits één positie naar links geschoven. Dit is in de richting van de meest significante bit (bit 7). Hierbij wordt de oorspronkelijke waarde van bit 7 opgeslagen in de carry bit van het condition code register. Bit 0 wordt gereset. De Logic Shift Right instructies hebben hetzelfde effect, maar nu wordt er in tegengestelde richting geschoven. Elke bit schuift één plaats naar rechts, dus in de richting van de minstwaardige bit (bit 0). De oude waarde van bit 0 wordt in bit C van het condition code register bewaard. Bit 7 krijgt de waarde 0.

De Rotate Left instructies zorgen ervoor dat elk bit één plaats naar links schuift, net als bij de logic shift left instructies. Nu wordt bit 0 niet gereset, maar geladen met de oorspronkelijke waarde van de carrybit van het condition code register. De nieuwe waarde van C wordt gelijk aan de oorspronkelijke waarde van bit 7.

Bij de Rotate Right instructies schuiven alle bits één positie naar rechts. In tegenstelling tot bij de logic shift right instructies wordt bit 7 hierbij niet gereset, maar neemt de oude waarde van de carry aan. De carry bit zelf wordt overschreven met de vorige waarde van bit 0.

Met behulp van de rotate-instructies kan de informatie in een geheugenplaats of in een accumulator worden rondgeschoven. Bedenk wel dat dit gebeurt via de carry bit van het condition code register. Er wordt dus in feite over 9 bits geroteerd!

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ #+	extend op ~ #	inher op ~ #		
LSLA LSLB LSL		08 6 2	68 6+ 2+	78 7 3	48 2 1 58 2 1	A B C ← 76543210 M	?**** ?**** ?****
LSRA LSRB LSR		04 6 2	64 6+ 2+	74 7 3	44 2 1 54 2 1	A B 0 → 76543210 → C M	-0*-* -0*-* -0*-*
ROLA ROLB ROL		09 6 2	69 6+ 2+	79 7 3	49 2 1 59 2 1	A B ← 76543210 ← M → C →	----- ----- -----
RORA RORB ROR		06 6 2	66 6+ 2+	76 7 3	46 2 1 56 2 1	A B → 76543210 → M ← C ←	---*--- ---*--- ---*---

Fig. 6.17 Overzicht van de schuifinstructies

Fig. 6.17 laat het overzicht zien van de schuif- en roteerinstructies met hun adresseringsmethoden.

Alle schuifinstructies setten of resetten de flags N, Z en C overeenkomstig het resultaat van de bewerking. De LSR-instructies schuiven een 0 in bit 7, zodat de tekenbit positief wordt en de negative flag altijd wordt gereset.

Bij het schuiven naar rechts wordt de V-bit niet gewijzigd. Er kan in dit geval niet van overflow worden gesproken. Bij het schuiven naar links wordt de V-bit geset, indien de tekenbit (bit 7) bij het schuiven verandert. Zo niet, dan wordt de V-bit gereset. We komen hierop nog terug in paragraaf 6.5.

De H-bit blijft bij de schuifinstructies onaangetast, behalve bij de LSL-instructies, waar de waarde van H niet bekend is. Dat wil zeggen H kan wel veranderen, maar aan de waarde van H mag geen betekenis worden toegekend.

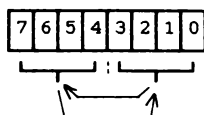
Voorbeeld 6.26

ROLA		roteer (A) via C één plaats naar links
LSRB		schuif (B) één plaats naar rechts; (B7) := 0
ROR	>\$5A	roteer (\$005A) via C één plaats naar rechts
LSL	[A,Y]	schuif de inhoud van de geheugenlocatie met het adres ((A)+(Y)) en ((A)+(Y)+1) één plaats naar links; bit 0 van deze locatie wordt gereset

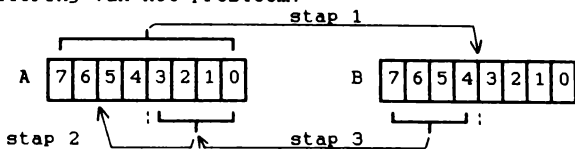
probleemstelling:

voorbeeld: beginsituatie 1100 1010
eindsituatie 1010 1100

accumulator A



detaillering van het probleem:



oplossing:

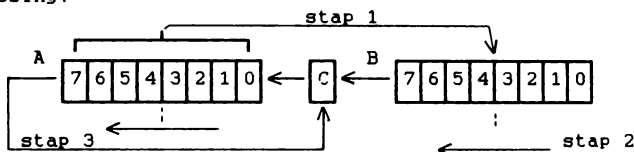


Fig. 6.18 Het verwisselen van twee nibbles

Voorbeeld 6.27 (Nibbles verwisselen)

Schuifinstructies worden vrij vaak toegepast. Stel dat we de linker en de rechter nibble van accumulator A moeten verwisselen (zie fig. 6.18 bovenaan). De rechter nibble kan eenvoudig op de plaats van de linker nibble worden gezet door vier keer naar links te schuiven. De linker nibble gaat hierbij verloren. Derhalve slaan we deze eerst in accumulator B op. De linker nibble van B moeten we nu zien te plaatsen in het rechter deel van A (zie de middelste tekening van fig. 6.18). Achtereenvolgens moeten de stappen 1, 2 en 3 gezet worden. Er bestaat echter geen instructie om de inhoud van het ene register in het andere register te schuiven. Alleen de carry kan in een register geschoven worden en wel met behulp van een rotate-instructie. We moeten derhalve het linker deel van B via de carry in het rechter deel van A schuiven. Dit leidt tot de oplossing van het probleem, zoals in de onderste tekening van fig. 6.18 is aangegeven. Hierbij wordt (A) eerst gekopieerd in B (stap 1). Vervolgens moet B7 in de carry gezet worden (stap 2), teneinde deze bit in A te kunnen schuiven. Dit gebeurt bij stap 3, waarbij de inhoud van A wordt geroteerd naar links, zodat tegelijkertijd de rechter nibble van A één plaats naar links schuift. De stappen 2 en 3 moeten hierna nog drie keer worden herhaald. Dan is de inhoud van de linker nibble van B in de rechter helft van A geschoven, terwijl de rechter helft van A in de linker nibble van A is geschoven.

6.4.6 Bittest-instructies

Bittest-instructies zijn bedoeld om na te gaan of bepaalde bits 0 dan wel 1 zijn. U moet u voorstellen dat deze bits verbonden zijn met sensoren, bijvoorbeeld niveaudetectors van vloeistoffen in vaten, rookdetectors, waterdetectors en dergelijke.

De MC6809 kent de volgende bit-testinstructies:

BITA

BITB

De BITtest-instructie

BITA M

voert een logische and-functie uit op de (acht) bitparen van de accumulatorinhoud en de operand:

(Ai) AND (Mi) voor $i = 0$ t/m 7.

Het (enige) verschil met de and-instructie is dat bij de bittest-instructie de inhoud van de accumulator niet verandert. De uitkomst van de bewerking wordt niet als zodanig opgeborgen. Wel worden de bits van het condition code register aangepast aan het resultaat van de bewerking. Hieruit kunnen dan conclusies worden getrokken omtrent het 0 of 1 zijn van bepaalde bits.

De N-flag zegt iets over bit 7 van de accumulator. Als $N=0$ is bit 7 gelijk aan 0, immers de inhoud van de accumulator is dan positief. Geldt $N=1$, dan is ook bit 7 gelijk aan 1, immers nu is de inhoud van de accumulator negatief. Zoals u al weet, fungeert bit 7 (bij 8-bits getallen) ook als tekenbit. Nu wordt de N-flag bij vrijwel alle instructies aangepast aan de tekenbit van het resultaat van de instructie. Dit betekent dat de bittest-instructie voor bit 7 overbodig is na de meeste instructies. Alleen als bit 7 moet worden getest na bijvoorbeeld een exchange- of pull-instructie, is de bittest-instructie voor bit 7 nodig. De exchange- en pull-instructies beïnvloeden de N-bit niet.

Belangrijker bij het testen van bits is de zero flag. Dit zal uit het volgende blijken.

Om na te gaan of een bepaalde bit 0 dan wel 1 is, maakt u alle andere bits 0 door het juiste masker te kiezen, waarmee de and-operatie (tijdens de bittest-instructie) wordt uitgevoerd. Is het resultaat van deze and-operatie gelijk aan 0, dan is ook de te testen bit 0. In dit geval is de Z-flag geset. Is de Z-flag gereset na de and-operatie, dan is niet de gehele uitkomst 0. Aangezien alle bits, behalve de te testen bit gereset zijn door de and-operatie, is de te testen bit verantwoordelijk voor het ongelijk aan 0 zijn van het resultaat. Dit betekent dat de te testen bit ongelijk 0, dus gelijk aan 1 is.

Voorbeeld 6.28

Bepaal de waarde van bit 2 van accumulator A. Stel dat (A2)=W. Reset de andere bits van A en laat A2 zelf ongemoeid. Hiertoe moeten we (A) "anden" met het masker %0000 0100, dus

```
(A) = %uuuu uWuu
masker = %0000 0100
resultaat = %0000 0W00 AND
```

Er geldt u&0=0 en W&1=W.

Als W=0 geldt:

resultaat = %00000000. Dit is gelijk aan 0, zodat Z=1.

Als W=1 geldt:

resultaat = %00000100. Dit is ongelijk aan 0, zodat Z=0.

We concluderen hieruit dat de te testen bit gelijk aan 0 is als Z=1 (het nul-zijn is waar) en dat de te testen bit gelijk aan 1 is als Z=0 (het nul-zijn is onwaar).

De bits Z en N (als ook V en C) van het condition code register kunnen met behulp van afzonderlijke spronginstructies worden getest. Dit wordt uitgebreid behandeld in paragraaf 6.6.

Voorbeeld 6.29

Stel we moeten testen of het vloeistofniveau in een tank een bepaalde maximale waarde heeft overschreden. In dit geval geeft de niveaudetector MAX een 1 af, anders een 0. MAX is verbonden met bit 1 van de locatie met het (symbolische) adres TANK. Indien MAX=1, moet het programma worden voortgezet op adres HOOG. In het andere geval wordt het programma zonder te springen voortgezet. Het stukje programma dat voor het bovenstaande zorg draagt, ziet er als volgt uit:

```

--      --
--      --
*      LDA      TANK          lees de toestand van de
                                signalen van de vloeistoftank
                                test het signaal MAX (bit 1)
      BITA      #%000000010
      BNE       HOOG          Branch if Not Equal zero,
*                                oftewel spring als MAX=1
--      --
--      --
HOOG    --      --
--      --
```

Wat opgemerkt is bij de and-instructies betreffende de bits van het condition code register, geldt ook voor de bittest-instructies, met dit verschil dat bij de bittest-instructies de inhoud van de accumulator niet wordt veranderd. Dit kan erg nuttig zijn, omdat de inhoud van de accumulator niet hoeft te worden gered (tijdelijk opgeborgen op een geheugenlocatie). Het is aan te raden niet de and-instructie te gebruiken in plaats van de bittest-instructie, ook al hoeft de inhoud van de accumulator niet bewaard (en dus niet gered) te worden. Gebruik altijd de bittestinstructie als u alleen de waarde van een bit wilt testen en gebruik de and-instructie als u een of meerdere bits wilt resetten. Hierdoor worden uw programma's beter leesbaar.

schuif (A) 4 plaatsen naar links
reset de vier linker bits van B
voeg de enen in A en B samen
zet het resultaat in PACBCD

Fig. 6.21 Structogram bij de omzetting van ASCII naar packed BCD

```

10          NAM      ASCII-BCD-CONVERSIE
20 *Progr.: J.G. Rouland
30 *Revisie 0      Datum: 25-03-1986
40 *
50 *ASCBCD zet de ASCII-cijfers in A en B om in een packed
60 *BCD-formaat in de geheugenlocatie met adres PACBCD.
70 *A bevat het meestwaardige cijfer.
80 *
90 *
100          ORG      $2A00          beginadres objectcode
110 PACBCD    RMB      1             reserveer 1 geheugenplaats voor
120 *                                                het resultaat
130 *
140 ASCBCD    LSLA                      zet de rechter nibble van A
150          LSLA                      op de plaats van de linker
160          LSLA                      nibble en reset de vier
170          LSLA                      rechter bits
180          ANDB    %#00001111      reset de linker nibble van B
190          STB     PACBCD          voeg de inhoud van
200          ORA     PACBCD          A en B samen en berg het
210          STA     PACBCD          resultaat op in PACBCD
220          SWI                      stop
230          END      ASCBCD

```

Regel 190 hebben we nodig, omdat we niet direct de inhoud van de accumulatoren A en B met elkaar kunnen "OR-en". Vandaar dat de inhoud van B tijdelijk wordt opgeslagen op een geheugenplaats. Hiervoor kan PACBCD worden gebruikt. De inhoud van deze locatie wordt even later (bij regel 210) overschreven met de packed BCD-waarde van de twee ASCII-cijfers.

Voorbeeld 6.31 (Serie-parallelomzetting)

Een ingangssignaal IN van één bit komt binnen op bit 3 van een register INPUT met als adres \$FF00. Een kloksignaal KLOK is verbonden met bit 4 van INPUT. Als KLOK=1 is de waarde van IN geldig. Omdat elke bit slechts één keer ingelezen mag worden, moeten we na het inlezen wachten tot KLOK=0. De volgende bit kan worden ingelezen, als KLOK opnieuw 1 is geworden, enzovoorts. Als er 8 bits zijn binnengehaald, worden deze als één byte in een geheugenbuffer opgeslagen. De minstwaardige bit wordt telkens het eerst ontvangen. Na het ontvangen van het

karakter "." moet worden gestopt met de invoer.
 Het beginadres van de geheugenbuffer is \$4000.
 Het structogram van fig. 6.22 laat zien welke acties er
 achtereenvolgens moeten worden ondernomen.

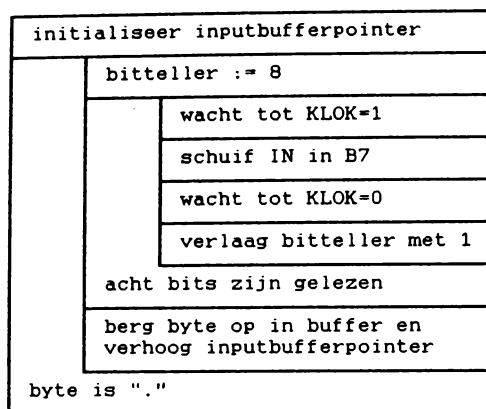


Fig. 6.22 Structogram van serie-parallelozmzetter

Nu het probleem op overzichtelijke wijze gesplitst is in kleine deelproblemen, kan het programma zonder veel moeite geschreven worden.

Om na te kunnen gaan of er al acht bits gelezen zijn, wordt een bitteller gebruikt. Deze krijgt de beginwaarde 8 (zie regel 190 en 200) en wordt telkens na het verwerken van een bit met 1 verlaagd (zie regel 310). Zolang de inhoud van de bitteller ongelijk aan 0 is, zijn er geen acht bits gelezen. Nadat er acht bits zijn binnengehaald, wordt de zo verkregen byte opgeborgen in de buffer (regel 330).

Aangezien eerst bit 0 ontvangen wordt, moet deze bit in B7 worden binnengeschoven. Na acht keer schuiven bevindt deze bit zich op de positie van B0. Er moet dus naar rechts worden geschoven. Hierbij maken we B7 gelijk aan 0 (regel 240). Vervolgens wordt in regel 250 de waarde van IN getest. Is deze gelijk aan 1, dan wordt B7 alsnog geset (regel 270). Anders wordt B7 gelijk aan 0 gelaten. Op deze manier wordt IN binnengeschoven in B.

We gebruiken twee maskers INM en KLOKM. Hiermee wordt de waarde van IN en KLOK getest (regel 250, respectievelijk 220 en 290). De maskers hebben door de EQU-directives (regel 140 en 150) een zodanige waarde gekregen, dat inderdaad de signalen IN en KLOK worden getest. Deze waarden hadden we ook direct in de betreffende instructies kunnen zetten. De gekozen methode maakt het programma gemakkelijker te begrijpen. Bovendien is het eenvoudiger te wijzigen, immers als KLOK en/of IN op een andere

bit aangesloten worden, hoeven alleen de EQU-directives te worden aangepast. Verder vergissen we ons niet zou gauw bij het schrijven van het programma. We hoeven slechts één keer de maskers te definiëren en kunnen ze in de rest van het programma gebruiken, zonder te hoeven nagaan op welke bit ze zijn aangesloten.

```

10      NAM      SERIE-PARALLEL-OMZETTING
20 *Progr.:  J.G. Rouland
30 *Revisie 0      Datum: 25-03-1986
40 *
50 *SERPAR zet serie-informatie (te beginnen met bit 0) om in
60 *parallel-informatie en bergt de aldus verkregen bytes op
70 *in een geheugenbuffer met beginadres $4000. Na het
80 *ontvangen van het teken "." wordt er gestopt.
90 *
100 *
110      ORG      $1300      beginadres objectcode
120 INBUF  EQU      $4000      beginadres inputbuffer
130 INPUT  EQU      $FF00      adres van INPUT register
140 KLOKM  EQU      %00010000  masker om KLOK-waarde te testen
150 INM    EQU      %00001000  masker om IN-waarde te testen
160 BITTEL RMB      1          locatie voor bitteller
170 *
180 SERPAR LDX      #INBUF      initialiseer inputbufferpointer
190 BYTE   LDA      #8          initialiseer
200        STA      BITTEL      bitteller
210 TESTIN LDA      INPUT      wacht tot KLOK=1
220        BITA     #KLOKM      ofwel tot
230        BEQ      TESTIN      IN geldig is
240        LSRB      schuif (B) naar rechts; B7:=0
250        BITA     #INM        berg de waarde
260        BEQ      INNUL      van IN op
270 INEEN  ORB      #%10000000  in B7
280 INNUL  LDA      INPUT      wacht
290        BITA     #KLOKM      tot
300        BNE      INNUL      KLOK=0
310        DEC      BITTEL      verlaag bitteller met 1
320        BNE      TESTIN      spring als geen 8 bits gelezen
330        STB      ,X+        zet byte in buffer, verhoog pointer
340        CMPB     #'.'        spring als byte ongelijk is aan
350        BNE      BYTE      de ASCII-waarde van "."
360        SWI      stop
370        END      SERPAR

```

De label INEEN wordt niet gebruikt. Hij is echter handig om aan te geven, dat IN=1 indien de betreffende instructie wordt uitgevoerd. De leesbaarheid van het programma wordt hierdoor vergroot.

Heel wat eenvoudiger zou het programma eruit hebben gezien als het signaal KLOK verbonden zou zijn met bit 7 van INPUT en het signaal IN met bit 0.

De regels 220 en 230 kunnen dan vervangen worden door

BPL TESTIN

immers bit 7 (KLOK) is de tekenbit. Door te springen als het getal (INPUT) positief is, wordt er in feite gesprongen als KLOK=0 is. Dit gebeurt ook in de regels 220 en 230.

Om dezelfde reden kunnen de regels 290 en 300 worden vervangen door

BMI INNUL

immers in beide gevallen wordt gesprongen als (INPUT) negatief is ofwel als KLOK=1 is.

Verder zouden de regels 240 t/m 270 vervangen kunnen worden door de instructies

LSRA

RORB

Deze instructies hebben tot gevolg dat de carry bit van het condition code register gelijk wordt aan bit A0, ofwel aan het signaal IN, dat vervolgens wordt binnengeschoven in B7.

De regels 140 en 150 zouden tenslotte kunnen vervallen, omdat we de maskers niet meer nodig hebben.

U kunt hieruit concluderen dat de software eenvoudiger kan zijn door een handige keuze van de aansluitingen van de signalen (de hardware-verbindingen).

Voorbeeld 6.32 (Checksum bepalen)

Een eenvoudige checksum van een aantal data kan worden bepaald door de exclusive-or-functie van deze data te berekenen.

Gegeven een buffer met data. De eerste locatie in de buffer moet de checksum gaan bevatten. De tweede locatie bevat het aantal gegevens van de buffer. Zie ook figuur 6.23.

Het structogram met de oplossing voor dit probleem is gegeven in fig. 6.24.

DATBUF	checksum
DATBUF+1	aantal
DATBUF+2	data 1
DATBUF+3	data 2
DATBUF+4	data 3
DATBUF+5	data 4

laad adres van databuffer
reset byte-teller
reset checksum
niet alle data verwerkt
checksum:=checksum EOR data
verhoog byteteller met 1
berg de checksum op

Fig. 6.23 Bufferindeling Fig. 6.24 Structogram voor bepalen van checksum

```

10          NAM      CHECKSUM BEPALEN
20 *Progr.: J.G. Rouland
30 *Revisie 0      Datum: 25-03-1986
40 *
50 *CHESUM bepaalt de checksum (exclusive-or) van de n
60 *databytes in een geheugenbuffer met beginadres DATBUF
70 *(-$8000). De checksum moet in de eerste bufferlocatie
80 *worden geplaatst. De tweede locatie bevat het aantal data
90 *n van de buffer. Daarna volgen de data zelf.
100 *
110 *
120          ORG      $1500          beginadres objectcode
130 DATBUF   EQU      $8000          beginadres databuffer
140 *
150 CHESUM   LDX      #DATBUF+2      X wijst naar de eerste data
160          CLRA                      A is de offset van het adres
170 *                                van de volgende data ten
180 *                                opzichte van het basisadres X
190          CLRB                      reset de checksum
200 NEXT     CMPA     -1,X           spring als (A)=aantal, ofwel
210          BEQ      FINISH          als alle data verwerkt zijn
220          EORB     A,X           bepaal de nieuwe checksum
230          INCA                      A wijst naar de volgende data
240          BRA      NEXT
250 FINISH   STB      -2,X           zet de checksum op de eerste
260 *                                locatie in de buffer
270          SWI                      stop
280          END      CHESUM

```

Voorbeeld 6.33 (Wachttijd)

Vaak zijn er wachttijden nodig alvorens een volgende opdracht kan worden uitgevoerd. Denk bijvoorbeeld aan het verzenden van data. Zou de microprocessor achter elkaar alle data uit een geheugenbuffer bijvoorbeeld naar een printer sturen, dan kan de printer deze datastroom onmogelijk op tijd verwerken. Tussen het versturen van twee bytes moet een bepaalde tijd worden gewacht. Wachttijden kunnen softwarematig worden gerealiseerd. Hiertoe wordt er een programmalus gemaakt, waarin niets anders hoeft te gebeuren dan het verlagen van een teller. Is de teller 0 geworden, dan wordt de programmalus verlaten. Hoe hoger de tellerstand, hoe vaker de programmalus wordt doorlopen en hoe groter de wachttijd is. De wachtlus kan er als volgt uitzien:

```

10          LDD      #AANTAL        (D) := aantal keren dat de lus
20 *                                doorlopen moet worden
30 WACHT     SUBD     #1             verlaag (D) met 1
40          BNE      WACHT          spring als (D) ongelijk aan 0

```

In appendix B vindt u de executietijd van elke instructie, uitgedrukt in cyclustijden van de klokpuls. Stel dat een klokcyclus 1 μ s duurt, dan vindt u voor de duur van de gebruikte instructies:

10: 3 μ s (deze instructie wordt slechts 1 keer uitgevoerd)
30: 4 μ s
40: 3 μ s

Het één keer uitvoeren van de wachtlus kost dus 7 μ s. Dit betekent dat de totale wachttijd $3 + \text{AANTAL} \cdot 7 \mu$ s bedraagt. Voor $\text{AANTAL} = 14$ bedraagt de wachttijd 101 μ s. Willen we een wachttijd van precies 100 μ s, dan moeten we AANTAL kleiner kiezen en de ontbrekende tijd opvullen met een instructie na de wachtlus. Deze instructie heeft verder geen functionele betekenis en wordt derhalve wel dummy-instructie genoemd. Kiezen we $\text{AANTAL} = 13$, dan moeten we een 6 μ s durende dummy-instructie toevoegen, bijvoorbeeld LDD \$1000, en wel vóór regel 10 of ná regel 40.

6.5 Rekenkundige instructies

6.5.1 Algemeen

De rekenkundige instructies zijn, zoals de naam al aangeeft, bedoeld voor het uitvoeren van berekeningen. Er moeten vaak berekeningen van allerlei aard plaatsvinden. Om bijvoorbeeld het gemiddelde van een aantal meetwaarden te bepalen, moeten eerst al deze meetwaarden worden opgeteld en vervolgens moet de som worden gedeeld door het aantal waarden. Zo zijn er nog veel meer voorbeelden van berekeningen denkbaar.

We onderscheiden de volgende elementaire rekenkundige instructies:

1. optelinstructies
2. aftrek-instructies
3. vermenigvuldiginstructies
4. deelinstructies
5. vergelijk-instructies
6. bijzondere rekenkundige instructies

6.5.2 Optelinstructies

De MC6809 kent de volgende optelinstructies:

INC			
INCA	ADDA	ADCA	ABX
INCB	ADDB	ADCB	
	ADD		

Een INCrement-instructie heeft tot gevolg dat de inhoud van een geheugenplaats, respectievelijk de inhoud van accumulator A of B wordt verhoogd met 1. Zie voor de addressing mode fig. 6.25. De instructie INC be2vloedt de bits N, Z en V, terwijl H en C ongewijzigd blijven. De instructies INCA en ADDA #1 zijn derhalve niet identiek. Houd dit goed in het oog bij het

programmeren! Indien u de carry na het verhogen met 1 als zodanig wilt toepassen, moet u de instructie ADDA #1 gebruiken.

Voorbeeld 6.34

INCA	(A) := (A) + 1
INC <\$CC	tel 1 op bij de inhoud van de geheugenlocatie \$CC van pagina, die wordt aangewezen door DP

instr	addressing modes					beschrijving	(CC) bit 53210 HNZVC
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
INCA					4C 2 1	(A) := (A)+1	----
INCB					5C 2 1	(B) := (B)+1	----
INC		0C 6 2	6C 6+ 2+	7C 7 3		(M) := (M)+1	----

Fig. 6.25 Overzicht van de INCrement-instructies

Bij een ADD-instructie wordt er een willekeurig getal opgeteld bij de inhoud van het gespecificeerde register. Dit betekent dat de instructie

ADDA M

als gevolg heeft dat

(A) := (A) + (M)

Merk op dat de oude inhoud van accumulator A verloren is gegaan. De inhoud van de geheugenplaats verandert niet.

De Add-with-Carry-instructies ADCA en ADCB zijn hetzelfde als ADDA, respectievelijk ADDB, met dien verstande dat behalve een getal ook de momentele waarde van de carry bit van het condition code register bij de gespecificeerde accumulator wordt opgeteld. Dus

ADCB M

levert als resultaat

(B) := (B) + (M) + C

De instructie ABX (Add B to X) is een inherent addressing instructie die alle condition code flags ongewijzigd laat. Hierbij wordt de inhoud van accumulator B als unsigned number opgeteld bij de inhoud van het indexregister X. Deze instructie stelt u in staat een positieve offset (van maximaal 255) op te tellen bij X. Deze instructie kost slechts één byte geheugenruimte en 3 klokcycli executietijd, zoals u kunt zien in fig. 6.26.

De ADD- en de ADC-instructies passen de bits H, N, Z, V en C aan het resultaat van de optelling aan. Deze bits zijn reeds uitgebreid behandeld in hoofdstuk 2 en paragraaf 3.4. Let op dat de instructie ADDD de half-carry H ongewijzigd laat!

Voorbeeld 6.35

```

        ADDD    #$A3D5      (D) := (D) + $A3D5
        ADCA    >$22        (A) := (A) + ($0022) + C
        ADDB    [A,Y]       (B) wordt verhoogd met de
                             inhoud van de geheugenlocatie
                             waarvan het adres staat op de
                             geheugenplaatsen, aangewezen
                             door (A)+(Y) en (A)+(Y)+1
    
```

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		53210 HNZVC
ADDA	8B 2 2	9B 4 2	AB 4+ 2+	BB 5 3		(A) := (A) + (M)	*****
ADDB	CB 2 2	DB 4 2	EB 4+ 2+	FB 5 3		(B) := (B) + (M)	*****
ADDD	C3 4 3	D3 6 2	E3 6+ 2+	F3 7 3		(D) := (D) + (M) (M+1)	-----
ADCA	89 2 2	99 4 2	A9 4+ 2+	B9 5 3		(A) := (A) + (M) + C	*****
ADCB	C9 2 2	D9 4 2	E9 4+ 2+	F9 5 3		(B) := (B) + (M) + C	*****
ABX					3A 3 1	(X) := (X) + (B) uns.	-----

Fig. 6.26 Overzicht van de instructies ADD, Add with Carry en Add B to X

6.5.3 Aftrek-instructies

Aftrek-instructies dienen vanzelfsprekend voor het bepalen van het verschil van twee getallen.

De MC6809 kent de volgende aftrek-instructies:

```

        DEC          NEG
        DECA    SUBA    SBCA    NEGA
        DECB    SUBB    SBCB    NEGB
        SUBD
    
```

Bij de DECrement-instructies wordt 1 afgetrokken van de inhoud van een geheugenlocatie, dan wel van de inhoud van accumulator A of B. Zie fig. 6.27 voor de addressing modes en de beïnvloeding van de condition code bits.

De SUBtract-instructies verlagen de inhoud van de gespecificeerde accumulator met een willekeurig getal. De SuBtract-with-Carry-instructies verminderen de inhoud van A of B met een willekeurig getal en trekken er tevens de huidige waarde van de carry af. Let op dat de data wordt afgetrokken van de inhoud van de accumulator en niet omgekeerd! Lees: "Subtract from A/B/D the operand".

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
DECA DECB DEC					4A 2 1 5A 2 1	(A) := (A) - 1 (B) := (B) - 1 (M) := (M) - 1	---- ---- ----

Fig. 6.27 Overzicht van de DECRement-instructies

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
SUBA SUBB SUBD	80 2 2 C0 2 2 83 4 3	90 4 2 D0 4 2 93 6 2	A0 4+ 2+ E0 4+ 2+ A3 6+ 2+	B0 5 3 F0 5 3 B3 7 3		(A) := (A) - (M) (B) := (B) - (M) (D) := (D) - (M) (M+1)	?**** ?**** -****
SBCA SBCB	82 2 2 C2 2 2	92 4 2 D2 4 2	A2 4+ 2+ E2 4+ 2+	B2 5 3 F2 5 3		(A) := (A) - (M) - C (B) := (B) - (M) - C	?**** ?****

Fig. 6.28 Overzicht van de instructies SUBtract en SuBtract with Carry

Voor wat betreft de adresseringsmethoden en het wijzigen van de flags van het condition code register verwijzen we naar de corresponderende opmerkingen bij de optelinstructies (zie ook fig. 6.28). Er is echter één uitzondering en die betreft de half-carry flag. Na het uitvoeren van de instructie SUBA, SUBB, SBCA of SBCB is de half-carry ongedefinieerd! Dit betekent dat H een waarde aanneemt, die niets met de half-carry functie te maken heeft en die derhalve niet voorspelbaar is (zie ook fig. 6.28).

Voorbeeld 6.36

```

DECB          (B) := (B) - 1
SUBD  NUM     (DH) := (DH) - (NUM) en
              (DL) := (DL) - (NUM+1)
SBCA  [$17]   (A) := (A) - (($0017)) - C

```

De NEGate-instructies plaatsen als het ware een min-teken voor het geadresseerde getal. Het betreffende getal wordt hiertoe van 0 afgetrokken. Dit gebeurt in het two's complement. Was het oorspronkelijke getal positief, dan krijgt het na de negate-instructie een even grote negatieve waarde. Was het getal negatief, dan wordt het na de instructie NEG even groot positief.

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
NEGA NEGB NEG		00 6 2	60 6+ 2+	70 7 3	40 2 1 50 2 1	(A) : --(A) (B) : --(B) (M) : --(M)	?**** ?**** ?****

Fig. 6.29 Overzicht van de NEGate-instructies

Voorbeeld 6.37

NEGA (A) := 0 - (A) = -(A)
 NEG \$24,U (\$0024+(U)) := 0 - (\$0024+(U))

De negate-instructies passen de flags N, Z, V en C aan de uitkomst van de berekening aan, terwijl de half-carry geen betekenis heeft en een onbekende waarde aanneemt (zie fig. 6.29).

6.5.4 Vermenigvuldiginstructies

Ook vermenigvuldigen komt vrij frequent voor. Hiertoe heeft de MC6809 de volgende instructies:

ASL MUL
 ASLA
 ASLB

De Arithmetic Shift Left instructies schuiven het geadresseerde getal één positie naar links. Hierbij wordt de rechter bit (bit 0) gelijk aan 0. Het zal u opvallen dat dit hetzelfde is als bij de Logic Shift Left instructies. Bekijkt u de opcodes van de ASL- en de LSL-instructies (zie appendix B), dan ziet u dat deze gelijk zijn. Dit betekent dat deze instructies identiek zijn. Toch is het handig dat er twee verschillende mnemonics bestaan. Door het juiste gebruik hiervan wordt het programma beter leesbaar. Immers gebruiken we LSL, dan gaat het kennelijk om een reorganisatie van de bits. Gebruiken we daarentegen ASL, dan willen we het betreffende getal met twee vermenigvuldigen. Door naar links te schuiven, krijgt elke bit een waarde die twee keer zo groot is. Dit houdt in dat het hele getal twee keer zo groot wordt.

In het geval dat we met unsigned getallen werken, moet aan de carry het gewicht van bit 8 toegekend worden, immers bit 7 schuift in de carry. Als bit 0 gewicht 1 heeft, zal de carry het gewicht 256 moeten hebben.

Werken we daarentegen met two's complement getallen, dan moet er op de overflow flag worden gelet. Overflow treedt hierbij wederom op bij een fout van de tekenbit. In dit geval is dat een verandering van het teken. Bij vermenigvuldigen met (+)2

mag het teken niet wijzigen. Gebeurt dit toch, dan past het resultaat kennelijk niet meer in het 8-bits register. Dit wordt gesignaleerd doordat V=1 wordt (zie ook paragraaf 2.11).

De half-carry bit heeft bij deze instructies geen betekenis en neemt een onbekende waarde aan. N, Z, V en C worden overeenkomstig het resultaat van de bewerking geset of gereset (zie fig. 6.30).

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ #+	extend op ~ #	inher op ~ #		
ASLA ASLB ASL					48 2 1 58 2 1	A B C←76543210←0 M	***** ***** *****

Fig. 6.30 Overzicht van de Arithmetic Shift Left instructies

Voorbeeld 6.38

LDA	#\$F3	(A) := \$F3 = -13 (bij two's complement) of 243 (bij unsigned)
ASLA		(A) := \$E6 = -26, respectievelijk 230; V:=0, dus de two's complement uitkomst is goed; C:=1, dus de unsigned uitkomst is fout
LDB	#\$16	(B) := \$16 = +22 (bij two's complement en bij unsigned)
ASLB		(B) := \$2C = +44; V:=0 en C:=0, dus de uitkomst is in beide gevallen goed
LDA	#\$80	(A) := \$80 = -128 (bij two's complement) of +128 (bij unsigned)
ASLA		(A) := 0; V:=1 en C:=1, dus het resultaat is zowel voor two's complement als voor unsigned getallen niet juist

De instructie MULTiply is een inherent addressing instructie. Hier wordt het produkt uitgerekend van de unsigned inhouden van de accumulatoren A en B. Zo kunnen we twee willekeurige (8-bits) getallen met elkaar vermenigvuldigen. Het produkt wordt opgeslagen in accumulator D. Hierbij gaan de oorspronkelijke getallen in A en B verloren, immers D bestaat uit de aaneenschakeling van A en B.

MUL (D) := (A) · (B)

Is het produkt gelijk aan 0, dan wordt de zero flag geset.

anders wordt de zero flag gereset. De bits H, N en V blijven onveranderd (zie fig. 6.31).

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
MUL					3D 11 1	(D) := (A) * (B) uns	53210 HNZVC --*-e

Opm.: e. C := bit 7 van accumulator B

Fig. 6.31 De MULtipliy-instructie

De carry krijgt hier een zeer speciale behandeling. Deze neemt de waarde aan van B7 van het produkt (bit 7 van accumulator B). Dit is handig bij het afronden van het produkt op 1 byte:

MUL bereken het produkt van (A) en
ADCA #0 (B) op 8 bits nauwkeurig

Door deze carry (zijnde B7) bij A op te tellen, wordt de waarde van de minstwaardige byte op de juiste wijze afgerond. Nemen we eens aan dat A en B getallen zijn met de binaire komma in het midden. Na vermenigvuldigen staat de binaire komma wederom in het midden, dus precies tussen A en B in. A bevat het hele deel van het produkt en B bevat de fractie. Als B7=1, is de fractie groter of gelijk aan 1/2 en moet er naar boven worden afgerond. Dit betekent dat we A met 1 moeten verhogen. Is B7=0, dan is de fractie kleiner dan 1/2 en moet er naar beneden worden afgerond. In het laatste geval moet A niet worden verhoogd met 1. Daar na de instructie MUL geldt dat C=B7, kan de afronding worden gerealiseerd door C op te tellen bij A met behulp van de instructie ADCA #0.

Er wordt op deze manier steeds afgerond op een hele byte. Afronden doen we om geheugenruimte en executietijd te sparen. Bij vermenigvuldigen worden de getallen snel groter. Vermenigvuldigen van grote getallen kost veel tijd. Er moeten dan een groot aantal vermenigvuldigingen en een groot aantal optellingen gedaan worden. Er kunnen immers slechts twee bytes tegelijk worden vermenigvuldigd. Daar er telkens hele bytes worden vermenigvuldigd en aangezien een geheugenlocatie een gehele byte bevat, heeft het totaal geen zin om ergens binnen een byte af te ronden, omdat dit noch geheugenruimte, noch executietijd spaart. Afronden binnen een byte heeft derhalve geen zin.

6.5.5 Deelinstructies

De MC6809 kent geen echte deelinstructie DIVIDE, zoals de MC68000 die heeft. Wel kan er worden gedeeld door 2 met behulp van de schuifinstructies:

ASR	LSR
ASRA	LSRA
ASRB	LSRB

De Arithmetic Shift Right instructies zorgen voor een verschuiving van één positie naar rechts. Bit 0 schuift in de carry en bit 7 blijft ongewijzigd. Dit laatste impliceert dat de tekenbit niet verandert. Een positief getal blijft positief en een negatief getal blijft negatief. De waarde wordt echter twee keer zo klein. Elke bit krijgt immers het gewicht van de naastliggende rechter bit en die heeft de halve waarde. De carry moet derhalve worden gerekend als bit -1. Heeft bit 0 gewicht 1, dan heeft de carry het gewicht 1/2. Met behulp van de ASR-instructies kunnen we een two's complement getal delen door 2.

Voorbeeld 6.39

LDA	##10001010	(A) := \$8A = -118 (of 138 bij unsigned)
ASRA		(A) := %1100 0101 = \$C5 = -59; V:=0, dus de uitkomst is goed (bij unsigned is \$C5 = 197, wat fout is); C:=0
LDB	##01010111	(B) := \$57 = +87
ASRB		(B) := %0010 1011 = \$2B = +43; V:=0, zodat de uitkomst correct is; C:=1

Willen we een unsigned getal delen door 2, dan moeten we gebruik maken van de instructie Logic Shift Right. Hierbij wordt bit 7 gelijk gemaakt aan 0. Bit 7 wordt derhalve niet opgevat als tekenbit, maar als een bit met het bijbehorende gewicht (128, indien bit 0 het gewicht 1 heeft).

Voorbeeld 6.40

LDA	##10001010	(A) := \$8A = 138 (of -118 bij two's complement)
LSRA		(A) := %0100 0101 = \$45 = 69

Voor wat betreft de adresseringsmethoden en de beïnvloeding van de flags verwijzen we naar de fig. 6.32.

De overflow bit blijft ongewijzigd. De zero bit wordt aangepast aan het resultaat van de bewerking. De carry bit neemt de oude waarde van bit 0 over. Afronden kan dan eenvoudig geschieden door de carry op te tellen bij het gehalveerde getal:

LSRA		deel (A) door 2
ADCA	#0	rond af

De carry heeft het gewicht 1/2 (indien bit 0 het gewicht 1 heeft). Als C=0, is de uitkomst precies goed, als C=1 is de uitkomst 1/2 te klein. Afronden moet nu naar boven plaatsvinden, zodat het resultaat van de deling met 1 moet worden verhoogd. Door de carry bij het quotiënt op te tellen.

wordt de afronding op de juiste wijze gerealiseerd.
 De negatieve bit geeft het teken van het resultaat aan. N wordt dus gelijk aan bit 7 van het resultaat. Dit houdt in dat N altijd 0 wordt na een LSR-instructie.
 De half-carry heeft geen betekenis en neemt een onbekende waarde aan bij de ASR-instructies. H blijft ongewijzigd bij de LSR-instructies.

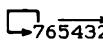
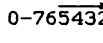
instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ #+	extend op ~ #	inher op ~ #		
ASRA ASRB ASR		07 6 2	67 6+ 2+	77 7 3	47 2 1 57 2 1	A  B M	***- ***- ***-
LSRA LSRB LSR		04 6 2	64 6+ 2+	74 7 3	44 2 1 54 2 1	A B  M	-0*- -0*- -0*-

Fig. 6.32 Overzicht van de instructies ASR en LSR

6.5.6 Vergelijkinstructies

Moet er een maximum worden bepaald van een aantal meetwaarden, of moeten in het algemeen getallen met elkaar worden vergeleken (bijvoorbeeld om na te gaan of de kamertemperatuur hoger is dan de ingestelde waarde), dan kunnen we onze toevlucht zoeken tot de vergelijkinstructies.

De MC6809 kent de volgende vergelijkinstructies:

```

TST
TSTA    CMPA
TSTB    CMPB
        CMPD
        CMPX
        CMPY
        CMPU
        CMPS
  
```

Bij de TeST-instructies wordt het geadresseerde getal vergeleken met 0. Dit gebeurt door 0 van het betreffende getal af te trekken. Het gevolg hiervan is dat de flags in het condition code register worden aangepast aan het resultaat, ofwel aan het getal zelf.
 Wordt de zero bit geset, dan is het onderzochte getal gelijk aan 0. Wordt Z=0, dan is het getal ongelijk aan 0.
 De negatieve bit geeft aan of het getal al dan niet negatief is.
 De overflow bit wordt altijd gereset, terwijl H en C onveranderd blijven (zie ook fig. 6.33).

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
TSTA TSTB TST		0D 6 2	6D 6+ 2+	7D 7 3	4D 2 1 5D 2 1	(A)-0 (B)-0 (M)-0	-**0- -**0- -**0-

Fig. 6.33 Overzicht van de TeST-instructies

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
CMPA	81 2 2	91 4 2	A1 4+ 2+	B1 5 3		(A)-(M)	?****
CMPB	C1 2 2	D1 4 2	E1 4+ 2+	F1 5 3		(B)-(M)	?****
CMPC	10 5 4	10 7 3	10 7+ 3+	10 8 4		(D)-(M) (M+1)	-----
	83	93	A3	B3			
CMPS	11 5 4	11 7 3	11 7+ 3+	11 8 4		(S)-(M) (M+1)	-----
	8C	9C	AC	BC			
CMPU	11 5 4	11 7 3	11 7+ 3+	11 8 4		(U)-(M) (M+1)	-----
	83	93	A3	B3			
CMPX	8C 4 3	9C 6 2	AC 6+ 2+	BC 7 3		(X)-(M) (M+1)	-----
CMPY	10 5 4	10 7 3	10 7+ 3+	10 8 4		(Y)-(M) (M+1)	-----
	8C	9C	AC	BC			

Fig. 6.34 Overzicht van de CoMPare-instructies

Wilt u een getal met een willekeurige waarde vergelijken, dan moet u de CoMPare-instructie gebruiken. Hiermee kan de inhoud van elke accumulator of van elk pointerregister worden vergeleken met een getal of met de inhoud van een geheugenplaats. De instructie CMP werkt eigenlijk hetzelfde als SUB, met dien verstande dat de uitkomst van de aftrekking niet wordt opgeborgen in het betreffende register, maar dat alleen de waarden van de bits van het condition code register worden aangepast. Dit betreft de bits N, Z, V en C. Bit H krijgt bij CMPA en CMPB een onbekende waarde en blijft onaangetaast bij de overige CMP-instructies (zie ook fig. 6.34).

Let op dat de inhoud van het gespecificeerde register wordt vergeleken met de operand. Lees de instructie als "compare (register) with the operand". Bij het resultaat ">" betekent dit, dat (register) > operand en niet andersom!

Door de flags te testen krijgen we informatie over het kleiner, groter of gelijk zijn van twee getallen. Het testen van de diverse flags gebeurt met behulp van de branch-instructies. Hierop zal uitgebreid worden teruggekomen in paragraaf 6.6.

Voorbeeld 6.42

CMPU	#A5DC	vergelijk de inhoud van U met het getal \$A5DC
CMPD	\$4867	vergelijk (D) met (\$4867)(\$4868)
CMPB	\$2AD,Y	vergelijk (B) met (\$02AD+(Y))

6.6.7 Bijzondere rekenkundige instructies

De MC6809 kent nog enkele bijzondere rekenkundige instructies:

DAA	LEAX
SEX	LEAY
	LEAU
	LEAS

De instructie Decimal Adjust A stelt ons in staat BCD-optellingen te doen in accumulator A. Met behulp van de instructie ADD kunnen we getallen optellen. De microprocessor behandelt de getallen echter als gewogen binaire getallen. Laten we de microprocessor getallen optellen die in BCD-code zijn weergegeven, dan kunnen er fouten optreden.

Voorbeeld 6.43

BCD 0110 0101	(=65)	
<u>BCD 0001 0011</u>	(=13)	
BCD 0111 1000	(=78)	Dit is correct!
BCD 0010 1000	(=28)	
<u>BCD 0100 0110</u>	(=46)	
BCD 0110 1110	(=6?)	

Bij de tweede optelling van voorbeeld 6.43 is de rechter nibble van de som geen correct BCD-getal. 9 is het hoogste decimale cijfer. Dit betekent dat er voor grotere cijfers een tiental moet worden doorgegeven aan de meerwaardige (linker) nibble. Met andere woorden we moeten de te grote nibble met 10 verminderen ($\%1110 - \%1010 = \%0100$) en de linker nibble met 1 verhogen ($\%0110 + \%0001 = \%0111$). Nu vinden we de correcte waarde van de som: BCD 0111 0100 (=74). Dit is inderdaad de som van 28 en 46.

Wordt de linker nibble groter dan 9, dan moeten we er 10 van aftrekken en vervolgens de carry 1 maken. De carry stelt bij BCD-getallen een honderdtal voor.

Voorbeeld 6.44

BCD 0110 1001	(-69)	
BCD 0001 1000	(-18)	
BCD 1000 0001	(-81)	De uitkomst is niet goed!

In voorbeeld 6.44 is de uitkomst weliswaar een correct BCD-getal, maar de som klopt niet. $69+18$ is ongelijk aan 81. De oorzaak hiervan ligt in het feit dat er een half-carry is ontstaan. Een half-carry wordt opgeteld bij bit 4 met het gewicht 16. Bij BCD-getallen heeft bit 4 echter slechts het gewicht 10. Dit betekent dat er een getal 16 is getransporteerd naar de linker nibble. Dit getal wordt echter in BCD-code slechts als 10 opgevat. Hierdoor is de uitkomst 6 te laag. Bij het ontstaan van een half-carry moeten we bij de rechter nibble derhalve 6 optellen om de juiste som te krijgen ($\%0001 + \%0110 = \%0111$). De uitkomst wordt nu BCD 1000 0111 ($=87$), hetgeen correct is.

Ontstaat er een carry bij de optelling, dan moeten we 6 optellen bij de linker nibble teneinde de juiste uitkomst te verkrijgen. De carry zelf heeft gewicht 100.

Resumerend kunt u stellen dat er bij het optellen van BCD-getallen enige correcties nodig kunnen zijn om de juiste som te krijgen:

- Is een van de nibbles van de som geen juist BCD-getal, dan moet er van de betreffende nibble 10 worden afgetrokken, terwijl de naastliggende linker nibble met 1 moet worden verhoogd.
Overigens is dit gelijk aan het verhogen van de betreffende nibble met 6, immers $10+6 = 16 = \$10$, zodat (afgezien van de "1-onthouden") geldt $10+6 = 0$, oftewel $-10 = +6$. Verlagen met 10 staat dus gelijk aan verhogen met 6, afgezien van de "1-onthouden". Die hebben we echter hier juist nodig.
- Is er een half-carry of een carry ontstaan, dan moet de rechter, respectievelijk de linker nibble met 1 worden verhoogd.

Het optellen van BCD-getallen zou een vrij moeizame zaak worden, ware het niet dat er een DAA-instructie bestaat. Deze instructie zorgt namelijk voor de nodige correcties na de optelling van twee BCD-getallen. Hiertoe maakt DAA gebruik van de waarden van C en H, zoals die door ADDA of door ADCA ontstaan zijn.

Let op dat DAA alleen betrekking heeft op accumulator A en dus alleen werkt na de instructie ADDA of ADCA!

DAA laat de half-carry ongewijzigd, V wordt altijd gereset, terwijl N, Z en C overeenkomen met het resultaat van de bewerking (zie fig. 6.35).

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ #+	extend op ~ #	inher op ~ #		
DAA SEX					19 2 1 1D 2 1	decimal adjust A Ai:=B7, i=0 t/m 7	***0* ***0-

Fig. 6.35 De instructies Decimal Adjust A en Sign Extension

De instructie Sign EXTend B to A is een inherent addressing instructie. Zoals de omschrijving al zegt, wordt het teken van B uitgebreid in A. Dit wil zeggen dat (A):=\$00 als B7=0 en (A):=\$FF als B7=1. Hiermee wordt bereikt dat een 8-bits two's complement getal in B wordt omgezet in een 16-bits two's complement getal in D. D is immers de aaneenschakeling van A en B. In paragraaf 2.11 hebt u reeds gezien dat een positief getal naar links mag worden uitgebreid met nullen en een negatief getal mag naar links worden uitgebreid met enen. Hierbij blijven teken en grootte van het two's complement getal ongewijzigd.

Voorbeeld 6.45

LDB	#9A	(B) := \$9A = -102
SEX		(D) := \$FF9A = -102
LDB	#\$71	(B) := \$71 = +113
SEX		(D) := \$0071 = +113

De inhoud van A gaat bij de instructie SEX verloren. (B) blijft gelijk.

Zie fig. 6.35 voor de beïnvloeding van de flags.

De Load Effective Address instructies komen alleen voor met de indexed en indirect addressing modes (zie fig. 6.36). Ze zijn op te vatten als optelinstucties, omdat de offset van de indexed addressing instructie wordt opgeteld bij de inhoud van het indexregister, dat in de operand wordt aangeduid. Het resultaat wordt opgeborgen in het indexregister, dat wordt aangegeven door de opcode. Het register dat wordt aangegeven door de operand, blijft ongewijzigd, tenzij opcode en operand hetzelfde register aanduiden. De offset is een two's complement getal (zie paragraaf 4.7).

De instructie

LEAX 4,X

heeft tot gevolg dat de inhoud van X wordt verhoogd met 4 en

LEAX -1,X

doet de inhoud van X met 1 verlagen.

Het grote verschil met de instructie

LDX -1,X

is dat hier X wordt geladen met de informatie die staat op de geheugenplaatsen met de adressen $-1+(X)$ en $-1+(X)+1$. Bij de LEA-instructie wordt niet de data geladen, maar het effectieve adres van de data. De LEA-instructies zijn eigenlijk bedoeld om de pointerregisters te initialiseren. Zoals echter al aangetoond is, kunnen ze ook worden gebruikt om twee's complement getallen op te tellen. Hierbij moet u in het oog houden, dat de flags in het condition code register niet worden gewijzigd. Dit betekent dat u geen waarschuwing krijgt als er bijvoorbeeld overflow optreedt!

Een uitzondering hierop vormt de zero bit, die door de instructies LEAX en LEAY wel wordt aangepast aan het resultaat van de bewerking.

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ #+	extend op ~ #	inher op ~ #		
LEAS			32 4+ 2+			(S):=eff. adres	-----
LEAU			33 4+ 2+			(U):=eff. adres	-----
LEAX			30 4+ 2+			(X):=eff. adres	---*--
LEAY			31 4+ 2+			(Y):=eff. adres	---*--

Fig. 6.36 Overzicht van de Load Effective Address instructies

Voorbeeld 6.46

LEAX	\$58,Y	(X) := \$58 + (Y)
LEAU	[A,S]	(UH) := ((A)+(S)) en
		(UL) := ((A)+(S)+1)
LEAS	D,X	(S) := (D) + (X)
LEAY	,--U	(U):=(U)-2 en (Y):=(U)=(U)oud-2
LEAY	,X++	(Y):=(X)oud en (X):=(X)+2
LEAX	,X++	(X):=(X), immers (X) is het effectieve adres; (X) blijft dus ongewijzigd!

Bij voorbeeld 6.46 blijkt dat (X) ongewijzigd blijft bij het uitvoeren van de instructie LEAX ,X++. De instructie zegt dat het effectieve adres moet worden geladen in X. Het effectieve adres in deze instructie is (X). Dus X wordt met zijn eigen inhoud geladen. Daarna zou u mogelijk verwachten, dat (X) wordt verhoogd met 2. Dit gebeurt echter niet, omdat dit tot gevolg zou hebben dat X niet geladen wordt met het effectieve adres (vóór de increment)! De interne sequentie bij het uitvoeren van de auto-increment/decrement instructies gaat als volgt (i1 en i2 kunnen X, Y, U en S zijn):

a. Voor "LEAi1 ,i2+":

1. bereken het effectieve adres: (TMP) := (i2)
2. voer de post-increment uit: (i2) := (i2)+1
3. laad i1 met effectieve adres: (i1) := (TMP)

b. Voor "LEA i1, -i2":

1. bereken het effectieve adres: (TMP) := (i2)-1

2. voer de pre-decrement uit: (i2) := (i2)-1

3. laad i1 met effectieve adres: (i1) := (TMP)

Het uitvoeren van auto-increment/decrement instructies met 2 gaat op soortgelijke wijze.

6.5.8 Voorbeelden van rekenkundige instructies

Voorbeeld 6.47 (Kwadraten van two's complement getallen)

In een tabel staat een reeks 1-byte two's complement getallen. De reeks wordt afgesloten met \$00. De tabel begint op adres \$5000. De kwadraten moeten komen te staan vanaf adres \$6000. De reeks kwadraten dient te worden afgesloten met het getal \$FFFF. Aangezien de instructie MUL uitgaat van unsigned numbers, moeten we eerst de absolute waarde bepalen van de getallen, alvorens ze te kwadrateren.

Het bijbehorende structogram wordt gegeven in fig. 6.37.

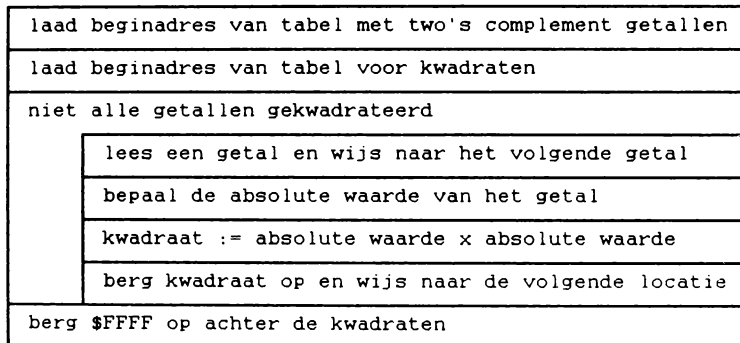


Fig. 6.37 Structogram voor het bepalen van de kwadraten van two's complement getallen

```
10          NAM      KWADRATEER 2'S COMPLEMENT GETALLEN
20 *Progr.: J.G. Rouland
30 *Revisie 0      Datum: 26-03-1986
40 *
50 *KWADR2 zet de kwadraten van een reeks two's complement
60 *getallen in het geheugen vanaf adres $6000 en sluit de
70 *reeks kwadraten af met het getal $FFFF. De reeks getallen
80 *zelf begint op $5000 en wordt afgesloten met $00.
90 *
100 *
110          ORG      $1800          beginadres objectcode
120 BUFTWO   EQU      $5000          beginadres getallenbuffer
130 BUFKWA   EQU      $6000          beginadres kwadratenbuffer
```

140	*			
150	KWADR2	LDX	#BUFTWO	initialiseer datapointer
160		LDU	#BUFKWA	initialiseer resultpointer
170	NEXT	TST	,X	spring als data=0, dus als alle
180		BEQ	KLAAR	getallen gekwadrateerd zijn
190		LDA	,X+	lees data, verhoog datapointer
200		BPL	POSIT	bepaal de
210		NEGA		absolute waarde van de data
220	POSIT	TFR	A,B	bereken het
230		MUL		kwadraat
240		STD	,U++	berg het kwadraat op en verhoog
250	*			de resultpointer
260		BRA	NEXT	
270	KLAAR	LDD	#\$FFFF	sluit de reeks
280		STD	,U	kwadraten af
290		SWI		stop
300		END	KWADR2	

Voorbeeld 6.48 (BCD-optelling)

Het probleem is om twee n-bytes BCD-getallen bij elkaar op te tellen en hun som als BCD-getal in het geheugen te zetten. Voor de som dienen n+1 bytes gereserveerd te worden.

Op adres DATA (=9000) staat het getal n. Dan volgen de twee n-bytes BCD-getallen. Daarachter moet de som opgeborgen worden (op n+1 geheugenlocaties). Fig. 6.38 toont de geheugenindeling. Als n=0 doen we niets. Het programma stopt dan.

We moeten het optellen beginnen bij de minstwaardige bytes. Daar we over voldoende pointers beschikken, gebruiken we aparte pointers voor beide getallen en voor de som.

De pointers kunnen telkens worden verlaagd met behulp van de auto-decrement addressing. Daar dit een pre-decrement is, moeten we de pointers initialiseren op een waarde die 1 hoger is dan de eigenlijke gewenste waarde. Voor de volledigheid zij hier nog opgemerkt, dat eerst de meestwaardige byte in het geheugen staat en dat de minderwaardige bytes hierna, dus op hogere adressen, volgen.

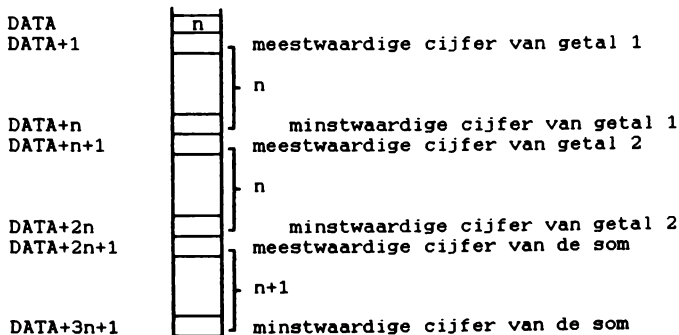


Fig. 6.38 De locaties van de BCD-getallen in het geheugen

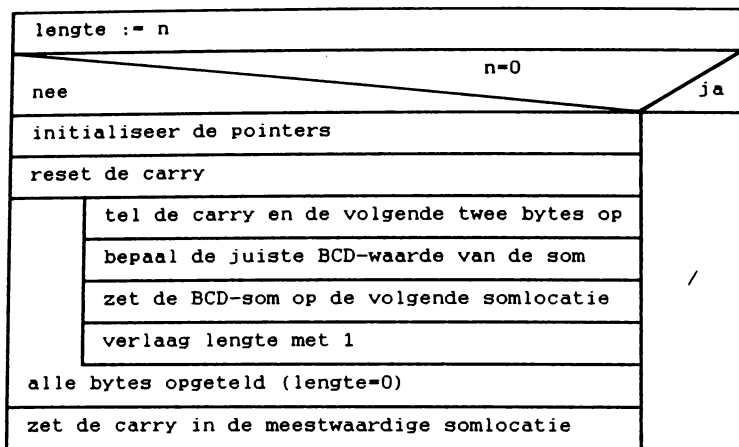


Fig. 6.39 Structogram voor optelling van twee n-bytes BCD-getallen

Fig. 6.39 laat het structogram zien. Het teken "/" geeft aan dat in de betreffende rechthoek geen actie hoeft te worden ondernomen. Door "/" te plaatsen geven we aan, dat er niets is vergeten.

```

10          NAM      SOMMEER TWEE N-BYTES BCD-GETALLEN
20 *Progr.:   J.G. Rouland
30 *Revisie 0   Datum: 26-03-1986
40 *
50 *ADDBCD telt twee n-bytes BCD-getallen bij elkaar op en
60 *plaatst de (n+1) bytes som achter de getallen in het
70 *geheugen. n is kleiner dan 128 en bevindt zich op geheugen-
80 *locatie $9000. De getallen volgen daarna. Als n=0 stopt
90 *het programma onverrichterzake.
100 *
110          ORG      $2000          beginadres objectcode
120 DATA     EQU      $9000          adres van n
130 *
140 ADDBCD    LDB      DATA          stop als
150          BEQ      STOP              n=0
160          LDX      #DATA          pointer X wijst 1 positie verder
170          LEAX     B,X              dan de minstwaardige byte van
180          LEAX     1,X              getal 1
190          LEAY     B,X              pointer Y wijst 1 positie verder
200 *          dan minstwaardige byte van getal 2
210          LEAU     B,Y              pointer U wijst 1 positie verder
220          LEAU     1,U              dan minstwaardige byte van som
230          ANDCC    #$FE          reset de carry bit
240 REKEN     LDA      ,-X              verlaag pointer X en lees een byte
250          ADCA     ,-Y              verlaag pointer Y, tel C+byte bij A

```

260	DAA		corrigeer voor BCD
270	STA	,-U	verlaag pointer U en berg som op
280	DECB		spring terug als niet alle bytes
290	BNE	REKEN	zijn opgeteld
300	ADCB	#0	berg carry op in de locatie van
310	STB	,-U	de meestwaardige byte van som
320 STOP	SWI		stop
330		END	ADBCD

Zoals u ziet wordt in regel 250 de carry van de vorige optelling ook bijgeteld. Derhalve moet (in regel 230) de carry 0 worden gemaakt, opdat het optellen van de minstwaardige bytes ook correct gebeurt. Hier is er immers nog geen inkomende carry.

Het is natuurlijk van belang dat de uitgaande carry van een optelling niet door andere instructies wordt veranderd, voordat hij als inkomende carry bij de volgende optelling is gebruikt. Dit gebeurt inderdaad niet, want de tussenliggende instructies STA, DECB, BNE en LDA (regel 270, 280, 290 en 240) laten de carry ongewijzigd. Zo ontdekt u het belang van het feit, dat vrij veel instructies de carry ongemoeid laten. De laatste uitgaande carry moet worden opgeborgen als (meest belangrijke) onderdeel van de som. Dit gebeurt in de regels 300 en 310. Merk op dat (B) gelijk aan 0 is bij regel 300.

Het is niet mogelijk de carry in accumulator A te plaatsen met behulp van de instructies

```
CLRA
ADCA  #0
```

omdat de instructie CLRA de carry reset! Merk het belang op van het raadplegen van de tabellen (appendix B) voor wat betreft de beïnvloeding van de flags!

U kunt de carry wel opbergen in A met behulp van de instructies

```
ROLA
ANDA  #%00000001
```

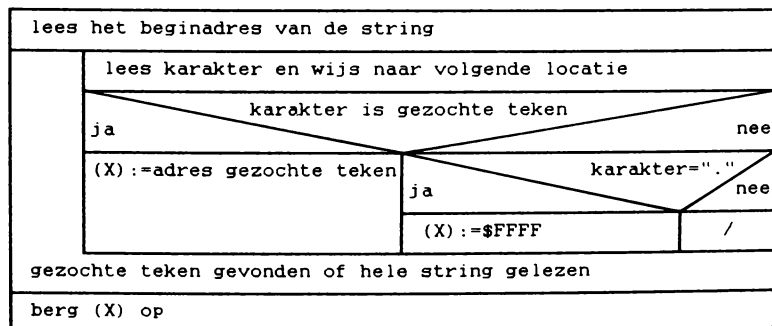


Fig. 6.40 Structogram voor zoeken naar teken in string

Voorbeeld 6.49 (Zoek character)

De opdracht luidt: Doorzoek een string (rij karakters) op het voorkomen van een bepaald teken. Zet het adres van de eerste locatie van de string waar het teken wordt aangetroffen, op de geheugenplaats met het adres CHAR+1 en CHAR+2. CHAR (= \$1000) is het adres waar het teken staat, dat in de string moet worden opgezocht. Ontbreekt dit teken in de string, dan moet als adres \$FFFF worden opgeborgen. De string begint op adres STRING (= \$1A00). De "." geeft het einde van de string aan.

Het structogram is gegeven in fig. 6.40.

Het programma ziet er als volgt uit:

```
10          NAM      ZOEK TEKEN IN STRING
20 *Progr.:  J.G. Rouland
30 *Revisie 0      Datum: 26-03-1986
40 *
50 *ZOEKTK zoekt het eerste adres van voorkomen van een
60 *bepaald teken (zoals staat op adres $1000) in een string,
70 *die begint opadres $1A00. Het gevonden adres, dan wel
80 *$FFFF wordt opgeborgen op adres $1001 en $1002. De string
90 *eindigt met de ASCII-waarde van de ".".
100 *
110          ORG      $4800          beginadres van objectcode
120 CHAR      EQU      $1000        adres van te zoeken teken
130 STRING    EQU      $1A00        beginadres van de string
140 *
150 ZOEKTK    LDX      #STRING      initialiseer stringpointer
160 NEXT      LDA      ,X+          lees karakter, verhoog pointer
170          CMPA     CHAR          spring als het gelezen karakter
180          BNE      PUNT          niet het gezochte teken is
190          LEAX     -1,X          (X) := adres van gezochte teken
200          BRA      TEST
210 PUNT      CMPA     '.'          spring als het gelezen karakter
220          BNE      TEST          ongelijk aan "." is
230          LDX      #$FFFF        (X):=$FFFF, immers einde string
240 *          bereikt en teken niet gevonden
250 TEST      CMPA     CHAR          ga naar BERGOP als
260          BEQ      BERGOP        teken is gevonden
270          CMPA     '.'          of als einde van string
280          BNE      NEXT          is bereikt
290 BERGOP    STX      CHAR+1        berg (X) op
300          SWI      stop
310          END      ZOEKTK
```

De regels 250 t/m 280 zijn overbodig voor wat betreft het resultaat van het programma. Ze kunnen worden weggelaten als in regel 200 wordt gesprongen naar BERGOP en in regel 220 naar NEXT. In dat geval wordt er echter afgeweken van het structogram, want de test in de repeat-until-structuur wordt overgeslagen. Er wordt immers overheen gesprongen. Daar we gestructureerd willen programmeren, volgen we strict het structogram, ook al kost dit extra instructies.

6.6 Spronginstructies

6.6.1 Algemeen

Spronginstructies zijn nodig om de normale doorloop van een programma te wijzigen. Zonder sprongen worden de instructies achtereenvolgens uitgevoerd, zoals ze na elkaar in het geheugen staan. Op deze wijze wordt elke instructie slechts één keer uitgevoerd. Om instructies te kunnen herhalen, hebben we een spronginstructie nodig. In paragraaf 6.3.6 is dit al uitgebreid behandeld. Hier hebt u gezien hoe de spronginstructies van essentieel belang zijn bij het realiseren van een keuzestructuur, een do-while-structuur en een repeat-until-structuur. U hebt hierbij tevens gezien, dat sommige sprongen altijd moeten worden uitgevoerd en andere alleen onder bepaalde voorwaarden. Er bestaan dan ook voorwaardelijke en onvoorwaardelijke spronginstructies.

6.6.2 Onvoorwaardelijke spronginstructies

De MC6809 kent de volgende onvoorwaardelijke sprongen:

BRA	BRN	BSR	SWI
LBRA	LBRN	LBSR	SWI2
JMP	NOP	JSR	SWI3
		RTS	RTI

De instructies BRanch Always, Long BRanch Always en JuMP veroorzaken altijd een sprong. Na de sprong wordt het programma op de normale wijze voortgezet vanaf de instructie waar naartoe is gesprongen.

De instructies BRanch Never, Long BRanch Never en No Operation veroorzaken nooit een sprong. We kunnen echter ook zeggen dat deze instructies ervoor zorgen, dat er enkele geheugenplaatsen in het programmageheugen worden overgeslagen, namelijk die plaatsen, waarop deze instructies zelf staan. Er wordt in feite naar de eerstvolgende instructie gesprongen.

De instructie NOP kost 1, BRN 2 en LBRN 4 bytes geheugenruimte. Na het ophalen van deze instructies wordt het programma weer gewoon voortgezet met de instructie, die als volgende in het geheugen staat.

U kunt NOP, BRN en LBRN toevoegen om een nauwkeurige vertraging te maken. NOP kost 2, BRN 3 en LBRN 5 klokcycli (zie ook fig. 6.41).

Met behulp van de instructies Branch to SubRoutine, Long Branch to SubRoutine en Jump to SubRoutine wordt uit het programma weggesprongen naar een geheugenlocatie, waar een zogenaamde

subroutine staat. Een subroutine is een groep instructies, die wordt afgesloten met de instructie ReTurn from Subroutine. Deze RTS zorgt ervoor dat het programma, na het uitvoeren van de subroutine, weer wordt vervolgd met de instructie die na BSR, LBSR of JSR komt. Hierdoor kan een subroutine vanuit diverse plaatsen in het programma worden aangeroepen. Na executie van de subroutine wordt het programma voortgezet met de instructie, volgend op de aanroep, ook wel de subroutine call genoemd. Bij de instructies BRA, LBRA en JMP wordt ook weggesprongen, echter er wordt niet meer automatisch teruggekeerd naar de plaats waar het programma verlaten werd. Hier wordt de flow van het programma dus definitief gewijzigd. Bij een subroutine call wordt het programma, na het uitvoeren van de subroutine, weer gewoon vervolgd. Er wordt als het ware een groep instructies tussengevoegd.

Subroutines zullen uitgebreid worden behandeld in hoofdstuk 8.

instr	addressing modes					beschrijving	(CC) bit 53210 HNZVC
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ #+	extend op ~ #	inher op ~ #		
JMP		0E 3 2	6E 3+ 2+	7E 4 3		jump	-----
JSR		9D 7 2	AD 7+ 2+	BD 8 3		jump to subr.	-----
NOP					12 2 1	no operation	-----
RTI					3B h 1	return from int.	iiii
RTS					39 5 1	return from subr	-----
SWI					3F 19 1	softw. int. 1	1
SWI2					10 20 2	softw. int. 2	-----
					3F		-----
SWI3					11 20 2	softw. int. 3	-----
					3F		-----

- Opm.: h. 6 klokcycli na FI, 15 klokcycli na andere interrupt.
 i. (CC) wordt van de stack gehaald.
 1. SWI set de condition code bits I en F.

instr	addr. mode	spring als	(CC) bit 53210 HNZVC
	relat op ~ #		
BRA	20 3 2	altijd	-----
LBRA	16 5 3	altijd	-----
BRN	21 3 2	nooit	-----
LBRN	10 5 4	nooit	-----
	21		-----
BSR	8D 7 2	altijd naar subroutine	-----
LBSR	17 9 3	altijd naar subroutine	-----
	8D		-----

Fig. 6.41 Overzicht van de onvoorwaardelijke spronginstructies

De Software Interrupt instructies SWI, SWI2 en SWI3 zijn ook sprong-instructies. Hier wordt het programma softwarematig onderbroken om een zogenaamde software interrupt service routine uit te voeren. De interrupt service routine eindigt met RTI (ReTurn from Interrupt service routine). Hierna wordt het programma weer voortgezet. Interrupts worden echter pas behandeld in hoofdstuk 9, zodat er hier niet verder op wordt ingegaan.

In fig. 6.41 vindt u de addressing modes van de onvoorwaardelijke spronginstructies. Merk op dat de flags ongewijzigd blijven.

De instructies NOP, RTS, SWI, SWI2, SWI3 en RTI zijn inherent addressing instructies.

De branch-instructies zijn alle relative addressing instructies, terwijl de jump-instructies kunnen voorkomen als direct, extended, indexed en indirect addressing instructies.

De short branch en de long branch verschillen alleen in de lengte van het (two's complement) relatieve adres. De operand van de short branch is één byte groot, zodat hier maximaal 128 plaatsen terug of 127 plaatsen vooruit kunnen worden gesprongen. Bij de long branch met een 2-bytes relatieve adres kunnen 32.768 plaatsen terug of 32.767 plaatsen vooruit worden gesprongen. Dit betekent dat met een long branch het gehele 64k-geheugen kan worden bereikt.

U hoeft het relatieve adres niet zelf uit te rekenen. Dat doet de assembler voor u, zoals u in hoofdstuk 5 hebt gezien. U kunt volstaan met te noteren

BRA VERDER ga naar VERDER

De assembler rekent uit hoe groot het relatieve adres moet zijn, opdat het programma wordt voortgezet op de geheugenlocatie met het adres VERDER. Het effectieve adres wordt aangegeven door VERDER bij de gewenste instructie als label te gebruiken. De assembler vertaalt VERDER niet door zijn absolute fysische geheugenadres, maar door een relatieve adres, dus door de afstand tot de huidige positie, de huidige inhoud van de program counter.

De instructie

JMP VERDER ga naar VERDER

heeft hetzelfde effect. Nu vertaalt de assembler het symbolische adres VERDER in de bijbehorende fysische waarde. Wordt het programma in een ander geheugengebied geladen dan de bedoeling van de programmeur is, dan veranderen alle adressen, dus ook het adres van de label VERDER. Nu wordt er naar een verkeerd adres gesprongen, immers de operand verandert niet, behalve bij gebruik van een zogenaamde relocatable loader. Deze verhoogt alle operandadressen tijdens het laden van het programma met de juiste offset. Hiertoe moet het programma relocatable geassembleerd zijn met behulp van een zogenaamde

relocatable assembler. De relocatable assembler maakt een lijst van symbolen met hun relatieve adressen. Bij deze relatieve adressen wordt door de relocatable loader een offset (de zogenaamde relocation constante) opgeteld, zodat de absolute adressen kunnen worden ingevuld.

Bij de BRA-instructie wordt steeds naar het juiste adres gesprongen, ook al wordt het programma in een ander geheugegebied geladen. Dit komt doordat de afstand tussen de spronginstructie en de instructie waar naartoe wordt gesprongen, gelijk blijft. Dit is onafhankelijk van de plaats van het programma in het geheugen. Een relocatable loader hebben we dus niet nodig als we relatieve spronginstructies gebruiken.

Voorbeeld 6.50 (Het verschil tussen BRA of LBRA en JMP)

Veronderstel dat (ten gevolge van ORG \$2000) de label TERUG de waarde \$200F, BRANCH de waarde \$2031, VERDER de waarde \$205B en JUMP de waarde \$20A1 heeft. Het sourceprogramma en het objectprogramma zijn gegeven in fig. 6.42.

assemblyprogramma			machinetaalprogramma		
	ORG	\$2000	adressen		
	--	--		--	--
	--	--		--	--
TERUG	--	--	200F	--	--
	--	--		--	--
	--	--		--	--
BRANCH	BRA	VERDER	2031	20	28
	--	--	2033	--	--
	--	--		--	--
	--	--		--	--
VERDER	--	--	205B	--	--
	--	--		--	--
	--	--		--	--
JUMP	JMP	TERUG	20A1	7E	200F
	--	--	20A4	--	--

Fig. 6.42 Voorbeeld van een positie-onafhankelijke sprong (BRA) en een positie-afhankelijke sprong (JMP)

U kunt duidelijk zien, dat de label VERDER wordt vertaald door het relatieve adres \$28 (= \$205B - \$2033). Bij het uitvoeren van het programma wordt het effectieve (fysieke) adres berekend door het relatieve adres op te tellen bij de inhoud van de program counter. Dit levert $0028 + 2033 = 205B$. Realiseer u hierbij dat de program counter al naar de volgende instructie wijst, als de BRA-instructie is opgehaald door de microprocessor.

De label TERUG wordt vertaald in het absolute fysieke geheugenadres \$200F.

Zou nu het programma, bijvoorbeeld door het operating system, worden geladen vanaf adres \$3000, dan worden alle adressen \$1000 hoger.

Bij het uitvoeren van het programma wordt het effectieve adres van de BRA-instructie gelijk aan $\$0028 + \$3033 = \$305B$. Dit is inderdaad het gewenste adres.

Bij de JMP-instructie wordt er gesprongen naar $\$200F$, terwijl de bedoelde instructie nu niet meer op $\$200F$, maar op $\$300F$ staat. Het programma loopt dus verkeerd af. We zeggen dat het programma positie-afhankelijk is. Willen we een positie-onafhankelijk programma schrijven, dan mogen we alleen (relative) branch-instructies gebruiken en geen direct of extended jump-instructies.

6.6.3 Voorwaardelijke spronginstructies

De voorwaardelijke spronginstructies zijn alle relative addressing instructies. De sprong wordt alleen uitgevoerd als aan bepaalde voorwaarden is voldaan. Het zijn dus in feite gecombineerde test- en spronginstructies. Ze testen een of meerdere bits van het condition code register.

De voorwaardelijke sprongen kunnen worden verdeeld in twee groepen:

1. sprongen die tests uitvoeren met betrekking tot unsigned numbers en
2. sprongen die tests uitvoeren met betrekking tot two's complement numbers.

Bij de eerste categorie wordt getest op de flags Z en C, bij de tweede categorie op de flags Z, N en V. De Z-bit is voor elk soort getal van toepassing. De carry bit heeft alleen betrekking op unsigned getallen, terwijl N en V alleen te maken hebben met getallen met een tekenbit, dus met two's complement getallen.

Sprongen met betrekking tot unsigned numbers

De MC6809 kent de volgende spronginstructies die betrekking hebben op unsigned numbers (zie ook fig. 6.43):

BCC	LBCC	(Long) Branch if Carry Clear
BCS	LBCS	(Long) Branch if Carry Set
BLO	LBLO	(Long) Branch Lower
BLS	LBLS	(Long) Branch Lower or Same
BEQ	LBEQ	(Long) Branch if Equal
BHS	LBHS	(Long) Branch if Higher or Same
BHI	LBHI	(Long) Branch if Higher
BNE	LBNE	(Long) Branch if Not Equal

Met de instructies branch if carry clear/set kan worden getest of de carry 0 of 1 is en als functie daarvan wordt er gesprongen. Deze instructie kan bijvoorbeeld worden uitgevoerd na een add-instructie om na te gaan of de som van twee unsigned numbers in de accumulator past. Na een schuifinstructie kan hiermee worden nagegaan of de uitgeschoven bit 0 of 1 is.

De overige instructies zeggen iets over de verhouding van twee unsigned getallen. Deze instructies worden toegepast na een aftrek- of na een vergelijkinstructie. Laten we, aan de hand van voorbeeld 6.51, eens nagaan wat er in de diverse gevallen gebeurt met de significante flags Z en C.

Voorbeeld 6.51

G=\$2A (= 42)	K=\$51 (=81)	Q=\$B7 (=183)
H=\$C7 (=199)	L=\$51 (=81)	R=\$73 (=115)
\$63 (gaat niet)	\$00 (= 0)	\$44 (= 68)
Z:=0 en C:=1	Z:=1 en C:=0	Z:=0 en C:=0

Als we een getal G vergelijken met een groter getal H door het verschil G-H te berekenen, dan zullen we ontdekken, dat het aftrekken niet gaat zonder te lenen. Dit betekent dat de borrow bit C wordt geset. Met andere woorden uit C=1 kunnen we concluderen dat G<H. Dit betekent dat de instructies BLO, LBLO en de complementaire instructies BHS en LBHS de C-bit testen. Bij de eerste twee instructies wordt gesprongen als C=1, bij de laatste twee als C=0.

We concluderen hieruit dat de instructies BLO en BCS, LBLO en LBCC, BHS en BCC, LBHS en LBCC, paarsgewijze identiek zijn. U kunt dit controleren door de opcodes van deze instructies met elkaar te vergelijken (zie fig. 6.43). Dat er toch verschillende mnemonics zijn voor dezelfde instructie, stelt ons in staat om beter leesbare programma's te schrijven. Bovendien wordt ons hierdoor enig denkwerk bespaard.

Voorbeeld 6.52

LDA	G	spring als
CMPA	H	G groter of gelijk is aan H
BHS		(ofwel als C=0)

Alleen in het geval dat we twee gelijke getallen K en L met elkaar vergelijken, zal het verschil 0 zijn, zodat Z=1 wordt. Hieruit kunnen we concluderen dat K=L. Bij de instructies BEQ en LBEQ wordt de sprong dus uitgevoerd als Z=1. Bij BNE en LBNE wordt gesprongen als Z=0.

C en Z kunnen na een vergelijk- of aftrekinstructie nooit beide 1 zijn. Immers als Z=1 wordt, zijn de getallen gelijk aan elkaar en hoeft er niet te worden geleend.

Bij het vergelijken (dan wel aftrekken) van een getal Q met een kleiner getal R, worden zowel de carry bit als de zero bit gereset. Dit betekent dat bij de instructies BHI en LBHI alleen wordt gesprongen als zowel Z als C gelijk aan 0 zijn, met andere woorden als geldt dat (C OR Z)=0. Bij de complementaire instructies BLS en LBS wordt derhalve gesprongen als (C OR Z)=1.

Het zal u zonder twijfel duidelijk zijn, dat aan de voorwaarde (C OR Z)=1 wordt voldaan, indien geldt dat C=1 of Z=1 of C=Z=1.

Aan de voorwaarde (C OR Z)=0 wordt alleen dan voldaan als C en Z beide 0 zijn, dus als zowel geldt C=0 als Z=0. Merk op dat dit iets anders is dan C=0 of Z=0! Dit laatste wordt in logische vorm als volgt weergegeven: (C AND Z)=0. Immers aan deze voorwaarde is voldaan als C of Z gelijk is aan 0.

instr	addr. mode	spring als	(CC) bit
	relat op ~ #		53210 HNZVC
BCC	24 3 2	C=0	-----
LBCC	10 n 4	C=0	-----
	24		-----
BCS	25 3 2	C=1	-----
LBCS	10 n 4	C=1	-----
	25		-----
BLO	25 3 2	C=1; (R)<(M)	-----
LBLO	10 n 4	C=1; (R)<(M)	-----
	25		-----
BLS	23 3 2	(C OR Z)=1; (R)<=(M)	-----
LBLS	10 n 4	(C OR Z)=1; (R)<=(M)	-----
	23		-----
BEQ	27 3 2	Z=1	-----
LBEQ	10 n 4	Z=1	-----
	27		-----
BHS	24 3 2	C=0; (R)>=(M)	-----
LBHS	10 n 4	C=0; (R)>=(M)	-----
	24		-----
BHI	22 3 2	(C OR Z)=0; (R)>(M)	-----
LBHI	10 n 4	(C OR Z)=0; (R)>(M)	-----
	22		-----
BNE	26 3 2	Z=0	-----
LBNE	10 n 4	Z=0	-----
	26		-----

Opm.: n. 5 cycli als er niet wordt gesprongen, 6 als er wel wordt gesprongen.

Fig. 6.43 Overzicht van de voorwaardelijke spronginstructies die betrekking hebben op unsigned numbers

Sprongen met betrekking tot two's complement getallen

De MC6809 kent de volgende spronginstructies die betrekking hebben op two's complement numbers (zie ook fig. 6.44):

BMI	LBMI	(Long) Branch if Minus
BPL	LBPL	(Long) Branch if Plus
BVC	LBVC	(Long) Branch if overflow Clear
BVS	LBVS	(Long) Branch if overflow Set
BLT	LBLT	(Long) Branch if Less Than
BLE	LBLT	(Long) Branch if Less or Equal
BEQ	LBEQ	(Long) Branch if Equal
BGE	LBGE	(Long) Branch if Greater or Equal
BGT	LBGT	(Long) Branch if Greater Than
BNE	LBNE	(Long) Branch if Not Equal

Met behulp van de instructies BMI, LBMI, BPL en LBPL kan worden nagegaan of een getal negatief of positief is. Bij de eerste twee instructies wordt de sprong uitgevoerd, als de negatieve bit gelijk aan 1 is (ofwel als het getal negatief is). Bij de laatste twee instructies wordt er gesprongen als N=0, dus als het getal positief is.

De negatieve bit is in feite een kopie van de meestwaardige bit (de tekenbit) van het resultaat van de vorige bewerking.

Derhalve kunt u de betreffende instructies ook gebruiken om de meestwaardige bit te testen.

instr	addr. mode	spring als	(CC) bit
	relat op ~ #		53210 HNZVC
BMI	2B 3 2	N=1	-----
LBMI	10 n 4	N=1	-----
	2B		-----
BPL	2A 3 2	N=0	-----
LBPL	10 n 4	N=0	-----
	2A		-----
BLT	2D 3 2	(N EOR V)=1; (R)<(M)	-----
LBLT	10 n 4	(N EOR V)=1; (R)<(M)	-----
	2D		-----
BLE	2F 3 2	(Z OR (N EOR V))=1; (R)<=(M)	-----
LBLE	10 n 4	(Z OR (N EOR V))=1; (R)<=(M)	-----
	2F		-----
BEQ	27 3 2	Z=1	-----
LBEQ	10 n 4	Z=1	-----
	27		-----
BGE	2C 3 2	(N EOR V)=0; (R)>=(M)	-----
LBGE	10 n 4	(N EOR V)=0; (R)>=(M)	-----
	2C		-----
BGT	2E 3 2	(Z OR (N EOR V))=0; (R)>(M)	-----
LBGT	10 n 4	(Z OR (N EOR V))=0; (R)>(M)	-----
	2E		-----
BNE	26 3 2	Z=0	-----
LBNE	10 n 4	Z=0	-----
	26		-----

Opm.: n. 5 cycli als er niet wordt gesprongen, 6 als er wel wordt gesprongen.

Fig. 6.44 Overzicht van de voorwaardelijke spronginstructies die betrekking hebben op two's complement numbers

Voorbeeld 6.53

Een rookdetector geeft een signaal ROOK af, dat via bit 7 van register SIGNAL kan worden binnengehaald in de microprocessor. Als ROOK=1 moeten de sprinklers in werking worden gesteld en moet de brandweer worden gewaarschuwd. Het signaal ROOK kan als volgt worden getest:

```

LDA    SIGNAL    spring als
BMI    BRAND    ROOK=1

```

Dit gaat eenvoudiger dan

LDA	SIGNAL
BITA	*\$10000000
BNE	BRAND

hetgeen ook correct is, maar meer geheugenruimte en meer executietijd kost.

De instructies BVC, LBVC, BVS en LBVS stellen u in staat om de overflow bit te testen. U kunt na een optel- of aftrekinstructie van twee two's complement getallen nagaan of er overflow is opgetreden, met andere woorden of de uitkomst in het register past. Is dit niet zo, dan moeten er maatregelen worden getroffen. Er moet meer ruimte geschapen worden om de uitkomst op te bergen, nadat het resultaat gecorrigeerd is. De sprongen BVC en LBVC worden uitgevoerd als V=0. Bij BVS en LBVS wordt er gesprongen als V=1.

De overige instructies dienen om na te gaan of een two's complement getal T kleiner, gelijk of groter is dan een two's complement getal W. Deze spronginstructies worden gebruikt na een aftrek- of vergelijkinstructie. We zullen nu aan de hand van voorbeeld 6.54 nagaan wat er gebeurt met de relevante flags Z, N en V.

Als de twee two's complement getallen gelijk zijn, geldt altijd Z=1 en N=V=0. Zijn de getallen verschillend, dan zal Z altijd gelijk aan 0 zijn. De Z-bit wordt getest door de instructies BEQ, LBEQ, BNE en LBNE. Dit is reeds besproken bij de unsigned spronginstructies.

Voorbeeld 6.54 (Situaties waarvoor geldt: T<W)

G=\$15	(=+21)	K=\$EA	(=-22)	M=\$B8	(=-72)	Q=\$C2	(=-62)
H=\$23	(=+35)	L=\$44	(=+68)	P=\$53	(=+83)	R=\$F0	(=-16)
\$F2	(=-14)	\$A6	(=-90)	\$65	(=+101)	\$D2	(=-46)
Z=0, N=1, V=0		Z=0, N=1, V=0		Z=0, N=0, V=1		Z=0, N=1, V=0	

We kunnen nu concluderen dat T<W, indien Z=0, terwijl N en V verschillend zijn. Nu volgt uit N=1 dat Z=0, want een negatief getal is ongelijk aan 0, omdat tenminste de tekenbit gelijk is aan 1. Uit V=1 volgt eveneens dat Z=0, want er is nooit sprake van overflow als de uitkomst 0 is. Het getal 0 past immers in een 8-bits register. Voldoende voorwaarde voor T<W is derhalve (N EOR V)=1. Aan deze voorwaarde is alleen voldaan als N=0 en V=1 of als N=1 en V=0, met andere woorden, als N en V verschillend zijn.

De complementaire situatie T is groter of gelijk aan W treedt dus op als (N EOR V)=0. Z mag hierbij 0 (T>W) of 1 (T=W) zijn. Hieruit vinden we tevens de voorwaarde voor T>W. Dan moet gelden (N EOR V)=0, terwijl tevens moet gelden Z=0. Dit betekent dus dat voldaan moet zijn aan de conditie (Z OR (N EOR V))=0.

Uit dit laatste leiden we af dat de complementaire situatie T kleiner of gelijk aan W geldt bij $(Z \text{ OR } (N \text{ EOR } V))=1$. Hiermee hebben we alle mogelijke combinaties gehad.

We zullen aan de hand van voorbeeld 6.55 laten zien, dat voor $(Z \text{ OR } (N \text{ EOR } V))=0$ inderdaad geldt dat $T>W$.

Voorbeeld 6.55 (Situaties waarbij $T>W$)

G=\$7F (+=127) K=\$5E (+=94) M=\$2C (+=44) Q=\$AA (--86)
H=\$39 _ (+=57) L=\$EB _ (--21) P=\$9A _ (--102) R=\$9F _ (--97)
\$46 (+=70) \$73 (+=115) \$92 (--110) \$0B (+=11)
Z=0, N=0, V=0 Z=0, N=0, V=0 Z=0, N=1, V=1 Z=0, N=0, V=0

Alle spronginstructies laten de flags H, N, Z, V en C ongewijzigd (zie fig. 6.43 en 6.44).

We zullen nog enkele belangrijke aanbevelingen doen omtrent het gebruik van test- en spronginstructies. We zullen deze aanbevelingen nader toelichten aan de hand van enkele voorbeelden.

Kies steeds de meest voor de hand liggende instructie, indien er meer mogelijkheden zijn. Zorg zodoende dat uw programma goed leesbaar is. Ga geen onnodige kronkelwegen bewandelen bij het testen van gegevens.

Kies telkens de juiste spronginstructie door u goed te realiseren wat de betekenis van die instructie is.

Zorg ervoor dat de test goed is voorbereid, dat wil zeggen dat de spronginstructie door de juiste instructie wordt voorafgegaan.

Voorbeeld 6.56 (Test of een getal positief is)

Om na te gaan of een getal positief is, ligt het voor de hand de negative bit te testen:

BPL POSIT spring als getal positief is

en niet na te gaan of het getal groter of gelijk 0 is:

TSTA

BGE POSIT

Voorbeeld 6.57 (Test of een getal na rotatie negatief is)

Moet u nagaan of het resultaat van een rotatie-instructie negatief is, laat u er dan niet toe verleiden de instructie BLT toe te passen in plaats van BMI. BLT is bedoeld om na te gaan of G<H bij het vergelijken van G en H (dus bij het aftrekken van G en H). Zoals u hebt gezien, wordt er bij BLT gesprongen, als voldaan is aan de voorwaarde $(N \text{ EOR } V)=1$. Stel $(A)=$F2 (dus negatief), terwijl $C=V=1$ (ten gevolge van eerdere instructies). Dan geldt na de instructie RORA dat $(A)=$F9 (dus negatief) en $C=0$, $N=1$ en $V=1$ (V verandert immers niet bij RORA). Nu is niet aan de voorwaarde $(N \text{ EOR } V)=1$ voldaan, zodat er bij de instructie BLT NEGAT niet wordt gesprongen, terwijl dit wel de bedoeling is, want (A) is wel degelijk negatief. Bij$$

RORA

BMI NEGAT

wordt de sprong in dit voorbeeld wel uitgevoerd, immers N=1.

Voorbeeld 6.58 (Test of een getal kleiner of gelijk aan 0 is)

Willen we kijken of een getal kleiner of gelijk is aan 0, dan kan dit met behulp van de instructie BLE, mits er een vergelijkinstructie (CMP of TST) aan vooraf gaat. Met TST wordt de geadresseerde data vergeleken met 0:

```
TST    $3A01    spring als  
BLE    KLGE     ($3A01)≤0
```

BMI kan in dit geval niet worden toegepast, omdat hiermee alleen getest wordt of het getal negatief is, dat wil zeggen kleiner dan 0.

0 wordt opgevat als een positief getal, omdat de tekenbit gelijk aan 0 is! Dit betekent dat de instructie BPL het groter of gelijk zijn test en dus niet de complementaire instructie is van BGE (kleiner of gelijk).

Bovendien zijn de instructies BMI en BPL niet gelijk aan BLT, respectievelijk BGE. De eerste twee instructies testen alleen de tekenbit (N), terwijl de laatste twee de functie (N EOR V) testen. Verwissel deze instructies derhalve niet en gebruik BMI en BPL uitsluitend om de tekenbit van een getal te testen en BLT en BGE om na te gaan of het ene getal kleiner dan, respectievelijk groter of gelijk aan het andere getal is.

Voorbeeld 6.59 (Het verschil tussen BLO en BLT)

Let er op dat u de instructies BLO (Branch if Lower) en BLT (Branch if Less Than) niet verwisselt. De eerste heeft betrekking op unsigned getallen, de tweede op two's complement getallen. Nemen we als voorbeeld de unsigned numbers G=\$9C (=156) en H=\$14 (=20), dan geldt na de instructies

```
LDA    G  
CMPA   H
```

het volgende:

G=\$9C

H=\$14 _

\$88, zodat Z=0, N=1, V=0 en C=0

Dit betekent dat de sprong bij BLO niet wordt uitgevoerd, immers aan de voorwaarde C=1 is niet voldaan. Inderdaad is G niet kleiner dan H.

De sprong bij BLT wordt daarentegen wel uitgevoerd, omdat aan de voorwaarde (N EOR V)=1 voldaan is. Bij de keuze van deze laatste spronginstructie zal het programma foutief aflopen. Verwissel dus niet de unsigned en de two's complement branch-instructies!

Fig. 6.45 geeft een overzicht van de test- en spronginstructies met de bijbehorende voorwaarden waaraan voldaan moet zijn, opdat de betreffende sprong wordt uitgevoerd. Merk op dat het testen van de carry bit wijst op unsigned numbers, terwijl de overflow bit aangeeft dat het om two's complement numbers gaat.

unsigned		beschrijving	two's complement	
instr.	voorwaarde		instr.	voorwaarde
(L)BLO	C=1	kleiner dan	(L)BLT	(N EOR V)=1
(L)BLS	(C OR Z)=1	kleiner of gelijk	(L)BLE	(Z OR (N EOR V))=1
(L)BEQ	Z=1	gelijk aan	(L)BEQ	Z=1
(L)BHS	C=0	groter of gelijk	(L)BGE	(N EOR V)=0
(L)BHI	(C OR Z)=0	groter dan	(L)BGT	(Z OR (N EOR V))=0
(L)BNE	Z=0	ongelijk aan	(L)BNE	Z=0

Fig. 6.45 Overzicht van de unsigned en two's complement test- en spronginstructies

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ #+	extend op ~ #	inher op ~ #		
CWAI SYNC	3C c 2				13 m 1	(CC):=(CC) AND (M) synchr. met int.	53210 HNZVC bbbb -----

Opm.: b. Flags worden gereset of veranderen niet, afhankelijk van de waarde van het masker.
c. Tenminste 20 klokcycli.
m. Tenminste 4 klokcycli.

Fig. 6.46 De instructies CWAI en SYNC

De instructies CWAI en SYNC hebben te maken met hardware interrupts en worden in hoofdstuk 9 behandeld.

6.6.4 Voorbeelden met spronginstructies

Voorbeeld 6.60 (Som van n two's complement getallen)

We willen n 1-byte two's complement getallen optellen. Het aantal n staat op de geheugenplaatsen \$1000 en \$1001, gevolgd door de getallen. De som moet komen te staan op de adressen \$B000 t/m \$B002, te beginnen met de meestwaardige byte. Bij het optellen van maximaal 65.000 bytes zal de uitkomst hoogstens 3 bytes omvatten.

In dit geval, waarbij de ruimte voor de som slechts 3 bytes groot is, kan de optelling het eenvoudigst geschieden door elk getal eerst tot 3 bytes uit te breiden. We beginnen met SOM=0 en tellen er vervolgens steeds een 3-bytes two's complement getal bij, totdat we alle getallen hebben gehad.

Het optellen van deze bytes levert geen winst op aan instructies, indien we dit in een programmaalgoritme doen. Derhalve zetten we de drie optellingen met aparte instructies onder elkaar, in plaats van de optelling drie keer te herhalen.

Fig. 6.47 toont het structogram van de probleemoplossing.

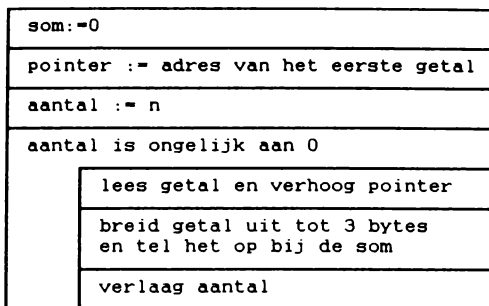


Fig. 6.47 Structogram voor optelprogramma voor two's complement bytes

Het uitbreiden van een getal tot meerdere (in ons geval 3) bytes kan op eenvoudige wijze gebeuren. Is het getal positief, dan moeten er (2) bytes met nullen worden voorgezet. Is het getal negatief, dan moeten er enen worden voorgezet. Hiertoe laden we het getal in B. De tekenuitbreiding komt dan in A (met behulp van de instructie SEX). Tellen we (B) op bij de minstwaardige byte van de som en vermeerderen we daarna de meerwaardige bytes van de som met (A)+C, dan verkrijgen we de juiste uitkomst.

Getallenvoorbeeld: \$0685 + \$C2

som = %0000 0110 1000 0101 = +1669

getal = % 1100 0010 = - 62

Zet getal in B en breid het teken uit in A:

(B):=%1100 0010 en (A):=%1111 1111

Tel (B) op bij somL:

somL = %1000 0101

(B) = %1100 0010 +

somL:= %0100 0111 en C:=1

Bereken vervolgens somH := somH + (A) + C:

somH = %0000 0110

(A) = %1111 1111

C = % 1 +

somH:= %0000 0110 en (V=0)

Er geldt nu: som = %0000 0110 0100 0111 = +1607 (dit klopt).

Het programma ziet er nu als volgt uit:

```

10          NAM      ADD 2'S COMPL. NUMBERS
20 *Progr.:   J.G. Rouland
30 *Revisie 0   Datum: 29-03-1986
40 *
50 *ADD2'S bepaalt de som van n 1-byte two's complement
60 *getallen. Deze getallen bevinden zich vanaf locatie $1002.
```

```

70 *Op $1000 en $1001 staat n. De som komt te staan op $B000
80 *t/m $B002, te beginnen met de meestwaardige byte.
90 *
100 *
110          ORG      $A000          beginadres objectcode
120 N      EQU      $1000          locatie van het aantal getallen
130 DATA  EQU      $1002          beginadres van de getallen
140 SOM    EQU      $B000          meestwaardige byte van de som
150 *
160 ADD2'S CLR      SOM            maak
170          CLR      SOM+1        de som
180          CLR      SOM+2        gelijk aan 0
190          LDX      #DATA        wijs naar het eerste getal
200          LDY      N            lees aantal op te tellen getallen
210 TEST    CMPY     #0
220          BEQ      EINDE        spring als aantal = 0
230          LDB      ,X+          laad getal en wijs naar volgende
240          SEX                      breid het getal uit met 1 byte
250          ADDB     SOM+2        tel low-byte van getal op bij
260          STB      SOM+2        minstwaardige byte van som
270          TFR      A,B          breid getal met nog een byte uit
280          SEX                      tot 3 bytes
290          ADCB     SOM+1        verhoog de 2 meestwaardige
300          ADCA     SOM          bytes van som met de twee
310          STD      SOM          meestwaardige bytes van getal
320          LEAY     -1,Y         verlaag aantal nog op te tellen
330          BRA      TEST        getallen met 1 en ga naar TEST
340 EINDE   SWI                      stop
350          END      ADD2'S

```

De meestwaardige byte van de som staat op het adres SOM = \$B000. De minstwaardige byte van som staat derhalve op SOM+2 = \$B002.

Bij regel 250 mag niet de instructie ADDD SOM+1 worden gebruikt, teneinde 2 bytes tegelijk op te tellen. Het gevaar bestaat hierbij dat het teken van het getal verloren gaat. Dit teken is nog nodig ten behoeve van de uitbreiding van het getal tot 3 bytes.

De instructie CMPY #0 (regel 210) is eigenlijk overbodig, omdat de test- en spronginstructie BEQ (regel 220) wordt voorafgegaan door LDY (regel 200) of door LEAY -1,Y (regel 320), zodat de Z-bit al betrekking heeft op de inhoud van Y, ook zonder CMPY #0. Ten behoeve van de leesbaarheid is de compare-instructie toch gebruikt. We houden ons op deze wijze strict aan het structogram. De testinstructie (aan het begin of aan het einde van de programmalus) bestaat in feite uit een testinstructie en een spronginstructie.

De regels 290 en 300 kunnen evenmin worden vervangen door ADDD SOM, omdat de carry van de vorige optelling moet worden bijgeteld. De instructie ADCD zou wel mogen, echter deze bestaat niet bij de MC6809.

Voorbeeld 6.61 (Correctie van een som bij overflow)

Overflow kan bij het optellen van two's complement getallen in slechts twee gevallen ontstaan: bij het optellen van twee positieve getallen (de uitkomst wordt dan negatief) en bij het optellen van twee negatieve getallen (de som wordt dan positief). Zie de volgende twee voorbeelden en de bijbehorende correctie.

%0100 1000 (=+ 72) correctie: %0 0100 1000
%0110 0101 (=+101) %0 0110 0101 +
%1010 1101 (negatief) %0 1010 1101 (=-173)
Bij V=1 en N=1 moeten we blijkbaar een 0 voor de som plaatsen.

%1010 0010 (=- 94) correctie: %1 1010 0010
%1100 0101 (=- 59) %1 1100 0101 +
%0110 0111 (positief) %1 0110 0111 (=-153)
Bij V=1 en N=0 moeten we kennelijk een 1 voor de som plaatsen.

Tijdens het optellen bestaat de correctie bij overflow kennelijk in het plaatsen van de inverse negative bit voor de som. Staat de som in register B, dan kan dit eenvoudig geschieden met behulp van de instructie SEX, waardoor het teken van B in A wordt uitgebreid. Alle bits van A zijn nu gelijk aan N, want N is gelijk aan de tekenbit van B, dus N=B7. We moeten echter de inverse van N in A zetten. Dit gaat met behulp van de instructie COMA (na de instructie SEX).

De som beslaat nu één byte meer dan voorheen. Indien de lengte van de som in de geheugenlocatie LENGTE staat, moeten we deze met 1 verhogen (met INC LENGTE, indien de lengte één byte omvat).

We veronderstellen dat pointer U naar de laatst weggeschreven sombyte wijst.

Het programmadeel voor de correctie van de two's complement som bij overflow ziet er nu als volgt uit:

100	ADCB	,X+	tel meestwaardige bytes en C op
110	BVC	ZETWEG	spring als er geen overflow is
120	SEX		voeg de juiste sombyte toe ter
130	COMA		correctie van de overflow
140	STA	-2,U	berg de toegevoegde sombyte op
150	INC	LENGTE	
160	ZETWEG	STB	berg som van meestwaardige
170	*		bytes op

Let op dat u (B) na regel 100 niet meteen mag opbergen, want de store-instructie reset de overflow bit! Dus u moet eerst de overflow bit testen en pas daarna (B) opbergen.

Voorbeeld 6.62 (Sorteer getallen)

Ons probleem is nu een reeks getallen in een tabel te sorteren, zodanig dat de getallen in toenemende grootte achter elkaar in de tabel komen te staan.

"Bubble-sort" is een sorteertechniek waarbij telkens twee opeenvolgende getallen van de tabel met elkaar worden vergeleken. Indien het eerste getal groter is dan het tweede, worden de twee getallen onderling van plaats verwisseld. Begonnen wordt met het vergelijken (en eventueel verwisselen) van de eerste twee getallen. Daarna worden nummer twee en drie met elkaar vergeleken en eventueel verwisseld. Dit gaat door totdat alle paren van de tabel zijn vergeleken en eventueel verwisseld. Daarna wordt er van voren af aan opnieuw begonnen, totdat er in de gehele tabel geen verwisseling meer heeft plaatsgevonden. Op dat moment is de tabel gesorteerd. Bij deze werkwijze borrelt het grootste getal als het ware op naar het einde van de tabel. We zullen deze procedure demonstreren aan de hand van een getallenvoorbeeld (fig. 6.48). Het cijfer voor de ":" geeft aan hoe vaak de tabel is doorlopen. Achter de ":" staat de tot dusver gesorteerde tabel zelf. De onderstreepte getallen zijn de twee getallen, die met elkaar verwisseld zullen worden. We geven alleen de toestanden aan, waarbij de getallen verwisseld moeten worden.

```

0:  2 18 7  5 10 25 16
    2  7 18  5 10 25 16
    2  7  5 18 10 25 16
    2  7  5 10 18 25 16
1:  2  7 5 10 18 16 25
    2  5  7 10 18 16 25
2:  2  5  7 10 16 18 25
3:  2  5  7 10 16 18 25

```

Fig. 6.48 Getallenvoorbeeld bij de werking van bubble-sort

Nadat de tabel voor de tweede keer is doorlopen, wordt nogmaals opnieuw begonnen. Gedurende de tweede sorteeractie hebben er verwisselingen plaatsgevonden, zodat er nog een keer door de tabel heengelopen moet worden om te onderzoeken of alle getallen nu in de juiste volgorde staan.

Nadat de tabel één keer is doorlopen, staat het grootste getal achteraan. Is de tabel twee keer doorlopen, dan staan de twee grootste getallen aan het einde, enzovoorts. Dit betekent dat telkens één getallenpaar minder hoeft te worden vergeleken. Fig. 6.49 laat het structogram van het bubble-sort programma zien, waarna het programma volgt.

Gegeven is dat de lengte van de tabel op adres \$2800 staat en dat de tabel zelf op adres \$2801 begint. Het gaat hierbij om 1-byte two's complement getallen.

De ready flag wordt gebruikt om te onthouden of er een verwisseling van getallen heeft plaatsgehad. In dat geval moeten alle getalparen nog een keer worden vergeleken. Als de ready flag nog steeds geset is, nadat alle paren zijn vergeleken, staan de getallen in de goede volgorde.

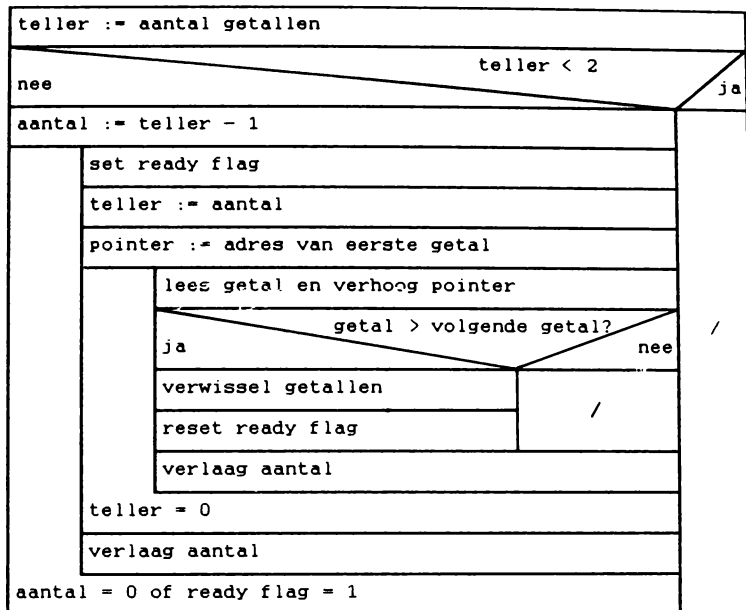


Fig. 6.49 Structogram voor bubble-sort

```

10      NAM      BUBBLE-SORT
20 *Progr.: J.G. Rouland
30 *Revisie 0    Datum: 29-03-1986
40 *
50 *BUBBLE sorteert een tabel met 1-byte two's complement
60 *getallen in volgorde van toenemende grootte. De lengte van
70 *de tabel staat op adres $2800 en de tabel begint op adres
80 *$2801.
90 *
100     ORG      $2400      beginadres objectcode
110 LENGTE EQU    $2800      lengte van de tabel
120 TABEL EQU    $2801      beginadres van de tabel
130 READY RMB    1          ruimte voor de ready flag
140 AANTAL RMB    1          ruimte voor aantal getallenparen
150 TELLER RMB    1          ruimte voor teller
160 *
170 BUBBLE LDB    LENGTE      lees aantal getallen
180      CMPB    #2          spring als aantal getallen
190      BLO     EINDE        kleiner is dan 2
200      DECB
210      STB     AANTAL        getallenparen
220 SETRDY LDA    #$80        set ready flag
230      STA     READY        (zijnde bit 7)

```

240	LDB	AANTAL	initialiseer
250	STB	TELLER	de teller
260	LDX	#LABEL	initialiseer de getallenpointer
270 NEXT	LDA	,X+	lees getal en verhoog pointer
280	CMPA	,X	spring als getal kleiner of
290	BLE	MINDER	gelijk is aan volgende getal
300 WISSEL	LDB	,X	laad het kleinere getal en berg
310	STB	-1,X	het op de vorige locatie op
320	STA	,X	zet het grotere getal daar achter
330	CLR	READY	
340 MINDER	DEC	TELLER	spring als nog niet alle paren
350	BNE	NEXT	zijn vergeleken
360	DEC	AANTAL	
370	BEQ	EINDE	als aantal=0 is rij gesorteerd
380	TST	READY	begin opnieuw als verwisselingen
390	BPL	SETRDY	van getallen hebben plaatsgehad
400	SWI		stop
410	END	BUBBLE	

In regel 390 kan BPL ook worden vervangen door BEQ, immers READY wordt 0 gemaakt als er een verwisseling heeft plaatsgevonden. Dit betekent ook dat in regel 220 elk getal ongelijk 0 gekozen had mogen worden.

6.7 Samenvatting

Datatransportinstructies

register-geheugen	opbergen in geheugen	ophalen uit geheugen
	STA	PSHS
	STB	PSHU
	STD	LDD
	STX	LDX
	STY	LDY
	STU	LDU
	STS	LDS

register-register	kopiëren	verwisselen
	TFR	EXG

Logische instructies

resetten	elke bit	per bit
	CLR	
	CLRA	ANDA
	CLRB	ANDB
		ANDCC

setten		per bit		
		ORA		
		ORB		
		ORCC		
inverteren	elke bit	per bit		
	COM			
	COMA	EORA		
	COMB	EORB		
schuiven	linksom		rechtsom	
	schuiven	roteren	schuiven	roteren
	LSL	ROL	LSR	ROR
	LSLA	ROLA	LSRA	RORA
	LSLB	ROLB	LSRB	RORB
bittesten	BITA			
	BITB			

Rekenkundige instructies

optellen	+ 1	+ willekeurig getal	unsigned
		INC	
		INCA	ADDA
		INCB	ADDB
			ADDD
			ABX
aftrekken	- 1	- willekeurig getal	0-getal
	DEC		NEG
	DECA	SUBA	SBCA
	DECB	SUBB	SBCB
		SUBD	NEGB
vermenigvuldigen	met 2	met willekeurig unsigned getal	
	ASL	MUL	
	ASLA		
	ASLB		
delen	door 2		
	2's compl. unsigned		
	ASR	LSR	
	ASRA	LSRA	
	ASRB	LSRB	

vergelijken	met 0	met willekeurig getal	
	TST		
	TSTA	CMPA	
	TSTB	CMPB	
		CMPD	
		CMPX	
		CMPY	
diversen		CMPU	
		CMPS	
	laad eff.	decimal	teken-
	adres	adjust A	uitbreiding
	LEAX	DAA	SEX
	LEAY		
	LEAU		
	LEAS		

Spronginstructies

onvoorwaardelijk	(L) BRA	(L) BRN	(L) BSR	SWI
	JMP	NOP	JSR	SWI2
				SWI3
			RTS	RTI
voorwaardelijk	unsigned	two's complement		bijz.
	(L) BLO	(L) BLT		instr.
	(L) BLS	(L) BLE		CWAI
	(L) BEQ	(L) BEQ		SYNC
	(L) BHS	(L) BGE		
	(L) BHI	(L) BGT		
	(L) BNE	(L) BNE		
	(L) BCC	(L) BVC		
	(L) BCS	(L) BVS		
		(L) BPL		
		(L) BMI		

6.8 Opgaven

- 1 Bepaal van de volgende instructies de opcode en eventueel de postbyte en geef aan hoeveel geheugenruimte elke instructie vergt en hoe lang de uitvoering van elke instructie duurt. Gegeven is dat een klokcyclus 1 μ s duurt. Gebruik appendix B.
 - a. CLRA
 - b. CMPX #\$5A3D
 - c. STD <\$DE
 - d. LDA \$6089
 - e. CMPY ,X++
 - f. ADDD -400,U
 - g. LDB [\$6C]

- h. LBCC \$A7
 - i. LEAX \$22,PC
 - j. LBRA BEGIN
- 2 Bepaal de postbyte van de volgende instructies:
- a. TFR A,DP
 - b. TFR DP,A
 - c. TFR X,U
 - d. TFR U,PC
 - e. EXG Y,S
 - f. EXG A,B
 - g. EXG S,Y
 - h. EXG X,D
- 3 Waarom wordt X in het programma INIFF van voorbeeld 6.11 vergeleken met \$1800 in plaats van met \$17FF?
- 4 Assembleer zelf met behulp van appendix B de eerste oplossing van voorbeeld 6.12 (BLMOVE).
Uit hoeveel bytes bestaat het verkregen objectprogramma?
Bepaal de executietijd van dit programma, indien er 1000 bytes moeten worden verplaatst en als de duur van een klokcyclus 1 μ s bedraagt.
- 5 Bepaal aan de hand van appendix B het aantal bytes en de executietijd van het programma van de tweede oplossing van voorbeeld 6.12 (BLMOVE) als er 1000 bytes verplaatst moeten worden. U hoeft het programma niet te assembleren. De duur van een klokcyclus bedraagt wederom 1 μ s.
- 6 Schrijf een sourceprogramma dat de inhoud van 2500 geheugenplaatsen, beginnend op adres \$5A00, verplaatst naar een buffer, die begint op adres \$5300. Ontwerp eerst het structogram!
Kunt u in plaats van te beginnen met de eerste byte van het programma ook beginnen met de laatste byte? Motiveer uw antwoord.
- 7 Schrijf een sourceprogramma dat de inhoud van een geheugenbuffer verplaatst. Begin met de laatste byte van de buffer. Het beginadres van de sourcebuffer staat op de geheugenlocaties met de adressen SOURCE en SOURCE+1, de lengte staat op de locaties met de adressen LENGTH en LENGTH+1 en het beginadres van de destinationbuffer staat op de locaties met de adressen DESTIN en DESTIN+1.
Ontwerp eerst het structogram.
- 8 Is het mogelijk om met één bittest-instructie te testen of A2, A5 en A7 alle drie 0 zijn?
Zo ja, welk masker moet u dan kiezen, zo nee, motiveer uw antwoord.

- 9 Is het mogelijk om met één bittest-instructie na te gaan of A4 en A6 beide 1 zijn?
Zo ja, welk masker moet u dan kiezen, zo nee, motiveer uw antwoord.
- 10 Ontwerp een programma dat een aantal packed BCD-getallen omzet in de bijbehorende ASCII-waarden.
Ontwerp eerst het structogram.
De packed BCD-getallen staan in een buffer met beginadres \$7000. Het aantal bytes is vermeld op de eerste locatie van deze buffer.
De ASCII-waarden moeten achtereenvolgens komen te staan in een outputbuffer met beginadres \$7800
- 11 Ontwerp een programma dat een aantal 16-bits woorden in serie naar een printer stuurt. Hierbij moet de MSB telkens als eerste verzonden worden. De woorden bevinden zich in een buffer. De twee eerste locaties in de buffer bevatten het aantal te verzenden woorden. De buffer begint op adres \$3000. De bits worden via bit 2 van register OUTPUT (met als adres \$FF01) naar de printer gestuurd. Telkens als er een nieuwe bit gereed staat, moet KLOK=1 worden en gedurende ongeveer 1 ms gelijk aan 1 blijven. Daarna moet KLOK telkens ongeveer 1 ms 0 zijn. KLOK is bit 5 van OUTPUT.
Ontwerp eerst het structogram!
- 12 Ontwerp een programma dat de checksum (de exclusive-or-functie van de data) van een aantal (16-bits) woorden controleert. De checksum bevindt zich in de eerste twee locaties van een tabel met beginadres \$5000. Het aantal woorden bevindt zich op de twee volgende locaties in de tabel. Daarna komen achtereenvolgens de woorden, waarover de checksum bepaald is. Klopt de checksum, dan moet de rij woorden worden afgesloten met \$0000. Is de checksum verkeerd, dan moet de reeks woorden worden afgesloten met \$FFFF.
Maak eerst een structogram!
- 13 Hoe wordt een negatief getal afgerond bij het uitvoeren van de onderstaande instructies?
ASRA
ADCA #0
- 14 Wat moet er aan het programma van voorbeeld 6.48 (ADDBCD) veranderen, indien n groter kan zijn dan 127?
- 15 Waarvoor dient de instructie op regel 200 van voorbeeld 6.49 (ZOEKTK)?

- 16 Tel n 8-bits unsigned getallen op. het aantal n bevindt zich in de geheugenlocatie \$7700. De getallen bevinden zich daarachter. Schrijf de som (te beginnen met de meestwaardige byte) direct achter de rij getallen.
Maak eerst een structogram.
- 17 Vertaal een packed BCD-getal van 2 bytes in accumulator D naar een gewogen binair getal en zet de uitkomst wederom in accumulator D.
Maak eerst een structogram.
- 18 Toon aan dat er bij het optellen van 2^{16} 1-byte two's complement getallen geen overflow kan ontstaan, indien voor de som 3 bytes gereserveerd worden.
- 19 Mag de instructie BLE in regel 230 van het programma BUBBLE-SORT van voorbeeld 6.62 vervangen worden door BLT? Motiveer uw antwoord.
- 20 Wat moet er aan het programma BUBBLE-SORT van voorbeeld 6.62 veranderen, opdat het werkt voor unsigned getallen?
- 21 Wat moet er aan het programma BUBBLE-SORT van voorbeeld 6.62 veranderen om 2-bytes two's complement getallen te sorteren?
- 22 Gegeven een (in oplopende volgorde) gesorteerde tabel van karakters in ASCII-code. Het aantal elementen bevindt zich op het adres LENGTE. De tabel start vanaf adres TABEL. Ontwerp een programma dat een ASCII-karakter laadt vanaf een register met als adres INPUT en dit karakter op de juiste plaats in de tabel tussenvoegt. Alle volgende karakters schuiven één plaats op en (LENGTE) moet worden aangepast.
Begin met het ontwerpen van een structogram.
- 23 Ontwerp een programma dat een ASCII-karakter laadt vanaf een register met als adres INPUT. Vervolgens wordt elk karakter dat gelijk is aan (INPUT), verwijderd uit de tabel van opgave 22. Ook moet (LENGTE) worden aangepast.
Begin met het ontwerpen van een structogram.

7 DE STACK

7.1 Specifieke doelstelling

Na het bestuderen van hoofdstuk 7 bezit u de volgende kennis:

- * U weet hoe de stack georganiseerd is.
- * U kunt de push- en de pull-instructies toepassen.

7.2 Organisatie van de stack

De stack is een deel van het read/write memory van de computer. Hij heeft echter een speciale functie.

Alvorens de stack nader te beschouwen, gaan we eerst wat dieper in op de rest van het geheugen.

Het programma staat in een deel van het geheugen, dat we zouden kunnen aanduiden als het programmeergeheugen. Hierin staan de instructies achter elkaar, zoals u reeds hebt gezien in paragraaf 3.5, waar de uitvoering van de instructie ADDA \$332F is behandeld. Elke volgende byte staat telkens op een hoger adres (zie fig. 7.1).

De data waarmee het programma zijn bewerkingen doet, staan eveneens in het geheugen. Dit deel van het geheugen kunnen we aanduiden als het datageheugen. Ook hier staan de diverse bytes van de data achtereenvolgens op een locatie met een oplopend adres (zie fig. 7.2). Stel we hebben het getal \$B53E. Dit is een getal van 2 bytes. De linker byte, \$B5, is de meestwaardige byte en wordt ook wel de high-byte genoemd. De rechter byte, \$3E, is de minst-significante byte en wordt de low-byte genoemd. In het datageheugen wordt eerst de high-byte geschreven (of gelezen) en daarna, op het volgende (dus hogere) adres, de low-byte.

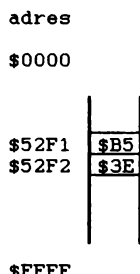
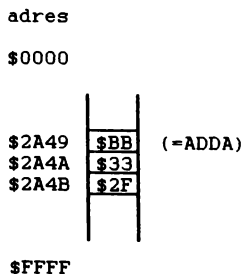


Fig. 7.1 Programmeergeheugen

Fig. 7.2 Datageheugen

De stack is een deel van de RAM van de computer en is

georganiseerd als een LIFO-memory (last-in/first-out). Dit is een geheugen, waarin de data die het laatst is weggeschreven, het eerst wordt opgehaald bij het lezen. Denk bijvoorbeeld aan een stapel borden in een cilinder met onderin een veersysteem, zodanig dat het bovenste bord net boven de rand van de cilinder uitsteekt. Dit betekent dat alleen dit bord weggenomen kan worden. Een bord aan de stapel toevoegen, kan eveneens alleen aan de bovenkant geschieden. Voegen we borden aan de stapel toe, dan daalt de stapel. Nemen we borden van de stapel af, dan stijgt de stapel. Zodoende is telkens alleen het bovenste bord bereikbaar. Bij het stapelgeheugen (de stack) schuift niet de gehele geheugeninhoud op en neer (zoals het geval is bij de stapel borden in de cilinder bij een restaurant), maar wordt de inhoud van de stack pointer groter of kleiner gemaakt. Dit gebeurt zodanig dat de stack pointer telkens wijst naar de laatste data, de data die aan de beurt is om gelezen te worden. Denk aan het bord dat aan de beurt is om van de stapel genomen te worden. De stack pointer wijst dus steeds naar de top van de stack.

7.3 Lezen en schrijven op de stack

Om ook met de "gewone" instructies, zoals lezen, schrijven, optellen en aftrekken, de stack te kunnen gebruiken, moet de data in de stack op dezelfde wijze opgeborgen worden als in het programmeergeheugen en in het datageheugen. Dat wil zeggen dat ook in de stack de low-byte op een hoger adres moet staan dan de high-byte.

adres

\$0000

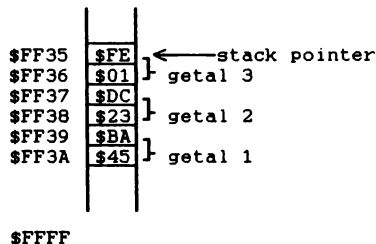


Fig. 7.3 De stack

In fig. 7.3 laten we een voorbeeld van de stack zien, waarin drie 2-bytes getallen staan. De stack pointer wijst naar de high-byte van getal 3, het getal dat het eerste aan de beurt is om gelezen te worden. Het lezen gaat nu goed, ook met de speciale leesopdracht pull (trek data van de stack). In de

operand van de pull-instructie wordt aangegeven om hoeveel bytes data het gaat en in welke registers die bytes moeten worden opgeborgen.

Bij de pull-instructie wordt de byte gelezen, die wordt aangewezen door de stack pointer. In het voorbeeld van fig. 7.3 is dit de high-byte \$FE. Daarna wordt de stack pointer met 1 verhoogd, zodat deze wijst naar de low-byte van getal 3 (\$01). Vervolgens wordt deze byte gelezen en de stack pointer wordt wederom met 1 verhoogd. Zodoende wijst de stack pointer naar de byte, die hierna aan de beurt is om te worden gelezen. Dit is de high-byte van getal 2. We zien dat de stack pointer telkens met 1 wordt verhoogd, nadat er een byte van de stack gehaald is met behulp van de speciale pull-instructie.

Het schrijven op de stack kan geschieden met behulp van de "gewone" store-instructies en met de speciale schrijfo opdracht push (duw data op de stack). Aangezien de stack een last-in/first-out geheugen is, moet het schrijven met de speciale push-instructie juist in omgekeerde volgorde plaatsvinden dan het lezen. Bij het schrijven van een getal, bestaande uit twee bytes, wordt eerst de stack pointer met 1 verlaagd, opdat deze een vrije stacklocatie aanwijst. Daarna wordt op die locatie de low-byte weggeschreven. Vervolgens wordt de stack pointer wederom met 1 verlaagd en tenslotte wordt op de dan aangewezen plaats de high-byte weggeschreven (zie fig. 7.4).

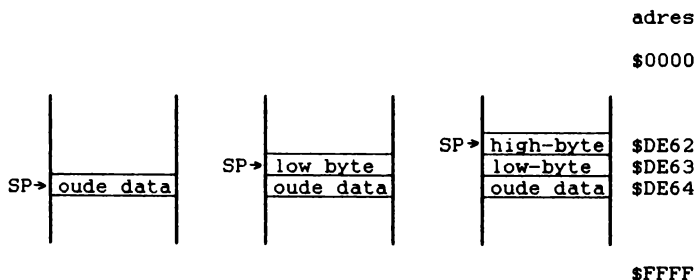


Fig. 7.4 De achtereenvolgende inhouden van de stack bij het wegschrijven van een 2-bytes getal met behulp van de speciale instructie push

7.4 Stack pointers

U hebt nu gezien dat de stack pointer telkens naar de top van de stack wijst.

Met behulp van de twee stack pointers U en S kunnen we twee stacks tegelijk gebruiken. Door X en Y als stack pointers toe te passen, hebben we zelfs de beschikking over vier stacks.

De stack pointers U en S zijn niet helemaal identiek. Wel zijn ze in het gebruik gelijk, met andere woorden, alle instructies die voor S bestaan, zijn er ook voor U. Zoals in hoofdstuk 8 zal blijken, wordt er bij het springen naar een subroutine door de microprocessor automatisch het terugkeeradres weggeschreven in de hardware stack, ook wel de system stack genoemd. Dit is de stack die aangewezen wordt door S.

Een subroutine bestaat uit een reeks instructies waar men, eventueel vanaf meerdere plaatsen, vanuit het programma naar toe kan springen. Na het uitvoeren van die reeks instructies, wordt het programma weer voortgezet op de plaats van waar weggesprongen werd.

Bij interrupts wordt de inhoud van de registers van het programmeermodel automatisch op de hardware stack geplaatst. Interrupts zijn (meestal korte) onderbrekingen van het programma met het doel om de microprocessor tussendoor even aan een andere taak te laten werken. Na het afhandelen van een interrupt worden de registers weer gevuld met de informatie van de hardware stack en de microprocessor kan het programma weer vervolgen, alsof er niets aan de hand is. Interrupts zullen uitgebreid worden behandeld in hoofdstuk 9.

U hebt hier twee voorbeelden gezien, waarbij de microprocessor automatisch iets op de hardware stack doet, zonder dat de programmeur er iets tegen kan ondernemen:

1. opbergen van terugkeeradres bij aanroep van subroutine en
2. opbergen van registerinhouden bij interrupt.

Derhalve wordt de hardware stack ook wel system stack genoemd. Hij is af en toe even nodig vanuit het systeem zelf, teneinde ervoor te zorgen dat bepaalde zaken goed worden geregeld.

De user stack, aangewezen door de stack pointer U, blijft van dit soort activiteiten verschoond. Dit betekent dat deze stack alleen door de programmeur, de gebruiker, wordt toegepast. Vandaar de naam user stack.

De stack pointers U en S kenmerken zich door twee specifieke instructies, namelijk push en pull (duw data op de stack, respectievelijk trek data van de stack). Hierbij wordt de stack pointer automatisch verlaagd (bij de push-instructie) en verhoogd (bij de pull-instructie). Gebruiken we U als stack pointer, dan moeten we de instructies PSHU, respectievelijk PULU toepassen. Is S de stack pointer, dan moeten we gebruik maken van de instructies PSHS en PULS.

We zullen nu laten zien dat het effect van deze instructies ook is te bereiken met de auto-increment/decrement-instructies. We doen dit door na te gaan wat er bij de verschillende instructies achtereenvolgens gebeurt, aan de hand van de voorbeelden 7.1 t/m 7.4.

Voorbeeld 7.1

PSHS	A	Verlaag S met 1 en zet daarna (A) op de locatie in de stack, die wordt aangewezen door S
STA	, -X	Verlaag X met 1 en zet daarna (A) in het geheugen op de plaats, aangewezen door X

Voorbeeld 7.2

PULS	B	Laad B met de inhoud van de locatie in de stack, die wordt aangewezen door S en verhoog daarna S met 1
LDB	, X+	Laad B met de inhoud van de geheugenplaats, die wordt aangewezen door X en verhoog daarna X met 1

Voorbeeld 7.3

PSHU	D	Verlaag U met 1, zet DL op de locatie in de stack, die wordt aangewezen door U, verlaag U wederom met 1 en zet tenslotte DH op de locatie in de stack, die nu wordt aangewezen door U
STD	, --Y	Verlaag Y met 2, zet daarna DH op de geheugenplaats, die wordt aangewezen door Y en zet tenslotte DL op de volgende geheugenplaats

Voorbeeld 7.4

PULU	D	Laad DH met de inhoud van de locatie in de stack, die wordt aangewezen door U, verhoog U met 1, laad daarna DL met de inhoud van de locatie op de stack, die wordt aangewezen door (de nieuwe) U en verhoog tenslotte U nog een keer met 1
LDD	, Y++	Laad DH met de inhoud van de geheugenplaats, die wordt aangewezen door Y, laad DL vervolgens met de inhoud van de volgende geheugenplaats en verhoog tenslotte Y met 2.

Uit bovenstaande voorbeelden moet u duidelijk zijn geworden, dat we met behulp van de auto-increment/decrement-instructies de indexregisters X en Y als stack pointers kunnen gebruiken. Naast de hardware stack en de user stack kunnen we dus ook nog een "X-stack" en een "Y-stack" definiëren. Let hierbij op dat we telkens de auto-increment/decrement met 1 moeten toepassen als het gaat om het opbergen of ophalen van 1 byte, terwijl we voor het opbergen en ophalen van een getal van 2 bytes de auto-increment/decrement met 2 moeten gebruiken.

7.5 Push- en pull-instructies

Met behulp van de push-instructies PSHS en PSHU kunnen we de inhoud van de registers op de hardware stack (of system stack), respectievelijk op de user stack plaatsen. We kunnen met behulp van één push-instructie meerdere registers op de stack opbergen. De registers, waarvan de inhoud op de stack moet worden gezet, moeten in het operandveld worden genoemd, onderling gescheiden door komma's. Op welke plaats de inhoud terecht komt, wordt bepaald door de inhoud van de betreffende stack pointer. Zoals u al weet, wijst de stack pointer steeds naar de laatst beschreven plaats in de stack. Bij de push-instructie wordt eerst de stack pointer met 1 verlaagd en daarna wordt er op de stack geschreven. Zodoende wordt voorkomen, dat er over vorige data heen wordt geschreven. Bedenk hierbij dat de stack in de richting van een lager adres wordt gevuld.

Voorbeeld 7.5

PSHS	A,Y,DP,CC	zet de inhoud van A, Y, DP en CC op de system stack (en verlaag system stack pointer S)
PSHU	D,X,S,PC	zet de inhoud van D, X, S en PC op de user stack (en verlaag user stack pointer U)

De volgorde waarin de registers op de stack worden gezet, ligt vast. Welke registerinhouden op de stack worden geplaatst, wordt bepaald door de waarde van de postbyte (zie fig. 7.5).

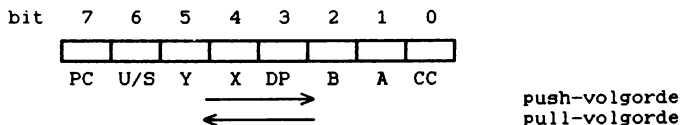


Fig. 7.5 Postbyte bij push- en pull-instructies

Wanneer een bepaalde bit van de postbyte gelijk aan 1 is, wordt het betreffende register bij een push-instructie op de stack gezet. Is de betreffende bit 0, dan wordt het overeenkomstige register niet op de stack gezet. Hierbij wordt begonnen bij bit 7 van de postbyte. Dit betekent dat eerst de program counter (al dan niet) op de stack gezet wordt, vervolgens de stack pointer, enzovoorts en tenslotte het condition code register. Bij 2-bytes registers wordt eerst de low-byte op de stack gezet en daarna de high-byte. Dit teneinde de volgorde van de informatie in het datageheugen en in de stack gelijk te houden.

Bij een 0 in een bit van de postbyte wordt het betreffende register overgeslagen. Hierbij wordt er geen locatie op de

stack overgeslagen! De registers hebben dus geen vaste plaats op de stack. Wordt een registerinhoud niet op de stack gezet, dat wordt de betreffende stacklocatie benut om de inhoud van een ander register op te slaan. De op te bergen registerinhouden komen dus aaneengesloten op de stack te staan.

Bit 6 van de postbyte behoeft nog enige uitleg. Bij deze bit zijn immers de beide stack pointers genoemd. Nu heeft het weinig zin om de inhoud van de stack pointer op te bergen op de plaats, die hij zelf aanwijst. Derhalve wordt door de microprocessor de inhoud van S op de user stack gezet (bij PSHU dus) en die van U op de system stack (bij PSHS), mits bit 6 gelijk aan 1 is natuurlijk. Anders wordt de inhoud van de stack pointer helemaal niet op de stack gezet.

Na het vertalen van de instructie PSHS A,Y,DP,CC heeft de postbyte de waarde %00101011, terwijl de postbyte van PSHU D,X,S,PC de waarde %11010110 zal hebben. De assembler zorgt ervoor, dat de juiste bits van de postbyte geset zullen worden. De volgorde van de namen van de registers is niet van belang, aangezien de volgorde van wegschrijven vastligt in de microprocessor zelf.

Nu u de postbytes kent, weet u ook wat de volgorde van de registers op de stack is (zie fig. 7.6 en 7.7).

We hebben aangenomen, dat voor het uitvoeren van de push-instructies geldt dat (S)=\$A0B2 en (U)=\$6240.

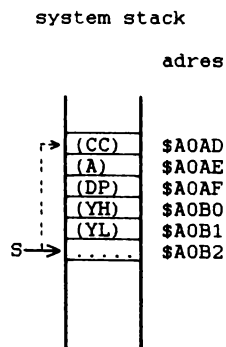


Fig. 7.6 PSHS A,Y,DP,CC

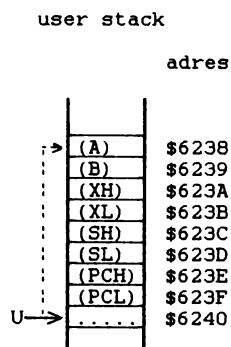


Fig. 7.7 PSHU D,X,S,PC

In fig. 7.6 en 7.7 ziet u dat na de push-instructies geldt, dat (S)=\$A0AD en (U)=\$6238. S en U wijzen wederom naar de top van de stack, behalve die van de stack pointer.

Bij push-instructies gaat de oorspronkelijke informatie van de betreffende geheugenplaats(en) verloren! De inhoud van de registers blijft echter ongewijzigd!

Push- en pull-instructies zijn immediate addressing instructies. In fig. 7.8 ziet u dat deze instructies de bits van het condition code register ongewijzigd laten, behalve natuurlijk de instructies PULS CC en PULU CC.

Voorbeeld 7.6

```

SUBSET EQU    %00110110    zet de registers Y, X, B en A
      PSHU    #SUBSET      op de user stack
      PSHS    #$FF        zet alle registers op de system
                           stack (uitgezonderd S)
  
```

instr	addressing modes					beschrijving	(CC) bit 53210 HNZVC
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ #+	extend op ~ #	inher op ~ #		
PSHS	34 g 2					reg. naar S-stack	----
PSHU	36 g 2					reg. naar U-stack	----
PULS	35 g 2					reg. van S-stack	----
PULU	37 g 2					reg. van U-stack	----

Opm.: g. 5 klokcycli + 1 cyclus voor elke byte die wordt gepushed of gepulled.

Fig. 7.8 Overzicht van de push- en pull-instructies

Zoals we met behulp van de push-instructies registerinhouden op de stack kunnen zetten, zo kunnen we omgekeerd met behulp van de pull-instructies PULS en PULU informatie van de system stack, respectievelijk van de user stack halen om deze in de registers te zetten.

Het lezen van de stack gebeurt in omgekeerde volgorde als het schrijven op de stack. Eerst wordt een byte gelezen, daarna wordt de betreffende stack pointer met 1 verhoogd. Dan wordt eventueel de volgende byte gelezen, enzovoorts.

Welke registers er vanaf de stack moeten worden gevuld met data, maken we kenbaar door de namen van de betreffende registers, van elkaar gescheiden door komma's, op te geven in het operandveld. De assembler bepaalt hieruit de waarde van de postbyte, net als bij de push-instructie. Alleen wordt er bij de uitvoering van een pull-instructie begonnen bij bit 0 (in plaats van bij bit 7). De volgorde van het lezen in de stack is omgekeerd aan die van het schrijven.

Als bit 6 van de postbyte 1 is, wordt bij een PULS het register U gevuld, terwijl bij een PULU het register S gevuld wordt.

Voorbeeld 7.7

```

PULS  B,PC,DP,Y,U    vul de registers B, PC, DP, Y
                     en U met informatie van de
                     system stack (en verhoog de
                     system stack pointer S)
  
```


PULU D,CC,S,X

vul de registers D, PC, S en X met informatie van de user stack (en verhoog de user stack pointer U)

In fig. 7.9 en 7.10 ziet u de twee instructies van voorbeeld 7.7 uitgewerkt. Op de stack is willekeurige informatie gezet en verder is ervan uitgegaan dat (S)=\$6AA3 en (U)=\$ABF0.

Na de PULS-instructie van het voorbeeld van fig. 7.9 wordt het programma niet vervolgd met de instructie, die staat op adres \$3027, maar met die welke staat op adres \$5499. Immers de inhoud van de program counter, \$3027, wordt tijdens het uitvoeren van de PULS-instructie gewijzigd in \$5499.

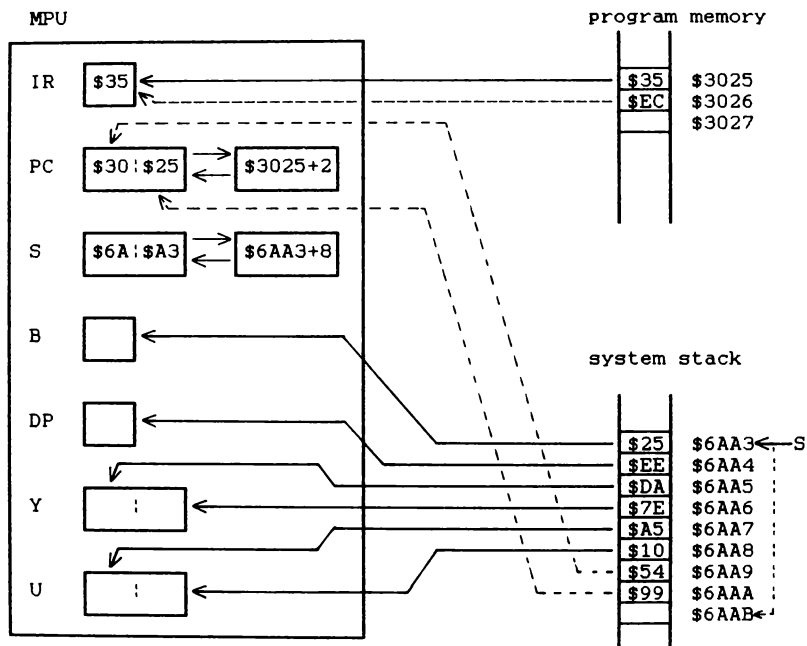


Fig. 7.9 Uitvoering van de instructie PULS B,PC,DP,Y,U

De system stack pointer S wordt in dit voorbeeld met 8 verhoogd en krijgt dus de waarde \$6AAB. Er worden hier 8 bytes van de system stack gehaald.

Bij het uitvoeren van de PULU-instructie van het voorbeeld van fig. 7.10 wordt de user stack pointer U verhoogd met 7. Er worden immers 7 bytes van de user stack gelezen. De inhoud van U wordt \$ABF7.

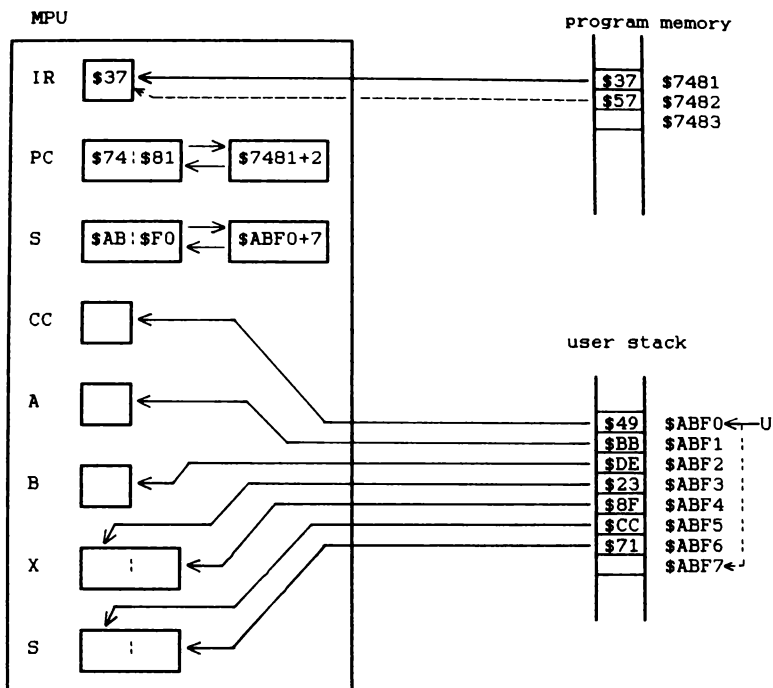


Fig. 7.10 Uitvoering van de instructie PULU D,CC,S,X

Bij pull-instructies gaat de inhoud van de geheugenplaatsen niet verloren! Natuurlijk worden wel de betreffende registers overschreven, zodat de oude informatie in die registers verloren is gegaan.

Fig. 7.11 laat zien dat bij een push, gevolgd door een pull met dezelfde registers, de inhoud van die registers niet verandert, dankzij het feit dat de pull in exact de omgekeerde volgorde wordt uitgevoerd als de push.

De volgorde van de namen in het operandveld is niet van belang. De assembler vertaalt deze namen immers in de juiste postbyte en in de postbyte ligt de volgorde vast van het lezen en schrijven op de stack. Het is wel zaak ervoor te zorgen dat dezelfde registers in het operandveld van de push- en de pull-instructie voorkomen, willen we de op de stack opgeborgen informatie later weer in dezelfde registers terugkrijgen!

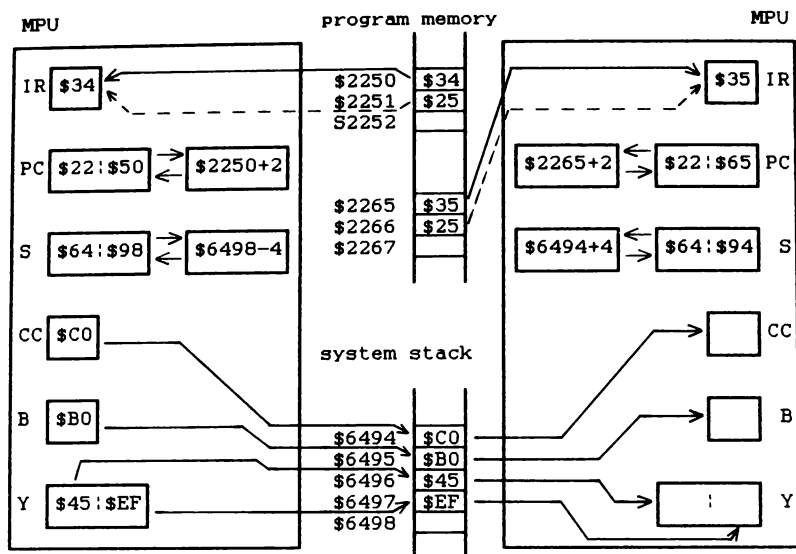


Fig. 7.11 Uitvoering van PSHS Y, CC, B gevolgd door PULS B, Y, CC

7.6 Samenvatting

Data worden op de stack op dezelfde wijze opgeborgen als in het datageheugen. Zowel op de stack als in het datageheugen staat de high-byte van de data op een lager adres dan de low-byte.

Met behulp van de instructies PSHS en PSHU, respectievelijk PULS en PULU, kan er data op de system stack, dan wel op de user stack worden geschreven of gelezen.

Met behulp van auto-decrement-, respectievelijk auto-increment-instructies kunnen de indexregisters X en Y als stack pointers worden toegepast. Op deze manier kan er een X-stack en een Y-stack worden gedefinieerd.

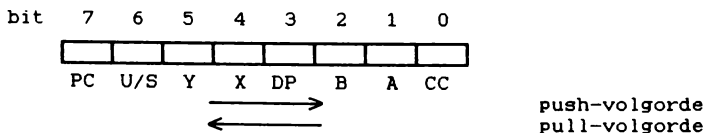


Fig. 7.12 Postbyte bij push- en pull-instructies

De volgorde waarin registers op de stack worden opgeborgen en van de stack worden gehaald, ligt vast in de microprocessor (zie ook fig. 7.12).

Om welke registers het gaat, wordt bepaald in de postbyte. Een 1 betekent dat het betreffende register wordt gepushed of gepulled, een 0 betekent dat het betreffende register wordt overgeslagen (zie fig. 7.12).

7.7 Opgaven

- 1 Bepaal de postbytes en de executietijd van de volgende instructies:

- a. PSHS A,CC,DP,X,U
- b. PULU X,Y,D,CC

- 2 Gegeven:

(D)=\$ABCD	(X)=\$55CC
(DP)=\$01	(Y)=\$EF27
(CC)=\$83	(U)=\$51F0
(PC)=\$2080	(S)=\$4E8A

Teken de inhoud van de stack en bepaal de inhoud van de stack pointer na het uitvoeren van de volgende instructie:

- a. PSHS #\$CA
- b. PSHU #\$75

- 3 Gegeven:

(D)=\$D1D0	(X)=\$0011
(DP)=\$A8	(Y)=\$2233
(CC)=\$0C	(U)=\$4455
(PC)=\$8079	(S)=\$EEFF

Bepaal de inhoud van de registers na het uitvoeren van de instructie PSHU D,PC,S,DP gevolgd door de instructie PULU S,A,CC,Y.

8 MODULAIR PROGRAMMEREN

8.1 Specifieke doelstelling

Nadat u dit hoofdstuk hebt bestudeerd, moet u het volgende kunnen:

- * Weten wat modulaire programma's zijn.
- * De definitie van een subroutine kennen.
- * De programmafow bij gebruik van subroutines kennen.
- * Subroutines gebruiken in uw programma's.
- * De definitie van macro's kennen.
- * Macro's, eventueel met parameters, toepassen.
- * De diverse methoden van parameteroverdracht tussen programmamodules kennen.
- * De parameteroverdracht tussen de programmamodules verzorgen en wel via registers, via vaste geheugenplaatsen en via een pointerregister.
- * Programmamodules van een goede documentatie voorzien.
- * Weten hoe een gestructureerde projectaanpak verloopt.

8.2 Inleiding

Nu u de instructieset en de diverse adresseringsmethoden van de MC6809 kent, wordt het tijd om gebruik te leren maken van een zeer handig hulpmiddel bij het programmeren, namelijk het toepassen van subroutines en macro's. Subroutines en macro's worden meer en meer gebruikt ten behoeve van het ontwerpen van modulaire programma's. Derhalve gaan we eerst nader in op modulair programmeren.

Voordelen van modulair programmeren

1. Modulaire programma's zijn overzichtelijker en beter te begrijpen.

Zeer grote en/of ingewikkelde programma's zijn niet erg overzichtelijk. Hierin kan een aanzienlijke verbetering worden aangebracht door zulke programma's te verdelen in kleinere deelprogramma's, die elk een deel van het totale probleem oplossen.

Zo'n deelprogramma wordt een module genoemd. Het opbouwen van een programma uit diverse modules heet modulair programmeren. Een module is een programmadeel dat een afzonderlijke eenheid vormt (zie fig. 8.1).

Een module zelf kan eventueel weer zijn onderverdeeld in andere

modules, die mogelijk ook weer in modules zijn opgedeeld. Zo kan er een soort hiërarchie van (hoofd)modules en submodules ontstaan. Fig. 8.1 laat hiervan een voorbeeld zien. Uit deze figuur blijkt tevens de top-down aanpak. Eerst is er een globale probleembeschrijving. Het probleem wordt vervolgens verdeeld in een aantal deelproblemen. Deze worden daarna nogmaals verdeeld in kleinere onderdelen. Zo ontstaan er tenslotte gedeelten, die beter zijn te overzien.

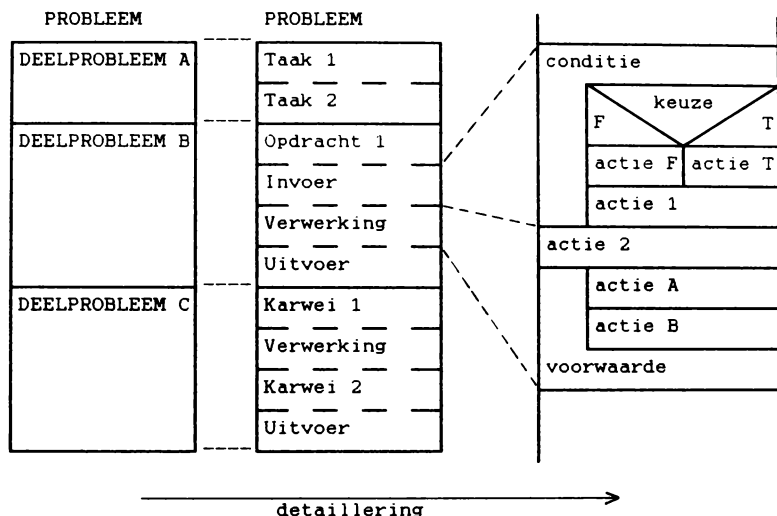


Fig. 8.1 Modulair programma

2. Een modulair programma kan eenvoudiger worden ontworpen en getest.

Een module is eenvoudiger te schrijven en te debuggen (fouten eruit halen) dan een volledig programma. Een deelprobleem is beter oplosbaar dan het gehele probleem. Een deelprobleem is kleiner en derhalve beter te overzien. Elk gedeelte wordt vertaald in een structogram, waaruit uiteindelijk het programma van de bijbehorende module wordt afgeleid.

3. Modulaire programma's zijn gemakkelijker te wijzigen.

Een module kan gemakkelijk uit de rest van het programma worden losgemaakt om elders (in hetzelfde programma of in andere programma's) te worden hergebruikt, dan wel om te worden vervangen door een andere module. Blijkt later dat een bepaalde module niet geheel voldoet, dan kan deze vrij gemakkelijk worden vervangen door een verbeterde module.

Omdat een module een zelfstandig, afgesloten programma-onderdeel is, hoeft hierbij niet het gehele programma te worden onderzocht om na te gaan of een wijziging van een module

consequenties heeft elders in het programma. Zolang de specificatie van de inkomende gegevens en de uitgaande gegevens van de module niet verandert, heeft het wijzigen van de module geen consequenties voor de rest van het programma.

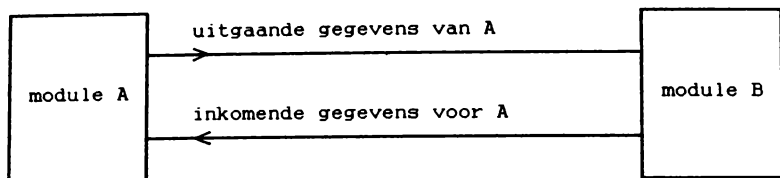


Fig. 8.2 De interface tussen twee modules

De verbindingen tussen de diverse modules noemt men de interfaces. Het gaat hierbij niet om fysische verbindingen (draden), maar om de soort en het aantal inkomende gegevens en uitgaande gegevens (zie fig. 8.2).

De interfaces tussen de diverse modules moeten goed worden vastgelegd. Er moet nauwkeurig worden omschreven waar elke module zijn invoergegevens vandaan haalt en waar de uitvoergegevens worden opgeborgen.

4. Aan een modulaair programma kan gemakkelijker door meerdere ontwerpers tegelijk worden gewerkt.

Elke ontwerper kan een of meerdere modules voor zijn rekening nemen.

Het is erg belangrijk dat de modules goed worden gedocumenteerd, vooral als ze door verschillende ontwerpers worden gemaakt.

Van modules die in vele programma's voorkomen, de zogenaamde standaardmodules, kunnen bibliotheken worden aangelegd. Deze modules hoeven slechts één keer ontworpen te worden om vervolgens in vele programma's te worden toegepast. Denk bijvoorbeeld aan input- en outputroutines. Modules die in een bibliotheek zijn opgenomen, worden door een linkage editor aan het programma toegevoegd. Een linkage editor is een standaardprogramma dat de diverse modules tot één programma aan elkaar knoopt. Hierbij worden de kruisreferenties, de verwijzingen van de ene module naar de andere, gerealiseerd.

5. De voortgang van de ontwikkeling van een modulaair programma is beter te controleren.

Bij modulaair programmeren krijgt men een beter idee van de voortgang van een softwareproject. Men kan eenvoudig zien welke modules al gereed zijn en welke nog niet. Als het hele probleem daarentegen met een enkel programma wordt opgelost, is het aanzienlijk moeilijker na te gaan hoeveel nu precies gereed is en wat er nog gedaan dient te worden.

Nadelen van modulaire programma's

Modulaire programma's kosten extra executietijd en extra geheugenruimte.

Doorgaans is er enige overhead nodig om de modules zelfstandige stukken software te laten zijn. Hierbij valt te denken aan het niet veranderen van de registerinhouden van de microprocessor, zodat de informatie in deze registers voor het programma buiten de module niet verloren gaat. Dit kan worden gerealiseerd door de registerinhouden aan het begin van de module op te bergen in het geheugen, bijvoorbeeld op de stack. Aan het einde van de module worden de registerinhouden dan weer teruggehaald uit het geheugen. De instructies die hiermee gemoeid zijn, kosten zowel geheugenruimte als tijd.

Zoals uit fig. 8.1 blijkt, kunnen modules meerdere malen in het programma voorkomen. Vooral in dat geval is het handig om gebruik te maken van subroutines en macro's. Subroutines en macro's zijn op zichzelf nog geen modules, maar kunnen onderdelen zijn van modules. Een module zorgt voor de overdracht van de juiste gegevens naar de subroutine of naar de macro en roept de subroutine of macro aan.

8.3 Subroutines

8.3.1 Programmaf low bij subroutines

Een subroutine is een programmadeel dat vanuit elk punt in het programma kan worden aangeroepen, teneinde te worden uitgevoerd. Na het uitvoeren van de subroutine wordt het programma voortgezet op de plaats, waar het door de subroutine werd onderbroken. Een aanroep van een subroutine heet subroutine call. Deze kan worden gerealiseerd met behulp van de instructies JSR, BSR of LBSR (Jump, Branch of Long Branch to SubRoutine). Om terug te keren naar het aanroepende programma, moet de subroutine worden afgesloten met de instructie RTS (ReTurn from Subroutine). Opdat naar de juiste plaats in het programma wordt teruggesprongen, moet het terugkeeradres worden bewaard. Dit zou kunnen gebeuren in een apart register of in een vaste geheugenplaats. In dit geval zou echter het terugkeeradres verloren gaan, indien binnen in de subroutine weer een subroutine wordt aangeroepen. Hierbij wordt immers het terugkeeradres ook in dat register of in die geheugenplaats opgeborgen. Zodoende gaat het oude terugkeeradres van het hoofdprogramma verloren. Om dit te voorkomen moet het terugkeeradres telkens op een andere plaats worden opgeborgen, indien er achter elkaar subroutines worden aangeroepen. Dit kan op eenvoudige wijze worden gerealiseerd door het terugkeeradres op de stack op te bergen.

Het springen naar een subroutine en het terugkeren in het aanroepende programma gaat als volgt in zijn werk:
 Als de volledige instructie JSR, BSR of LBSR door de processor is opgehaald uit het geheugen, wijst de program counter naar de volgende instructie in het geheugen. Dit is de instructie die moet worden uitgevoerd, nadat de subroutine is afgehandeld. De microprocessor voert nu automatisch de instructie PSHS PC uit. Hierdoor wordt het terugkeeradres op de system stack gezet. Vervolgens bepaalt de microprocessor met behulp van de operand het beginadres van de subroutine en zet dit in de program counter. Dit heeft tot gevolg dat het programma wordt voortgezet met de eerste instructie van de aangeroepen subroutine. De program counter wordt bij het ophalen van instructies telkens weer verhoogd, zodat de opeenvolgende instructies achter elkaar worden uitgevoerd, net als bij een gewoon programma. Is de instructie RTS opgehaald en gedecodeerd, dan voert de microprocessor automatisch de instructie PULS PC uit. Hierdoor wordt het terugkeeradres van de stack gehaald en in de program counter gezet, zodat het programma wordt voortgezet met de instructie na de subroutine call.

Bij de subroutine call en bij de instructie RTS voert de microprocessor de volgende acties uit:

JSR	SUBR	(S):=(S)-1	de inhoud van de program counter,
		((S)):=((PCL))	zijnde TERUG (zie fig. 8.3), wordt
		(S):=(S)-1	op de eerstvolgende vrije ruimte op
		((S)):=((PCH))	de system stack gezet en de program
		(PC):=SUBR	counter wordt geladen met het
			beginadres SUBR van de aangeroepen
			subroutine
RTS		(PCH):=((S))	de program counter wordt geladen met
		(S):=(S)+1	informatie van de system stack,
		(PCL):=((S))	zijnde het terugkeeradres TERUG,
		(S):=(S)+1	zodat (PC):=TERUG

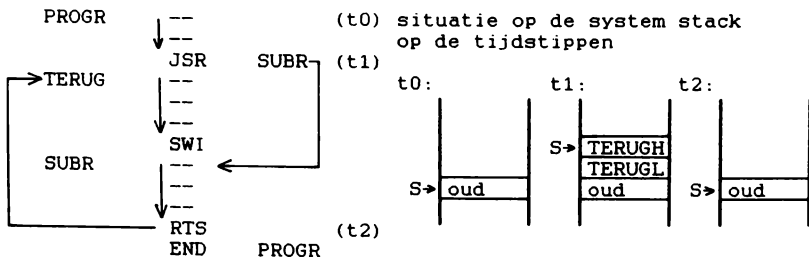
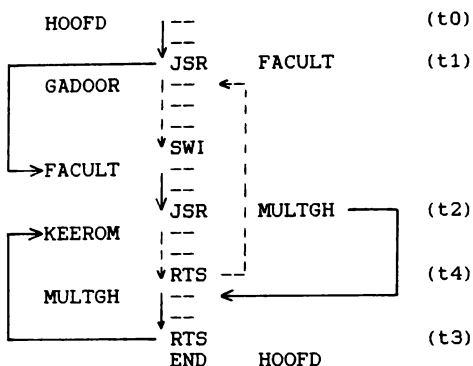


Fig. 8.3 Programmaflow bij een subroutine call

Fig. 8.3 toont de flow van het programma bij het voorkomen van subroutines. De pijlen geven hierin aan hoe de flow van het

dat telkens wordt teruggesprongen naar het vorige aanroepende programma.

Als voorbeeld is een programma gekozen dat de subroutine FACULT aanroept. FACULT berekent de faculteit van een 1-byte getal n ($n! = n \cdot (n-1) \cdot \dots \cdot 4 \cdot 3 \cdot 2 \cdot 1$). Op zijn beurt roept FACULT een subroutine MULTGH aan, die ervoor zorgt dat twee getallen met elkaar worden vermenigvuldigd. Hierbij kan een van de getallen groter zijn dan één byte ($6! = 720$ en past al niet meer in één byte).



situatie op de system stack op de tijdstippen

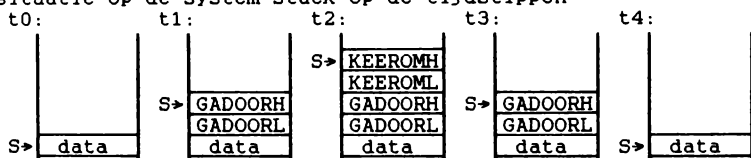


Fig. 8.5 Programmaflow bij geneste subroutines

Het is van groot belang dat het terugkeeradres op de stack niet wordt overschreven met andere informatie en dat de stack pointer naar het terugkeeradres wijst, indien de instructie RTS wordt uitgevoerd. Dit houdt in dat er binnen een subroutine precies even veel bytes van de stack moeten worden gelezen als er van tevoren zijn opgeborgen. Bij een PSHS-instructie wordt immers eerst de stack pointer verlaagd en daarna wordt er op de stack geschreven. Zodoende wordt er telkens op vrije plaatsen geschreven. Een eventueel opgeborgen terugkeeradres wordt derhalve niet overschreven, tenzij er eerst een PULS is uitgevoerd.

Bij een PULS-instructie gebeurt juist het omgekeerde. Eerst wordt er gelezen en daarna wordt de stack pointer verhoogd. Hierbij wordt de laatst opgeborgen data het eerst teruggehaald, zodat de stack pointer vervolgens naar de data wijst, die eerder werd weggeschreven. Dit is dan weer het terugkeeradres, mits van tevoren exact even veel bytes op de stack gepushed zijn.

8.3.2 Voordelen en nadelen van subroutines

Voordelen van subroutines

1. Subroutines maken het sourceprogramma en het objectprogramma altijd korter, indien zij meerdere malen nodig zijn in het programma.

De subroutine komt zowel in het sourceprogramma als in het objectprogramma slechts één keer voor. Op die plaatsen in het programma waar de subroutine moet worden uitgevoerd, hoeft deze slechts te worden aangeroepen.

2. Het programma wordt door het gebruik van subroutines beter leesbaar.

De detailoplossing van een deelprobleem vindt binnen een subroutine plaats. Om deze laatste reden is het meestal verstandig om programmadelen te vervangen door subroutines, ook al komen deze onderdelen slechts één keer in het programma voor. In dit geval wordt de subroutine gebruikt om te komen tot een modulaire opbouw van het programma.

Subroutines moeten niet te kort zijn, omdat anders de overzichtelijkheid van het programma niet toeneemt vanwege het grote aantal subroutines, die slechts enkele instructies vervangen. Het is beter deze instructies gewoon in het programma op te nemen. Bovendien kosten kleine subroutines vrij veel geheugenruimte ten gevolge van de overhead (de subroutine call en RTS). Ook moeten subroutines niet te groot zijn, omdat in dat geval de overzichtelijkheid van de subroutine zelf verloren gaat. Zulke lange subroutines kunnen dan beter worden opgebouwd uit een aantal kleinere subroutines. Een vuistregel is dat subroutines ongeveer 10 tot 50 instructies moeten bevatten.

Nadeel van subroutines

De executie van een programma met subroutines kost meer tijd. Elke keer dat een subroutine moet worden uitgevoerd, dient deze eerst te worden aangeroepen en aan het einde van de subroutine moet worden teruggesprongen naar het aanroepende programma. Dit kost telkens twee extra instructies, te weten de subroutine call en de spronginstructie RTS. Hiermee is extra tijd gemoeid. Doorgaans is dit echter geen probleem.

Overhead

Subroutines maken meestal gebruik van een of meerdere registers. Dit betekent dat de informatie in die registers verloren gaat. Indien deze informatie voor het aanroepende programma van belang is, moet deze gered worden. Dit kan gebeuren in een vrije geheugenruimte. Een ideale plaats om dit te doen is de stack. De programmeur hoeft dan niet zelf bij te houden waar nog vrije geheugenruimte aanwezig is. De stack pointer wijst steeds naar de laatst gevulde positie op de stack. Daarboven (op de lagere adressen) begint de vrije ruimte.

8.3.3 Het redden van registerinhouden

Registerinhouden kunnen ten behoeve van het aanroepende programma worden gered door

1. het aanroepende programma zelf.

Voordeel: Alleen de registers die van belang zijn voor het aanroepende programma, hoeven te worden gered.

Nadeel: De registers moeten bij elke aanroep van de subroutine worden gered, hetgeen telkens instructies kost in het aanroepende programma.

2. de subroutine.

Voordeel: De save-instructies komen nu slechts één keer voor en wel aan het begin van de subroutine. Hierdoor worden source- en objectprogramma korter, als de subroutine vaker wordt aangeroepen.

Nadeel: Er kan teveel tijd verloren gaan, omdat er mogelijk registerinhouden worden gered, die voor het aanroepende programma geen zinvolle informatie bevatten.

We herschrijven nu de subroutine WACHT zodanig, dat de informatie in de registers van de microprocessor niet veranderd is na terugkeer in het aanroepende programma. In deze subroutine wordt register D gebruikt. Bedenk dat het condition code register impliciet ook wordt gebruikt, omdat vrijwel elke instructie een of meerdere flags beïnvloedt. Dit register moet dus eveneens worden gered, want het is mogelijk dat het aanroepende programma de waarden nodig heeft, die de flags voor de call hadden.

10	WAIT	PSHS	D,CC	red D en CC
20		LDD	#13	
30	DECR	SUBD	#1	
40		BNE	DECR	
50		PULS	D,CC	herstel D en CC
60		RTS		

Vaak worden de instructies van de regels 50 en 60 gecombineerd tot

PULS D,CC,PC

Dit kan vanwege het feit dat de program counter bij een pull-instructie het laatst van de stack wordt gehaald. Met andere woorden eerst worden de overige registers hersteld en daarna volgt PC. De instructie RTS moet dan achterwege worden gelaten, want het terugkeeradres is reeds van de stack gehaald. Hiermee is 1 byte geheugenruimte gespaard alsmede 3 klokcycli aan executietijd. RTS kost 1 byte geheugenruimte en 5 klokcycli. De instructies PULS D,CC en PULS D,CC,PC kosten evenveel bytes, immers de postbyte geeft aan welke registers wel en welke niet vanaf de stack moeten worden geladen. De executietijd van PULS

bedraagt 5 klokcycli + 1 klokcyclus voor elke byte die van de stack wordt gehaald (zie appendix B). PC is een 2-bytes register, dus de PULS-instructie duurt 2 cycli langer door toevoegen van PC. De 5 cycli van RTS worden bespaard, dus de bezuiniging bedraagt 3 cycli.

In de voorbeelden zal de instructie RTS expliciet opgegeven worden ten behoeve van de leesbaarheid van het programma. Het einde van de subroutine wordt zodoende beter herkend.

8.3.4 De system stack pointer

Bij het gebruik van subroutines, of meer in het algemeen bij het gebruik van de system stack, dient de programmeur er zeker van te zijn dat de stack pointer S goed is geïnitieerd. De stack zal zich altijd in het read/write memory moeten bevinden, omdat er bij de aanroep van een subroutine automatisch op de system stack wordt geschreven door de processor.

Het initialiseren van de stack pointer kan worden gedaan door

1. het operating system
2. de programmeur

Bij het opstarten van de computer moet er worden gezorgd voor het laden van S met het topadres van de stack, zodanig dat er voldoende ruimte voor de stack beschikbaar is. In computersystemen waar tegelijkertijd meerdere programma's worden uitgevoerd, mag u de stack pointer S doorgaans niet wijzigen. U loopt hierbij het gevaar dat een ander programma in uw stackruimte terecht komt. Bovendien kan het operating system in de war raken. Bij multi-tasking en multi-user systemen zorgt het operating system ervoor dat elke taak of gebruiker een eigen stack toegewezen krijgt. Bij het overdragen van de processor aan een andere taak of gebruiker, past het operating system de stack pointer aan. Wordt de stack pointer door het gebruikersprogramma gewijzigd, dan kan het programma geheel verkeerd aflopen. Bovendien kunnen andere programma's worden vernield.

Maak dus gebruik van de stack, die wordt toegewezen door het operating system.

In besturingssystemen voor bijvoorbeeld wasmachines, moet de programmeur de stack zelf definiëren, omdat het operating system hier ontbreekt.

Voor de stack kan de benodigde ruimte worden gecreëerd met behulp van een RMB-directive, bijvoorbeeld op de volgende manier:

STACK	RMB	500
PRGRM	LDS	#PRGRM
	--	--

In dit voorbeeld wordt een ruimte van 500 bytes gecreëerd ten

behoefte van de stack. Na deze ruimte volgt onmiddellijk het programma PRGRM. Hierin wordt eerst de stack pointer geïntialiseerd. (S):=PRGRM, zodat S wijst naar de instructie LDS. Bij de instructie PSHS wordt de stack pointer eerst verlaagd en daarna wordt er geschreven. Zodoende wordt de eerste byte gepushed op de plaats direct boven de instructie LDS.

Deze methode is aan te bevelen in bijvoorbeeld ontwikkelssystemen waar slechts één taak tegelijk wordt uitgevoerd en waarbij uw programma wordt geladen in een door uzelf bepaald geheugengebied (met behulp van een ORG-directive). Het kan zijn dat een vorig programma de stack pointer in datzelfde geheugengebied geïntialiseerd heeft. Bij een PSHS-instructie zou hierdoor uw programma verminkt worden. Vandaar dat het verstandig is in dit geval de stack pointer zelf van de juiste waarde te voorzien.

8.4 Macro's

Om modulaire programma's te realiseren kan ook gebruik worden gemaakt van macro's. Een macro is een groep instructies die met een symbolische naam wordt aangeduid. In het programma wordt een macro aangeroepen met een zogenaamde macro call. De assembler vervangt deze macro call tijdens het assembleren door de betreffende groep instructies. Komt deze groep instructies meerdere malen voor, dan kan worden volstaan met de macro calls op de diverse plaatsen in het programma op te nemen. De macro zelf hoeft slechts één keer geschreven te worden. Macro's die in meerdere programma's voorkomen, kunnen in een bibliotheek (library) worden opgenomen.

U zult u nu terecht afvragen wat het verschil is tussen een macro en een subroutine.

Subroutine versus macro

De subroutine call wordt door de assembler als zodanig vertaald in machinecode. Bij de executie van het programma wordt er naar de subroutine gesprongen en na uitvoering van de subroutine wordt weer teruggesprongen naar het aanroepende programma. Er wordt zoveel keren naar de subroutine gesprongen als er calls zijn. De subroutine zelf komt zowel in het source program als in het object program slechts 1 keer voor. Komen bepaalde groepen instructies vaker voor, dan leidt het gebruik van subroutines behalve tot kortere source programs ook tot kortere object programs. De executietijd van een programma met subroutines neemt toe vanwege de extra spronginstructies JSR, BSR of LBSR en RTS.

Een macro call wordt door de assembler niet als zodanig vertaald, maar wordt vervangen door de macro zelf. De

instructies waaruit de macro bestaat, worden op elke plaats in het programma tussengevoegd, waar de macro wordt aangeroepen. Dit betekent dat het object program niet korter wordt door het toepassen van macro's. De executietijd neemt niet toe, omdat er niet naar de macro hoeft te worden gesprongen. De macro is immers op de gewenste plaatsen in het programma opgenomen. Indien een macro meerdere malen voorkomt in een programma, wordt het source program wel korter. De macro hoeft immers slechts één keer te worden geschreven.

Fig. 8.6 demonstreert het verschil tussen het gebruik van een subroutine en een macro.

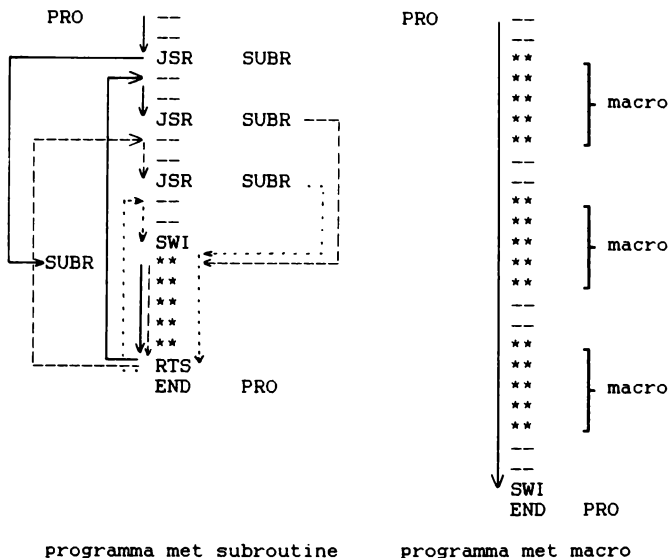


Fig. 8.6 Programmafow bij gebruik van subroutine en macro

Het gebruik van macro's

Om macro's te kunnen definiëren, heeft u een assembler nodig die dit toelaat. Zo'n assembler heet macro-assembler (zie ook paragraaf 5.5).

Een groep instructies moet worden voorzien van een naam. De programmeur kan vervolgens volstaan met het opgeven van die macronaam in plaats van de groep instructies.

Een macrodefinitie bestaat uit drie onderdelen:

1. de macro header, die de naam van de macro definieert.
2. de body van de macro; hierin staan de instructies die door de macronaam vervangen worden.
3. de pseudo-instructie die het einde van de macrodefinitie aangeeft.

Voorbeeld 8.1

De subroutine WACHT kunnen we als volgt definiëren als een macro:

```
10 WACHT    MACR
20          LDD    #13
30 DLYLUS   SUBD    #1
40          BNE    DLYLUS
50          ENDM
```

Regel 10 is de macro header. De regels 20, 30 en 40 vormen de macro body, terwijl regel 50 het einde van de macrodefinitie aangeeft.

De macro call bestaat eenvoudigweg uit de macronaam WACHT, die in het opcodeveld geplaatst wordt:

```
PRGRAM      --
             --
             WACHT          macro call
             --
             --
             SWI
             END    PRGRAM
```

Het vervangen van de macronaam door de macro body (tijdens het assembleren) heet macro expansion.

Merk nogmaals op dat een subroutine call een machine-instructie is, terwijl een macro call een signaal voor de assembler is om een macronaam door de overeenkomstige macro body te vervangen.

Formele en actuele parameters

Soms zijn er in een programma opeenvolgingen van instructies te onderkennen die vrijwel gelijk zijn, maar niet helemaal. Zulke groepen instructies lenen zich bij uitstek ervoor om door macro's te worden vervangen. Hiertoe moet de macro body formele parameters kunnen bevatten en de macro call actuele parameters. Bij de macro expansion worden de formele parameters vervangen door de bijbehorende actuele parameters.

De actuele parameters worden geplaatst in het operand veld van de macro call, onderling gescheiden door het teken <:> (punt-komma).

De formele parameters komen in de macro body voor als een symbool, voorafgegaan door een speciaal teken. In sommige assemblers wordt dit gerealiseerd met <:n>. Hierin is <:> (dubbele punt) het speciale teken om aan te geven dat er een formele parameter volgt, terwijl <n> de formele parameter zelf voorstelt. <n> kan bijvoorbeeld een van de cijfers 0 t/m 9 zijn. In dat geval kunnen er per macrodefinitie maximaal 10 formele parameters voorkomen. De eerste actuele parameter komt overeen met <:0>, de tweede met <:1>, enzovoorts. Realiseer u dat dit slechts een voorbeeld is. Bij de diverse macro-assemblers kunnen verschillende methoden gekozen zijn voor de

parametertoewijzing.

De formele parameters mogen in willekeurige volgorde voorkomen in de macrodefinitie. Ze mogen zelfs meerdere keren worden gebruikt. De actuele parameters moeten in de juiste volgorde bij de macro call worden opgesomd. Ze moeten hier staan in de volgorde die overeenkomt met de numerieke volgorde van de gebruikte formele parameters.

Voorbeeld 8.2

Veronderstel dat we het volgende programma hebben:

```
100      ORG      $4000
110 TAAK      --
120      TFR      DP,A          laad pointer Y met
130      CLRB                      het eerste adres van de
140      TFR      D,Y          huidige geheugenpagina
150      LDA      GETAL        bereken het 2's complement
160      LDB      CIJFER        produkt van
170      LBSR      MUL2'S      GETAL en CIJFER
180      ADDD      CONST      tel CONST op bij het produkt en
190      STD      4,Y          zet de uitkomst op de 5de en
200 *                      6de locatie van de huidige
210 *                      geheugenpagina
220      --
230      --
240      TFR      DP,A          laad pointer U met
250      CLRB                      het eerste adres van de
260      TFR      D,U          huidige geheugenpagina
270      LDA      GETAL        bereken het 2's complement
280      LDB      CIJFER        produkt van
290      LBSR      MUL2'S      GETAL en CIJFER
300      SUBD      CONST      verminder het produkt met CONST
310      STD      4,U          en zet de uitkomst op de 5de
320 *                      en 6de locatie van de
330 *                      huidige geheugenpagina
340      --
350      --
360      SWI
370      END      TAAK
```

De regels 120 t/m 190 lijken veel op de regels 240 t/m 310. Ze zijn echter niet helemaal gelijk, zodat we ze niet kunnen vervangen door dezelfde subroutine. In de regels 140 en 190 komt Y voor, in de regels 260 en 310 daarentegen U. In regel 180 treffen we de instructie ADDD aan, in regel 300 SUBD. Wel kunnen de beide groepen instructies worden vervangen door macro calls met enkele parameters. De macrodefinitie is identiek voor beide groepen instructies.

```
100      ORG      $4000
110 REKEN      MACR          macro header van de macro REKEN
120      TFR      DP,A          laad pointer <:0> met
```

130	CLRB		het eerste adres van de
140	TFR	D.:0	huidige geheugenpagina
150	LDA	GETAL	bereken het 2's complement
160	LDB	CIJFER	produkt van
170	LBSR	MUL2'S	GETAL en CIJFER
180	:1	CONST	verhoog of verminder het produkt
190	STD	4.:0	met CONST en zet de uitkomst
200 *			in de tabel, aangewezen door
210 *			pointer <:0>
220	ENDM		einde van de macrodefinitie
230 TAAK	--		
240	REKEN	Y;ADDD	pointer is Y; verhoog met CONST
250	--		
260	--		
270	REKEN	U;SUBD	pointer is U; verlaag met CONST
280	--		
290	--		
300	SWI		
310	END	TAAK	

U ziet dat het source program korter is geworden. Het object program wordt daarentegen niet korter bij het toepassen van macro's. Bij het assembleren van het programma met de macro calls ontstaat precies hetzelfde object program als wanneer de versie van het programma zonder macro's zou zijn geassembleerd.

In de macro body mogen subroutine calls en andere macro calls voorkomen.

Voordelen van macro's

1. Een programma wordt niet trager bij gebruik van macro's. Er hoeft immers niet naar de macro's te worden gesprongen. Gebruik derhalve macro's in plaats van subroutines, indien zeer hoge executiesnelheden zijn vereist.
2. Indien een programma onderdelen bevat die bijna gelijk aan elkaar zijn, wordt het sourceprogramma door gebruik van macro's korter.

Subroutines kunnen alleen gelijke programmadelen vervangen! Het gebruik van macro's met parameters kan in dit geval veel werk besparen bij het ontwikkelen van programma's. Ten gevolge van deze parameters wordt de doorzichtigheid van de programma's kleiner. Extra commentaar is hierbij zeker gewenst.

Nadelen van macro's

1. Het objectprogramma wordt niet korter bij gebruik van macro's (ook niet langer).
2. Er zijn meer restricties bij het gebruik van macro's dan bij het gebruik van subroutines.
3. Macro's met formele parameters maken een programma minder goed leesbaar.

Restricties bij het gebruik van macro's

1. De macronaam mag geen opcode of assembler directive zijn.
2. De macrodefinitie moet in het sourceprogramma vóór de macro call voorkomen (bij een two-pass macro-assembler). Tijdens de eerste assembleerstap wordt de macro body op de plaats van de macro call ingevuld. Tevens worden eventuele labels in adressen vertaald en opgenomen in een symbol table (zie ook paragraaf 5.5). Deze symbol table wordt tijdens de tweede assembleerstap gebruikt om de juiste adressen in te vullen bij het vertalen van de instructies. Zou de macrodefinitie na de macro call voorkomen, dan kan de symbol table tijdens de eerste assembleerstap niet volledig worden ingevuld, omdat de lengte van de macro nog niet bekend is. Er zou dan een tweede assembleerstap nodig zijn om de symbol table te completeren en een derde stap voor het vertalen van de instructies.
3. Een macrodefinitie mag niet worden genest. Het heeft geen zin om binnen een macrodefinitie een andere macrodefinitie op te nemen. Binnen een macrodefinitie mag gewoon een call van een andere macro voorkomen, mits de bijbehorende macro van tevoren is gedefinieerd. Een macro wordt tijdens de eerste assembleerstap vervangen door de bijbehorende macro body. De macrodefinitie, die zich hierin zou bevinden, moet niet door de assembler worden opgenomen in het programma, maar moet worden bewaard om te worden ingevuld op de plaats van de call van deze binnenste macro.
4. Een macrodefinitie mag geen macro call van zichzelf bevatten.

Op de eerste plaats kan de macro call niet worden vervangen door de macro body, omdat de macro nog niet volledig is gedefinieerd ter plaatse van de call. Op de tweede plaats zou het proces van het telkens vervangen van de macro call door de macro body (waarin wederom de call voorkomt), nooit eindigen.

5. Wordt een macro meer dan één keer aangeroepen, dan moeten de eventuele labels in de macro body worden vervangen door formele parameters. Deze worden tijdens de macro expansion vervangen door de bijbehorende actuele parameters.

Zou dit niet gebeuren, dan ontstaan er meervoudig voorkomende labels. Tijdens de eerste assembleerstap wordt de macro call in het sourceprogramma steeds door de macro body vervangen. Zodoende komt eenzelfde label op diverse plaatsen voor. Dit wordt voorkomen door een formele parameter als label te gebruiken. Bij elke macro call moet deze formele parameter door een andere actuele parameter worden vervangen. Voorbeeld 8.3 laat zien hoe dit in zijn werk gaat in het geval van de macro WACHT.

Voorbeeld 8.3

```
10 WACHT    MACR
20          LDD      #13
30 :0       SUBD     #1
40          BNE      :0
50          ENDM
```

```

PROGRAM      --
--
WACHT  LUS1      macro call
--
--
WACHT  LUS2      macro call
--
--
WACHT  LUS3      macro call
--
--
SWI
END      PROGRAM

```

8.5 Parameteroverdracht

8.5.1 Inleiding

In het algemeen moeten er bij modules, subroutines en macro's bepaalde gegevens in- en uitgevoerd worden. Bekijkt u nogmaals het programma TAAK, dan ziet u dat de invoergegevens voor de subroutine MUL2'S (de two's complement getallen GETAL en CIJFER) in de accumulatoren A en B staan, terwijl het uitvoergegeven van MUL2'S (het produkt) in accumulator D komt te staan. Dit overbrengen van gegevens tussen de verschillende modules wordt aangeduid met parameteroverdracht. Het kan hierbij gaan om invoer van gegevens, of uitvoer of beide. Bij de subroutine WAIT ontbreekt de parametertransfer helemaal. Deze routine heeft geen gegevens van buiten nodig en doet evenmin gegevens ontstaan, die voor andere programmadelen van belang zijn. Het effect van deze subroutine is een vertraging van ongeveer 100 μ s. Veelal moeten in programma's vertragingen van verschillende grootte worden opgenomen. Het kost dan erg veel tijd en geheugenruimte om voor al die vertragingen aparte subroutines te ontwerpen. Een macro met parameters zou het objectprogrammaodeloos lang maken. Eenvoudiger is het om een subroutine te ontwikkelen met een vertraging, die afhankelijk is van een bepaald invoergegeven. Hebben we een wachttijd nodig van een variabel aantal ms, dan kunnen we dit aantal aan de subroutine DMSEC meedelen via bijvoorbeeld accumulator D. De subroutine DMSEC kan er dan als volgt uitzien, aangenomen dat de duur van een klokcyclus 1 μ s bedraagt:

10	DMSEC	PSHS	CC,X	(8 μ s) red (CC) en (X)
20	DKEER	LDX	#90	(3 μ s)
30	LUS1M	LEAX	>-1,X	(8 μ s) verlaag (X) met 1
40		BNE	LUS1M	(3 μ s)
50		SUBD	#1	(4 μ s) spring als lus nog geen
60		BNE	DKEER	(3 μ s) (D) keer is uitgevoerd

70	PULS	CC,X	(8 μ s) herstel (CC) en (X)
80	RTS		(5 μ s) keer terug naar het
90 *			aanroepende programma

De kleine lus, bestaande uit de regels 30 en 40, wordt telkens 90 keer achter elkaar doorlopen. Dit heeft een vertraging van $90 \cdot 11 \mu\text{s} = 990 \mu\text{s}$ tot gevolg. De grote lus, bestaande uit de regels 20 t/m 60 wordt zo vaak doorlopen als het getal in D aangeeft. Een keer doorlopen van deze lus levert een wachttijd op van $3 + 990 + 4 + 3 \mu\text{s} = 1000 \mu\text{s} = 1 \text{ ms}$. Totaal ontstaat hierbij dus een wachttijd van (D) ms. Daar komt echter nog bij de executietijd van de PSHS- en PULS-instructies en de executietijd van de subroutine call en RTS. Bij gebruik van de extended addressing instructie JSR is dit samen $8 + 8 + 8 + 5 \mu\text{s} = 29 \mu\text{s}$. Dit is ten opzichte van 1 ms te verwaarlozen.

Bij de PSHS- en PULS-instructies wordt de inhoud van D niet gered. Dit hoeft waarschijnlijk ook niet, omdat (D) aangeeft hoeveel ms er moet worden gewacht. Deze informatie zal na de wachttijd niet meer nodig zijn. U wordt wel aangeraden in de documentatie van de subroutine te vermelden, dat (D)=0 wordt gemaakt door DMSEC.

Als (D)=0 is, wordt er $2^{16} = 65.536 \text{ ms}$ gewacht. Bij regel 50 wordt (D) eerst verlaagd, alvorens er bij regel 60 wordt gekeken of (D) gelijk aan 0 is. Is de beginwaarde van (D) gelijk aan 0, dan wordt (D)=\$FFFF wanneer regel 50 de eerste keer wordt uitgevoerd. Daarna moet (D) nog 65.535 keer met 1 worden verlaagd alvorens (D)=0 wordt en er niet meer wordt teruggesprongen naar DKEER.

Bij regel 30 is de operand gelijk aan $>-1,X$. Het teken ">" dient hier om ervoor te zorgen dat de assembler het getal -1 in 2 bytes opslaat in het programmeergeheugen. Dit is gedaan om een grotere executietijd te verkrijgen. Bij weglaten van het teken ">" zou de offset zijn opgenomen in de postbyte en zou de instructie 5 μs in plaats van 8 μs geduurd hebben.

Een programma kan als volgt gebruik maken van de subroutine DMSEC:

```
--
LDD    #35          wacht
BSR    DMSEC        35 ms
--
--
LDD    #5000        wacht
BSR    DMSEC        5 s
--
```

WAIT is een subroutine zonder parameters. DMSEC is een subroutine, die alleen een inputparameter heeft. Er zijn ook

modules die alleen outputparameters hebben. Denk hierbij bijvoorbeeld aan initialisatieroutines, die ervoor zorgen dat een aantal registers en/of geheugenlocaties worden voorzien van de juiste (constante) beginwaarden.

Voorbeeld 8.4

Subroutine CLEAR maakt de inhoud van alle registers 0, behalve de stack pointer S en de program counter PC. Er worden geen gegevens ingevoerd in de subroutine, maar als uitvoer dienen de registers A, B, CC, DP, X, Y en U. De waarden die deze registers in CLEAR hebben gekregen, worden door het aanroepende programma gebruikt.

```

10 CLEAR    CLRA
20          CLRB
30          TFR    A,CC
40          TFR    A,DP
50          TFR    D,X
60          TFR    D,Y
70          TFR    D,U
80          RTS

```

Voorbeeld 8.5

De subroutine MULTWO heeft zowel input- als outputparameters. De twee two's complement getallen die ingevoerd worden, moeten in A en B staan, het two's complement produkt wordt uitgevoerd via register D. MULTWO zou als volgt afgeleid kunnen worden.

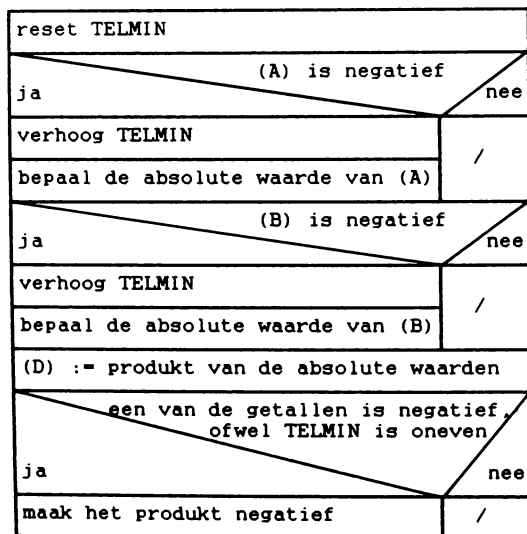


Fig. 8.7 Structogram voor een two's complement vermenigvuldiging

De instructie MUL bepaalt het produkt van twee unsigned numbers. Om het produkt van twee two's complement numbers te berekenen, kunnen we nagaan welk teken de uitkomst zal krijgen, vervolgens de absolute waarden van de twee getallen bepalen, hiervan het produkt berekenen en tenslotte het juiste teken toevoegen (zie fig. 8.7).

Een produkt van een aantal getallen is negatief als een oneven aantal van die getallen negatief is. Met teller TELMIN bepalen we hoeveel getallen negatief zijn.

TELMIN is oneven als bit 0 gelijk aan 1 is. Bit 0 heeft gewicht 1, terwijl de andere bits (bit 1 t/m bit 7) een macht van twee als gewicht hebben, dus een even gewicht. Dit betekent dat alleen bit 0 een oneven gewicht heeft en een getal derhalve even of oneven kan maken.

Het testen van bit 0 gaat eenvoudig door bit 0 in de carry te schuiven en vervolgens de carry bit te testen.

```

100 *****
110 * Subroutine MULTWO *
120 *****
130 *
140 *Progr.: J.G. Rouland
150 *Revisie 0      Datum: 20-04-1986
160 *
170 *MULTWO berekent het produkt van twee two's complement
180 *getallen, die zich bevinden in de accumulatoren A en B.
190 *Het produkt komt in accumulator D te staan.
200 *
210 *
220 TELMIN   RMB      1           reserveer ruimte voor TELMIN
230 *
240 MULTWO   CLR      TELMIN      TELMIN telt negatieve getallen
250         TSTA                      verhoog TELMIN
260         BPL      APOS          als (A) negatief is
270         INC      TELMIN        en bepaal de absolute
280         NEGA                      waarde van (A)
290 APOS     TSTB                      doe hetzelfde voor (B)
300         BPL      POSB
310         INC      TELMIN
320         NEGB
330 POSB     MUL                      vermenigvuldig absolute waarden
340         LSR      TELMIN        maak het produkt
350         BCC      POSIT        negatief, indien
360         COMA                      een van beide
370         COMB                      getallen negatief is
380         ADDD     #1            (ofwel als TELMIN oneven is)
390 POSIT    RTS

```

De regels 360, 370 en 380 zorgen ervoor dat het produkt negatief wordt. Aftrekken van 0 is immers hetzelfde als inverteren en 1 erbij tellen (zie paragraaf 2.10). Het condition code register wordt niet gered, omdat in dit

geval ook het condition code register dienst kan doen bij de parametertransfer. Het aanroepende programma kan aan de flags N en Z onmiddellijk zien of het produkt negatief of positief is en al dan niet gelijk aan 0.

8.5.2 Methoden van parameteroverdracht

In dit hoofdstuk heeft de parameteroverdracht steeds plaatsgevonden via registers. Er zijn echter meerdere mogelijkheden.

Parameteroverdracht kan geschieden via

1. registers
 2. vaste geheugenlocaties
 3. de stack
 4. een pointer
- De pointer kan zijn:
- a. een 16-bits register
 - b. het direct page register
 - c. 2 vaste geheugenlocaties
 - d. 2 stacklocaties

8.5.3 Parameteroverdracht via registers

De parameteroverdracht via registers is de meest eenvoudige, de meest directe en de snelste manier. Het nadeel is echter dat het aantal registers zeer beperkt is. Zijn er veel parameters over te dragen, dan moeten we onze toevlucht zoeken tot een van de overige drie methoden.

Het condition code register leent zich bij uitstek voor het uitvoeren van een zogenaamde return code. De return code geeft informatie over de afloop van de routine. Er wordt als het ware een status overgedragen. De return code verschaft informatie over het al of niet optreden van een fout, het al of niet succesvol zijn van een zoekactie, het groter, kleiner of gelijk zijn, het positief of negatief zijn, enzovoorts. Elk van deze toestanden kan door een afzonderlijke bit van het condition code register worden weergegeven.

Voor al de bits N, Z, V en C zijn hiervoor zeer geschikt, omdat ze met behulp van conditionele spronginstructies elk afzonderlijk kunnen worden getest.

Flag H daarentegen moet eerst met een bittest-instructie worden getest.

De bits E, F en I kunnen beter niet worden gebruikt, omdat ze een speciale betekenis hebben met betrekking tot het gedrag van de microprocessor bij het optreden van interrupts. Dit wordt in hoofdstuk 9 behandeld.

Voorbeeld 8.6

Er moet een subroutine worden geschreven, die nagaat of een 2-bytes getal al dan niet kleiner is dan 20.000. Als de tekenbit van dat getal 1 is, moet er een foutmelding gegeven worden. Het getal wordt ingevoerd via accumulator D. De foutmelding geven we weer via de flag N (1=fout) en het kleiner zijn geven we aan met C=1. C is alleen van belang als er geen fout gevonden is, dus als N=0.

5		ORG	\$3800	
10	CHECK	TSTA		set N als tekenbit van D gelijk
20		BPL	TEKENO	aan 1 is en keer dan terug
30		BRA	ENDSUB	naar aanroepende programma
40	TEKENO	CMPD	#20000	set C als (D)<20.000
50		ANDCC	##11110111	reset N
60	ENDSUB	RTS		terug naar aanroepende progr.
70		*		
80		*		
90	PROEF	--		
100		--		
110		BSR	CHECK	
120		BMI	FOUT	spring als een fout is gevonden
130		BCS	KLEIN	spring als getal < 20.000
140		--		vervolg als getal ≥ 20.000
150		--		
160		--		
170		BRA	ENDPRO	
180	KLEIN	--		vervolg als getal < 20.000
190		--		
200		--		
210		BRA	ENDPRO	
220	FOUT	--		vervolg bij fout
230		--		
240		--		
250	ENDPRO	SWI		stop
260		END	PROEF	

Regel 30 kan vervangen worden door de instructie RTS en de regels 170 en 210 door SWI. We hebben hier toch spronginstructies toegepast om te bereiken, dat zowel de subroutine als het hoofdprogramma slechts één uitgang heeft. Dit maakt het programma overzichtelijker en bovendien is dit gemakkelijker bij het testen. Indien we aan het einde van een module willen nagaan of bijvoorbeeld de registers de juiste informatie bevatten, kan dit geschieden door het programma te laten stoppen bij de uitgang van de module. Indien een module meerdere uitgangen heeft, wordt dit moeilijker, omdat de module niet altijd via dezelfde uitgang wordt verlaten.

De testinstructie TSTA (regel 10) kopieert de tekenbit van A, oftewel die van D, in de flag N. Hiermee is automatisch de foutmelding gerealiseerd.

De compare-instructie (regel 40) set de C-bit als (D) < 20.000

en reset C in het andere geval. Hiermee is de conditie kleiner dan op de gewenste wijze vastgelegd.
 Ten gevolge van de compare instructie kan de N-bit geset worden. Er moet echter gelden N=0, omdat er geen foutconditie is. Derhalve moet de N-bit gereset worden met de ANDCC-instructie (regel 50).

8.5.4 Parameteroverdracht via vaste geheugenlocaties

Parameteroverdracht via vaste geheugenplaatsen ligt voor de hand als het om grotere aantallen gegevens gaat. Het werkt erg eenvoudig. Het aanroepende programma zet de invoerparameters op bepaalde geheugenplaatsen. De routine haalt ze daar vandaan voor verwerking en zet de uitvoerparameters op bepaalde geheugenplaatsen. Het aanroepende programma kan de resultaten vervolgens op die plaatsen terugvinden.

Voorbeeld 8.7

Een programma zet vier 1-byte getallen op vaste geheugenplaatsen. De subroutine ADD4 berekent zowel de unsigned som als de two's complement som van deze vier getallen en bergt de resultaten op in vier andere geheugenlocaties. Realiseer u dat er voor de som 2 bytes nodig zijn. Voor het uitvoeren van de optelling wordt de macro SOM gebruikt.

5	ORG	\$4000	
10	GETAL1	RMB 1	reserveer
20	GETAL2	RMB 1	ruimte
30	GETAL3	RMB 1	voor de 4
40	GETAL4	RMB 1	getallen
50	SOMUNS	RMB 2	ruimte voor de unsigned som
60	SOMTWO	RMB 2	ruimte voor two's compl. som
70	*		
80	SOM	MACR	macro header
90	ABX		verhoog unsigned som met getal
100	SEX		verhoog
110	ADDD	SOMTWO	SOMTWO met
120	STD	SOMTWO	getal
130	ENDM		einde van macrodefinitie
140	*		
150	ADD4	PSHS CC,D,X	red registers
160	CLRA		
170	CLRB		
180	STD	SOMTWO	reset SOMTWO
190	TFR	D,X	reset unsigned som
200	LDB	GETAL1	
210	SOM		
220	LDB	GETAL2	
230	SOM		
240	LDB	GETAL3	

250	SOM		
260	LDB	GETAL4	
270	SOM		
280	STX	SOMUNS	berg unsigned som op
290	PULS	CC,D,X	herstel registers
300	RTS		terug naar aanroepende progr.
310	*		
320	PRO	--	begin van programma
330	--		

Natuurlijk kunnen we de macro SOM ook als subroutine definiëren. Het object program wordt hierdoor korter. Ook kunnen we het optellen in een programmalus doen en de instructies van SOM in deze lus opnemen.

8.5.5 Parameteroverdracht via een pointer

Vaak is het handig de parameteroverdracht te realiseren via een parameterblok. Alleen het beginadres hoeft in dit geval aan de routine te worden overgedragen via een pointer.

Voorbeeld 8.8

Stel we willen een subroutine TELOP schrijven, die een aantal unsigned getallen optelt. Er moet een melding gegenereerd worden, die aangeeft of de som al dan niet in één byte past. Het aanroepende programma zorgt ervoor, dat pointer U wijst naar het geheugenblok waar de getallen staan. De indeling van het geheugenblok is als volgt:
De eerste byte geeft het aantal op te tellen getallen weer. Daarna volgen de getallen zelf. Vervolgens is er een locatie gereserveerd voor de melding (\$00=fout, \$FF=goed). Daarachter volgt de som, mits deze in één byte past (zie fig. 8.8). Aangenomen mag worden dat het aantal op te tellen getallen tenminste 1 bedraagt. Het structogram van de oplossing is gegeven in fig. 8.9.

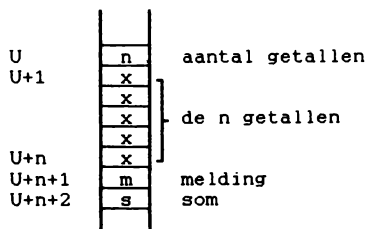


Fig. 8.8 Parameterblok van de subroutine TELOP



Fig. 8.9 Structogram bij het optellen van unsigned numbers

```

10 *****
20 * Subroutine TELOP *
30 *****
40 *
50 *Progr.: J.G. Rouland
60 *Revisie 0      Datum: 20-04-1986
70 *
80 *TELOP telt n 1-byte unsigned getallen op.
90 *Het aantal n staat in de geheugenlocatie, die wordt
100 *aangewezen door pointer U. Na n volgen de getallen.
110 *Daarachter komt de melding $FF te staan, indien de som in
120 *één byte past, gevolgd door de som zelf. Past de som niet
130 *in één byte, dan verschijnt slechts de melding $00.
140 *
150 *De subroutine laat de registerinhouden ongewijzigd.
160 *
170 *
180 TELOP   PSHS   CC,D,X      red registers
190         CLRA                som:=0
200         LDB    ,U          lees aantal
210         LEAX   1,U          pointer wijst naar eerste getal
220 ERBIJ   ADDA   ,X+         tel getal bij som, verhoog pointer
230         DECB                aantal := aantal - 1
240         BEQ    TESTC        spring als alle getallen opgeteld
250         BCC    ERBIJ        spring als som nog in 1 byte past
260 TESTC   BCS    FOUT         spring als som niet in 1 byte past
270         LDB    #$FF        geef de melding $FF = "goed"
280         STB    ,X+          en verhoog de pointer

```

290	STA	,X	berg de som op
300	BRA	HERSTL	
310	FOUT	,U	
320	INCB		
330	CLR	B,U	geef de melding \$00 = "fout"
340	HERSTL	PULS	herstel registers
350	RTS		keer terug

Bedenk bij regel 250 en 260 dat de voorgaande instructie DECB (regel 230) de carry niet verandert, zodat bij regel 250 en 260 inderdaad de carry van de optelling wordt getest.

De melding "goed" (=\$FF) kan in regel 280 meteen opgeborgen worden op de daarvoor bestemde geheugenplaats, omdat X op dat moment het adres van die geheugenplaats bevat. Door X te verhogen, wijst X vervolgens naar de locatie waar de som moet komen te staan, zodat deze meteen in de volgende regel kan worden opgeborgen op de juiste locatie.

Als de som niet meer in één byte past, hoeven nog niet alle getallen te zijn opgeteld. Dit betekent dat X waarschijnlijk niet naar de locatie voor de melding wijst. Deze locatie moet derhalve eerst worden berekend (regel 310, 320 en 330) alvorens de foutmelding (=\$00) kan worden opgeborgen.

U hebt nu kennis gemaakt met parameteroverdracht via registers, via vaste geheugenplaatsen en via een pointer, waarbij de pointer een register is. De overige methoden zullen worden behandeld in hoofdstuk 12.

8.6 Documentatie

Het is van groot belang de diverse modules te voorzien van voldoende documentatie. Wat in paragraaf 5.3.5 is gezegd over commentaar, geldt onverkort voor de modules, waaruit een programma is opgebouwd.

In de documentatie van een module horen de volgende zaken opgenomen te zijn:

- de naam van de module
- de naam van de ontwerper
- het nummer van de revisie en de datum van vrijgave
- een beschrijving van de functie van de module
- een lijst van alle invoer- en uitvoerparameters
- een lijst van geheugenplaatsen en registers die door de module veranderd worden
- een concreet voorbeeld kan erg verhelderend werken

Voorbeeld 8.9 (RAM opsporen)

We willen een subroutine ontwerpen om op te sporen in welk geheugengebied RAM aanwezig is. Dit kan gebeuren door op een

aantal adressen te schrijven en vervolgens na te gaan of de geschreven informatie inderdaad in het geheugen aanwezig is. Op plaatsen waar zich geen geheugen bevindt, zal niet de juiste informatie worden teruggelezen. Dit geldt ook voor locaties waar ROM (Read Only Memory) aanwezig is.

Het geheugen bestaat uit verschillende geheugen-IC's. Aangenomen wordt dat elk geheugen-IC tenminste 1 kbyte bevat. Om tijd te sparen testen we niet elk adres, maar volstaan we met slechts de eerste en de laatste locatie van elke pagina (van 256 bytes) te testen. Hiermee wordt de kans dat zich juist daar I/O-adressen bevinden, vrij klein. I/O-registers zijn vaak read/write registers, die als zodanig niet van RAM zijn te onderscheiden.

Om te voorkomen dat we ROM voor RAM aanzien, doordat er toevallig de testinformatie in staat, schrijven we op elke geheugenlocatie twee verschillende bitpatronen, bijvoorbeeld \$AA en het inverse patroon \$55.

Natuurlijk moeten we voorkomen dat de page getest wordt waarin de subroutine staat, anders zou deze routine tijdens het testen vernield kunnen worden. We gaan er van uit dat het testprogramma zich binnen één pagina bevindt.

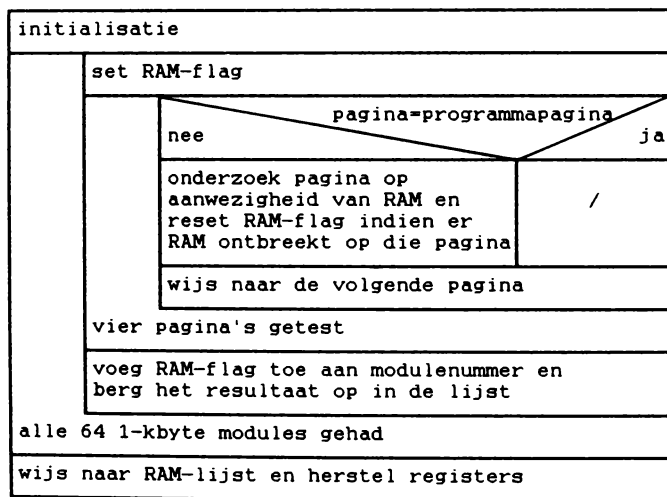


Fig. 8.10 Structogram voor RAM-opsporingsroutine

Direct achter de subroutine volgt een lijst met de nummers van de 64 1kbyte geheugenmodules, waarbij bit 7 aangeeft of de module al dan niet uit RAM bestaat (1-RAM, 0-geen RAM). Pointer U moet aan het einde van de subroutine het beginadres van deze lijst bevatten.

Elke 1-kbyte geheugenmodule bestaat uit 4 pagina's ($4 \cdot 256 = 1k$). Fig. 8.10 toont het structogram voor de oplossing van het probleem.

De subroutine ziet er als volgt uit:

```

10 *****
20 * Subroutine RAMMAP *
30 *****
40 *
50 *Progr.: J.G. Rouland
60 *Revisie 0      Datum: 20-04-1986
70 *
80 *RAMMAP genereert een lijst van nummers (0 t/m 63) van
90 *1-kbyte geheugenmodules, waarbij bit 7 = 1 aangeeft dat de
100 *betreffende module uit RAM bestaat; bit 7 = 0 betekent dat
110 *de betreffende module niet uit RAM bestaat.
120 *
130 *X bevat het beginadres van de lijst
140 *
150 *Het gehele testprogramma, inclusief de lijst, moet zich
160 *binnen één geheugenpagina (van 256 bytes) bevinden.
170 *
180 *RAMMAP verandert, behalve X, geen andere registers. Wel
190 *wordt op elke eerste en laatste geheugenlocatie van elke
200 *pagina $55 geschreven.
210 *
220 *
230      ORG      $3000      testprogr. staat op pagina $30
240 RAMBIT  RMB      1      ruimte voor RAM-flag
250 PROPAG  RMB      1      ruimte voor paginanummer
260 *                                van testprogramma
270 LIJST   RMB      64      ruimte voor modulenummers met
280 *                                de mededeling of er RAM is
290 *
300 *Macro TEST
310 *
320 *TEST gaat na of er in locatie <:1> van de huidige geheugen-
330 *pagina kan worden geschreven. Als testpatroon wordt <:0>
340 *gebruikt. Indien er niet geschreven kan worden, wordt de
350 *RAM-flag gereset.
360 *
370 TEST    MACR                      macro header
380        LDB      #:0              laad testpatroon
390        STB      <:1              test
400        CMPB     <:1              geheugenlocatie
410        BEQ      GOED              en reset
420 FOUT    CLR      RAMBIT          RAM-flag bij schrijffout
430 GOED     NOP
440        ENDM                      einde van macrodefinitie
450 *
460 RAMMAP   PSHS     CC,DP,D         red registers
470        LDU      #LIJST           initialiseer lijstpointer

```


480	LDD	#RAMBIT	berg paginanummer van
490	STA	PROPAG	testprogr. op in PROPAG
500	CLRA		begin bij
510	TFR	A,DP	pagina 0
520	SETRAM	LDB #80	set
530	STB	RAMBIT	RAM-flag
540	NXTPAG	CMPA PROPAG	sla de pagina van het
550	BEQ	INCRDP	testprogramma over
560	TEST	\$AA:\$00	test de eerste en de laatste
570	TEST	\$55:\$00	locatie van een pagina met de
580	TEST	\$AA:\$FF	testpatronen \$AA en \$55; reset
590	TEST	\$55:\$FF	de RAM-flag als er geen RAM is
600	INCRDP	INCA	DP wijst naar de
610	TFR	A,DP	volgende pagina
620	BITA	##00000011	spring indien nog geen vier
630	BNE	NXTPAG	pagina's getest zijn
640	LSRA		bepaal het nummer van
650	LSRA		van de vorige
660	DECA		1-kbyte module
670	ORA	RAMBIT	voeg RAM-flag=1 bij modulenummer
680	ZETWEG	STA ,U+	berg RAM-flag en nummer van
690	*		module op; verhoog lijstpointer
700	TFR	DP,A	herstel het nieuwe paginanummer
710	TSTA		spring als nog niet alle
720	BNE	SETRAM	1k-modules getest zijn
730	LDU	#LIJST	lijstpointer wijst naar begin lijst
740	PULS	CC,DP,D	herstel de registers
750	RTS		keer terug
760	*		
770	HOOFD	--	begin van hoofdprogramma
780	--		
790	SWI		stop
800	END	HOOFD	einde van het source program

Het nagaan of er een module getest is, gebeurt op de regels 620 en 630. Hierbij wordt gebruik gemaakt van het volgende. Elke 1-kbyte geheugenmodule bevat 4 pagina's. De eerste pagina van de eerste module heeft als nummer 0. Het nummer van de eerste pagina van elke volgende module is een veelvoud van 4. Een getal is een veelvoud van 4 als bit 0 en bit 1 beiden 0 zijn, want de bits 2 en hoger zijn allen veelvouden van 4. Met BITA ##0000 0011 wordt Z=1 als A0=A1=0, immers alleen dan is het resultaat van de and-functie gelijk aan 0. In dat geval wijst A naar de eerste pagina van de volgende module en wordt er (op regel 630) niet teruggesprongen naar NXTPAG.

Het nummer van de volgende te testen module is gelijk aan het vierde deel van het paginanummer. Het paginanummer is een unsigned getal, zodat delen door vier hetzelfde is als twee keer logisch naar rechts schuiven (zie regel 640 en 650). Het nummer van de vorige geteste module is 1 lager dan het nummer van de volgende te testen module (zie regel 660).

Het nummer van de module is hoogstens 63, zodat de bits 6 en 7 altijd 0 zijn. Bit 7 wordt gebruikt om aan te geven of de betreffende module RAM bevat. Doordat RAMBIT (RAM-flag) gelijk is aan \$00 (geen RAM) of \$80 (wel RAM), kan deze flag eenvoudig aan het modulenummer worden toegevoegd door te "OR-en" (zie regel 670). Hetzelfde effect kan worden bereikt met optellen.

Op regel 710 en 720 wordt gekeken of alle 256 pagina's getest zijn. Het hoogste paginanummer bedraagt \$FF. Is de inhoud van het direct page register gelijk aan \$00 (bij regel 700), dan hebben we alle pagina's getest. Daar de instructie TFR (regel 700) de bits van het condition code register niet wijzigt, is de instructie TSTA (regel 710) nodig, opdat Z=1 wordt als (A)=0 is (alle pagina's gehad). In dit geval moet de lijstpointer teruggezet worden op het begin van de lijst (regel 730).

Het uitvoeren van de RAM-test zelf gebeurt met behulp van de macro TEST (regel 560 t/m 590). In de macro wordt het testpatroon met behulp van direct addressing opgeborgen op een geheugenlocatie (zie regel 390) en vervolgens wordt hetzelfde testpatroon vergeleken met wat er op die geheugenlocatie staat (zie regel 400). Is dit gelijk, dan is er op dat adres read/write memory aanwezig.

De instructie NOP op regel 430 is nodig, omdat er niet gesprongen kan worden naar de assembler directive ENDM. Deze mag geen label bevatten, hetgeen ook logisch is, omdat een assembler directive niet wordt vertaald in een instructie en er mag alleen worden gesprongen naar een opcode van een instructie.

In het hoofdprogramma HOOFD kan eenvoudig worden getest of een bepaalde module uit RAM bestaat met behulp van de instructies

	JSR	RAMMAP	bepaal de pagina's met RAM
	--		
	--		
	TST	B,U	test de RAM-flag van module (B)
	BMI	RAMJA	spring als (B) een RAM module is
RAMNEE	--		
	--		
	--		
	BRA	VERDER	
RAMJA	--		
	--		
	--		
VERDER	--		
	--		

nits B het nummer van de te testen module bevat. Is er RAM aanwezig, dan wordt het programma voortgezet bij de label RAMJA. Is er geen RAM, dan wordt het programma vervolgd bij RAMNEE.

Op deze wijze worden er verschillende acties uitgevoerd bij het al dan niet uit RAM bestaan van een bepaalde geheugenmodule.

Voorbeeld 8.10 (Zoeken in een alfabetisch gesorteerde lijst)

We ontwerpen een subroutine, die een bepaald kenmerk opzoekt in een alfabetisch gesorteerde lijst.

Het hoofdprogramma zet het kenmerk in accumulator A en het beginadres van de lijst in register U. De eerste twee locaties in de lijst bevatten het aantal karakters van de lijst.

Indien het kenmerk gevonden wordt, moet Z=1 zijn, terwijl Y moet wijzen naar de locatie, waar het kenmerk zich bevindt. Komt het kenmerk niet in de lijst voor, dan moet Z=0 worden.

Een methode om dit probleem op te lossen, is vanaf het begin van de lijst elk karakter te vergelijken met het kenmerk, totdat het kenmerk gevonden is of totdat het einde van de lijst is bereikt. Bij lange lijsten kan het zoeken erg veel tijd in beslag nemen. Bij deze zoekprocedure hoeft de lijst niet gesorteerd te zijn.

Een snellere methode berust op het principe van "successive approximation" (opeenvolgende benadering). Hierbij wordt eerst gekeken of het kenmerk zich in de eerste of in de tweede helft van de lijst bevindt. Vervolgens wordt nagegaan of het kenmerk zich in het eerste of in het tweede kwart van de betreffende helft bevindt. Zo wordt doorgegaan met telkens halveren van het gebied, waarin het kenmerk zich kan bevinden. Een soortgelijke methode wordt gevolgd bij het opzoeken van een naam in een telefoonboek. Ook hier wordt ongeveer in het midden begonnen, en afhankelijk van het resultaat, een stuk naar voren of naar achteren verder gezocht.

Het gebruik van de indicatieflags FOUND en READY is handig om te bepalen wanneer de zoekactie beëindigd kan worden. Bovendien moet de flag FOUND worden doorgegeven aan het aanroepende programma. FOUND=1 geeft aan dat het gezochte kenmerk gevonden is. READY=1 geeft aan dat het zoekproces is voltooid.

Het adres van het te onderzoeken karakter wordt telkens gevonden door de offset te halveren en bij het laatste adres op te tellen of van het laatste adres af te trekken, afhankelijk van de uitkomst van de vergelijking van het kenmerk met het vorige karakter. De gehalveerde offset moet hierbij naar boven worden afgerond om te voorkomen dat bepaalde locaties in de lijst niet kunnen worden bereikt.

Stel we zoeken karakter <g> en het huidige adres is dat van karakter <h>, terwijl de nieuwe offset, 5, hiervan moet worden afgetrokken (zie fig. 8.11). Er wordt dan gewezen naar <c>. Vervolgens moet de halve offset, zijnde 2, worden opgeteld bij het adres. Dan wordt er gewezen naar <e>. Weer verhogen met de halve offset, nu 1, wijst naar <f>. De offset nogmaals halveren levert 0 op, zodat karakter <g> niet kan worden bereikt. Als de

offset 0,5 is geworden, moet niet worden afgerond, omdat anders het programma nooit eindigt. 0,5 wordt immers afgerond tot 1. Halveren levert weer 0,5 op. Dit zou opnieuw worden afgerond tot 1, enzovoorts.

	offset
a b c d e <u>f</u> g h i j	-5
a b <u>c</u> d e f g h i j	+2
a b c d e <u>f</u> g h i j	+1
a b c d e <u>f</u> g h i j	+0 einde zoekactie

Fig. 8.11 Zonder correct afronden van de offset zijn niet alle locaties te bereiken

Zoals uit fig. 8.12 blijkt, zijn bij juist afronden (naar boven) wel alle locaties in een lijst te bereiken.

	offset
a b c d e f g <u>h</u> i j	-5
a b <u>c</u> d e f g h i j	+3
a b c d e <u>f</u> g h i j	+2
a b c d e f g <u>h</u> i j	-1
a b c d e f g <u>h</u> i j	+0 einde zoekactie

Fig. 8.12 Bij correct afronden van de offset zijn alle locaties bereikbaar

Wijst de lijstpointer naar een bepaald adres, dan moet de inhoud hiervan eerst worden onderzocht, alvorens naar een volgend adres te verwijzen. Laten we de pointer aanvankelijk naar het beginadres van de lijst wijzen, dan moeten we deze locatie eerst onderzoeken, voordat we halverwege de lijst verder gaan. Anders komen we mogelijk niet meer op dit adres terecht (zie fig. 8.13).

	offset
<u>a</u> b c d	+2
a b <u>c</u> d	-1
a b <u>c</u> d	+0 einde zoekactie

Fig. 8.13 Er wordt niet meer teruggekeerd bij het beginadres

Plaatsen we de pointer vlak voor het eerste adres van de lijst karakters, dan lopen we het risico het laatste adres niet te bereiken (zie fig. 8.14).

	offset
. a b c d	+2
. a <u>b</u> c d	+1
. a b <u>c</u> d	+0 einde zoekactie

Fig. 8.14 Het laatste adres wordt niet bereikt als de pointer vóór de lijst begint

Het hiervoor geschetste probleem kunnen we eenvoudig oplossen door te beginnen met het testen van de eerste locatie en pas daarna halverwege de lijst te gaan zoeken.

	offset		offset
<u>a</u> b c d e	+3	<u>a</u> b c d e f ...	+5
a b c <u>d</u> e	+2	a b c d e <u>f</u> ...	-3
a b c d e _	-1	a b <u>c</u> d e f ...	-2
a b c d <u>e</u>	+0	<u>a</u> b c d e f ...	-1
		_ a b c d e f ...	+0

Fig. 8.15 Pointer kan buiten de lijst wijzen

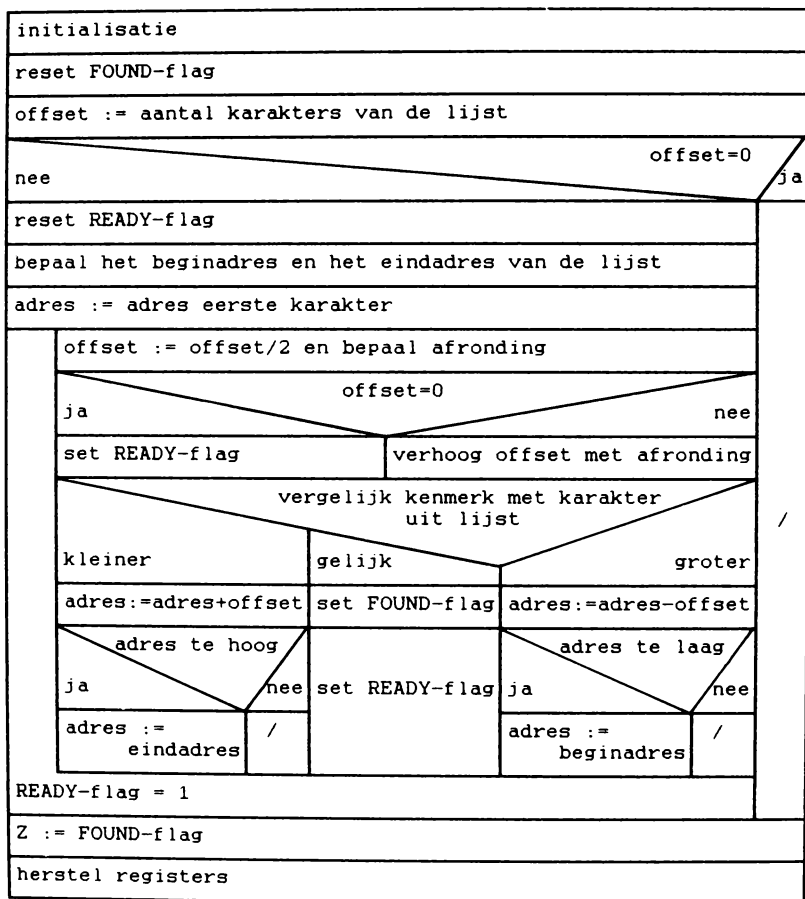


Fig. 8.16 Structogram voor zoeken in gesorteerde lijst

Een laatste probleem treedt op doordat we (ten gevolge van het afronden) buiten de lijst terecht kunnen komen (zie de voorbeelden van fig. 8.15). Derhalve moet hiervoor worden gewaakt en dient het adres te worden begrensd tot het eerste en laatste lijstadres.

Door Y als karakterpointer te gebruiken zal Y het adres van het gezochte karakter bevatten, zonder dat hiervoor extra instructies nodig zijn.

Het programma kan er nu als volgt uitzien:

```

100 *****
110 * Subroutine SZOEK *
120 *****
130 *
140 *Progr.: J.G. Rouland
150 *Revisie 0      Datum: 24-04-1986
160 *
170 *SZOEK is een snelzoekroutine die een kenmerk opspoort in
180 *een gesorteerde lijst karakters (in oplopende volgorde).
190 *
200 *Accumulator A moet het kenmerk bevatten.
210 *U bevat het beginadres van de lijst.
220 *De eerste twee locaties van de lijst bevatten het aantal
230 *karakters.
240 *
250 *Wordt het kenmerk niet gevonden, dan wordt Z=0 gemaakt.
260 *Komt het kenmerk wel in de lijst voor, dan wordt Z=1 en Y
270 *wijst naar de locatie in de lijst waar het kenmerk zich
280 *bevindt.
290 *
300 *SZOEK verandert alleen de inhoud van de registers CC en Y.
310 *
320 *
330 FOUND   RMB    1           reserveer ruimte voor diverse
340 READY   RMB    1           gegevens
350 KENMRK  RMB    1
360 OFFSET  RMB    2
370 FIRST   RMB    2
380 LAST    RMB    2
390 *
400 SZOEK   PSHS   D,U         red registers
410         STA    KENMRK      berg KENMRK op
420         CLR    FOUND
430         LDD     ,U++        laad aantal en wijs naar eerste
440 *                               karakter
450         STD     OFFSET      offset:=aantal karakters
460         BEQ     ZFLAG       spring als de lijst geen enkel
470 *                               karakter bevat
480 GADOOR   CLR    READY
490         LEAY    D,U         bereken het eindadres en
500         LEAY    -1,Y        berg dit op

```

510	STY	LAST	in het geheugen
520	TFR	U,Y	laad het beginadres en
530	STY	FIRST	berg het op in het geheugen
540 NEXT	LSR	OFFSET	offset := offset/2
550	ROR	OFFSET+1	
560	LDD	#0	berg de carry (afronding)
570	ADCB	#0	op in D
580	LDU	OFFSET	
590	BNE	ROUND	spring als offset ongelijk 0 is
600	INC	READY	set READY-flag
610	BRA	COMP	
620 ROUND	ADDD	OFFSET	verhoog offset met afronding
630	STD	OFFSET	
640 COMP	LDA	KENMRK	vergelijk kenmerk
650	CMPA	,Y	met karakter
660	BEQ	GELIJK	spring als kenmerk = karakter
670	BHI	HOGER	spring als kenmerk > karakter
680 LAGER	LDD	OFFSET	
690	LEAY	D,Y	adres := adres + offset
700	CMPLY	LAST	adres te hoog?
710	BLS	TEST	spring als adres niet te hoog
720	LDY	LAST	adres := hoogste adres in lijst
730	BRA	TEST	
740 GELIJK	INC	FOUND	(Y bevat het adres van het
750	INC	READY	gezochte karakter!)
760	BRA	TEST	
770 HOGER	TFR	Y,D	
780	SUBD	OFFSET	
790	TFR	D,Y	adres := adres - offset
800	CMPLY	FIRST	adres te laag?
810	BHS	TEST	spring als adres niet te laag
820	LDY	FIRST	adres := laagste adres in lijst
830 TEST	TST	READY	
840	BEQ	NEXT	spring als de READY-flag 0 is
850 ZFLAG	DEC	FOUND	Z := FOUND-flag
860	PULS	D,U	herstel registers
870	RTS		keer terug

In de regels 490 t/m 530 wordt het eerste en het laatste adres van de lijst berekend en opgeslagen in FIRST en LAST. Deze gegevens zijn verderop nodig bij het bewaken van de grenzen van de lijstpointer.

In de regels 540 en 570 wordt het gewicht 1/2 (=C) bewaard ten behoeve van de afronding van de offset. Dit moet gebeuren voordat de offset wordt vergeleken met 0, omdat hierbij de carry wordt gewijzigd! Bovendien moet het vergelijken van de offset met 0 plaatsvinden voordat afgerond wordt, omdat anders het programma nooit eindigt bij het niet voorkomen van het kenmerk. Immers 1/2 wordt weer afgerond op 1, zodat de afgeronde offset nooit lager wordt dan 1!

De instructie LDD #0 (regel 560) laat de carry ongemoeid, zodat

in regel 570 de carry wordt opgeteld, die is ontstaan bij het schuiven (regel 550).
Het afronden van de offset gebeurt door deze carry (= 1/2) als eenheid op te tellen bij de offset (regel 620 en 630).

Bij het vergelijken van alfabetisch gerangschikte karakters moeten deze worden gezien als getallen zonder teken. Derhalve moet de unsigned branch-instructie BHL worden toegepast (regel 670).

Adressen zijn unsigned numbers. Dit betekent dat in de regels 710 en 810 unsigned branch-instructies moeten worden gebruikt (BLS, respectievelijk BHS).

De FOUND-flag wordt aan het begin van het programma gereset (regel 420). Wordt het gezochte karakter gevonden, dan wordt de FOUND-flag geset (regel 740). FOUND is dan gelijk aan 1. In regel 850 wordt de Z-bit gelijk gemaakt aan de FOUND-flag. Dit werkt als volgt: Voor FOUND=0 (karakter niet gevonden) wordt FOUND na verlagen gelijk aan \$FF, zodat Z=0 wordt. Voor FOUND=1 (karakter gevonden) wordt FOUND gelijk aan \$00 na verlagen, zodat ook Z=1 wordt.

Voorbeeld 8.11 (Een snelle deelroutine)

Delen door de deler telkens van het deeltal af te trekken, is vrij traag, vooral als een groot getal wordt gedeeld door een klein getal. Een snellere manier is het delen door middel van herhaald schuiven en aftrekken. Dit werkt analoog aan een staartdeling, zoals we die op de basisschool hebben geleerd (zie het voorbeeld van fig. 8.17).

$$\begin{array}{r} 21 \ / \ 6465 \ \backslash \ 0307 \\ \underline{0} \\ 64 \\ \underline{63} \\ 16 \\ \underline{00} \\ 165 \\ \underline{147} \\ \text{rest } 18 \end{array}$$

Fig. 8.17 Staartdeling

Eerst wordt er nagegaan of de deler (een geheel aantal malen) af te trekken is van het linker cijfer van het deeltal. Zo ja, dan wordt een veelvoud van de deler van dit cijfer afgetrokken. Zo nee, dan wordt er 0 afgetrokken.

Vervolgens wordt het tweede cijfer van het deeltal aan de rechterzijde van het verkregen verschil toegevoegd en wordt er gecontroleerd of de deler hiervan (een aantal malen) is af te trekken. Het verschil van het vorige verschil en een veelvoud van de deler, respectievelijk 0, wordt uitgerekend en hieraan wordt het derde cijfer van het deeltal toegevoegd. Dit gaat zo door totdat alle cijfers van het deeltal aan de beurt zijn

geweest. Het laatste verschil levert de rest op.
Het aantal malen dat het deeltal aftrekbaar is, wordt telkens opgeschreven, te beginnen bij het meestwaardige cijfer. Dit levert uiteindelijk het quotiënt op.

Bij binair delen gaat het net zo. Het is zelfs nog eenvoudiger, omdat de deler telkens 0 keer of 1 keer van het deeltal moet worden afgetrokken. De uitkomst bestaat slechts uit enen en nullen.

Bij een 8-bits deeltal moet er 8 keer een aftrekpoging plaatsvinden.

Het cijfer voor cijfer bekijken van het deeltal kan handig geschieden door telkens een bit van het deeltal in het verschilregister te schuiven.

Hier volgt een voorbeeld voor 4-bits getallen:

deeltal = %0111 (= 7)
deler = %0010 (= 2)

Er moet nu 4 keer geschoven en eventueel afgetrokken worden:

1. schuiven: %0000 1110
deler: %0010 _
 gaat niet (C=1) resultaat = 0
2. schuiven: %0001 1100
deler: %0010 _
 gaat niet (C=1) resultaat = 00
3. schuiven: %0011 1000
deler: %0010 _
verschil: %0001 resultaat = 001
4. schuiven: %0011 0000
deler: %0010 _
verschil: %0001 resultaat = 0011 (= 3)

Er is nu vier keer geschoven, zodat alle cijfers van het deeltal aan de beurt zijn geweest. De uitkomst van de deling is 3, rest 1. Dit klopt.

Merk op dat de uitkomst ook uit vier bits bestaat. We kunnen deze uitkomst binnenschuiven in het 4-bits register van het deeltal, terwijl dit deeltal in het verschilregister schuift. In het verschilregister staat na de laatste bewerking de rest. In het deeltalregister staat dan het resultaat, het quotiënt.

Op deze wijze komen we tot de oplossing zoals in het structogram van fig. 8.18 wordt getoond. Hierna volgt het programma.

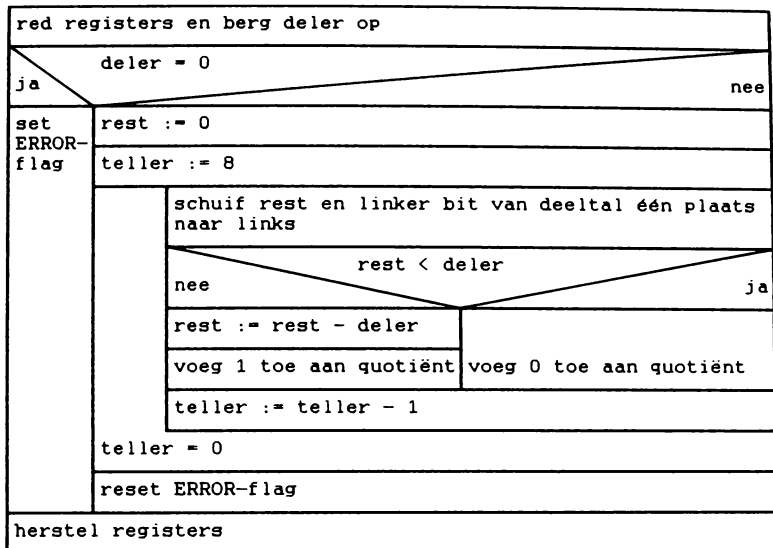


Fig. 8.18 Structogram voor snelle deelroutine

```

100 *****
110 * Subroutine FDIVU *
120 *****
130 *
140 *Progr.: J.G. Rouland
150 *Revisie 0 Datum: 29-04-1986
160 *
170 *FDIVU is een snelle deelroutine die het quotiënt berekent
180 *van twee 1-byte unsigned getallen.
190 *
200 *Het deeltal moet in accumulator A staan, de deler in B.
210 *
220 *Indien de deler 0 is, wordt de ERROR-flag geset. Dit is
230 *bit V van het condition code register.
240 *Als de deler ongelijk 0 is, wordt de ERROR-flag (V)
250 *gereset en komt het quotiënt in A en de rest in B.
260 *
270 FDIVU verandert alleen de inhoud van A, B en CC.
280 *
290 *
300 DELER RMB 1 reserveer ruimte voor de deler
310 *
320 FDIVU PSHS X red X
330 STB DELER berg de deler op
340 BNE DIVIDE spring als deler ongelijk 0 is
350 ORCC %#00000010 set ERROR-flag

```

360	BRA	ENDDIV	herstel X en keer terug
370	DIVIDE	CLRB	maak rest gelijk aan 0
380	LDX	#8	teller:=8
390	NEXT	LSLA	schuif rest 1 plaats naar links
400	ROLB		voeg linker bit deeltal toe
410	CMPB	DELER	spring als
420	BLO	DECR	rest < deler
430	SUBB	DELER	rest := rest - deler
440	INCA		voeg een 1 toe aan het quotiënt
450	DECR	LEAX -1,X	verlaag teller; spring als nog
460	BNE	NEXT	geen 8 keer is geschoven
470	ANDCC	#11111101	reset ERROR-flag
480	ENDDIV	PULS X	herstel X
490	RTS		keer terug

Door het deeltal bit voor bit naar buiten te schuiven, komt er telkens ruimte vrij, die gebruikt wordt om het quotiënt naar binnen te schuiven.

In regel 390 worden deeltal en quotiënt in accumulator A één plaats naar links geschoven. Hierbij wordt de linker bit van het deeltal in de carry geschoven, terwijl er rechts een bit vrij komt ten behoeve van het volgende cijfer van het quotiënt. Op deze plek wordt een 0 gezet bij het schuiven.

In regel 400 wordt de rest één plaats naar links geschoven, waarbij de carry (de linker bit van het overgebleven deel van het deeltal) in bit 0 komt te staan. Vervolgens wordt gekeken of de deler van deze nieuwe rest is af te trekken. Gaat dit niet, dan moet er een 0 worden toegevoegd aan het quotiënt. Dit is al gebeurd bij het opschuiven van deeltal en quotiënt (in regel 390), zodat er hiervoor geen aparte actie meer hoeft te worden ondernomen. Gaat de aftrekking wel, dan moet bit 0 van het quotiënt 1 worden. Dit kan dan gebeuren door er 1 bij op te tellen (regel 440), immers bit 0 was 0.

Het is natuurlijk ook mogelijk om het quotiënt niet op te geven als een geheel getal met een rest, maar als een niet-geheel getal met een aantal cijfers achter de binaire punt. Dit wordt gerealiseerd door een aantal malen verder te schuiven met de rest, waarbij er telkens een 0 wordt binnengeschoven. Hierbij wordt elke keer nagegaan of de deler van deze rest is af te trekken en dienovereenkomstig wordt aan het quotiënt een 1 of een 0 toegevoegd. Denk hierbij aan de decimale staartdeling. De binaire punt staat in het quotiënt rechts van de achtste bit, gerekend vanaf de meestwaardige bit (de linker bit).

8.7 Projectaanpak

8.7.1 Algemeen

U hebt reeds kennis gemaakt met een aantal voorbeelden, waarbij een oplossing voor een bepaald probleem werd gevraagd. De oplossing kwam tot stand na een aantal stappen gedaan te hebben. In deze paragraaf zal nog eens in zijn algemeenheid worden ingegaan op deze stappen.

Om in een zo kort mogelijke tijd tot een optimale oplossing van een probleem te komen, is het nodig het probleem structureel aan te pakken. Hierbij zijn een aantal fasen te onderscheiden, onafhankelijk of het nu gaat om het realiseren van een hardwareschakeling, een softwarepakket of een combinatie van beide, dan wel om het oplossen van een economisch probleem of het verwerken van gegevens van een weersatelliet, enzovoorts.

De fasen bij het oplossen van een probleem:

1. probleemdefinitie
2. probleemanalyse
3. planning en kostenberekening
4. ontwerp
5. bestellen onderdelen
6. fabricage
7. debugging
8. documentatie
9. vrijgave/overdracht
10. onderhoud

8.7.2 Probleemdefinitie

De probleemdefinitie bestaat uit het vastleggen van de functies, die moeten worden vervuld, en van de signalen en resultaten, die moeten worden gegenereerd.

Als het probleem betrekking heeft op het ontwerpen en bouwen van een CV-installatie, dan valt hierbij te denken aan de volgende probleemdefinitie:

Voor een basisschool moet er een CV-installatie worden geleverd met een zo laag mogelijk energieverbruik en een hoog gebruikersgemak, binnen een bepaald budget. De installatie moet zijn verdeeld in twee groepen. De ene groep omvat de hal en de vergader/koffiekamer, de andere omvat de rest van het schoolgebouw. Het gebouw moet voorzien zijn van een vorstbeveiliging. De verwarming dient automatisch in te schakelen op een zo laat mogelijk tijdstip, met dien verstande dat de lokalen om 8.30 uur een temperatuur van 20 graden kunnen bereiken en de rest van het schoolgebouw (zoals de hal) 18

graden. Voorts moet de verwarming zo vroeg mogelijk automatisch uitschakelen, echter zodanig dat de genoemde temperaturen gelden tot een tijdstip dat per dag van de week instelbaar is. Op zaterdag en zondag blijft de verwarming uit. Met behulp van een "vakantieschakelaar" kan de verwarming (tijdens vakanties) worden uitgeschakeld. Tenslotte moet er "overwerktimers" aanwezig zijn, die tot een maximum van 6 uur instelbaar moeten zijn, opdat de beide groepen van het gebouw, onafhankelijk van elkaar, na het einde van de schooldag kunnen blijven verwarmd ten behoeve van vergaderingen, ouderavonden en dergelijke. Tenslotte moet er een duidelijke en beknopte bedieningshandleiding worden meegeleverd.

Het spreekt vanzelf dat bij de probleemdefinitie een zeer nauwe samenwerking en een goede communicatie met de klant/gebruiker noodzakelijk is. De probleemdefinitie moet leiden tot een functionele specificatie.

8.7.3 Probleemanalyse

Bij de probleemanalyse wordt er nagegaan wat er allemaal moet gebeuren om tot de oplossing van het probleem te komen. Er wordt onderzocht welke acties en bewerkingen nodig zijn, welke signalen en ingangssgegevens er (nodig) zijn en welke uitgangssignalen en resultaten er gegenereerd moeten worden.

In het voorbeeld van de CV-installatie zal er een weersafhankelijke regeling noodzakelijk zijn, teneinde het energieverbruik laag te houden. Derhalve zijn er als ingangssignalen de buitentemperatuur en een instelbare dag- en nachttemperatuur. Om een juiste regeling mogelijk te maken zijn er verder nodig de temperaturen van het bij de ketel aan- en afgevoerde water. Voor het signaleren van een te lage (of te hoge) druk van het water in leidingen en radiatoren, moet ook de waterdruk worden gemeten. Als uitvoersignalen zijn nodig de sturing van de gasklep (voor de hoofdblander), de stuursignalen voor de twee waterpompen en de waterkleppen en een lichtsignaal in geval van te lage waterdruk (bijvoorbeeld ten gevolge van lekkage). Onderzocht moet worden of het toepassen van een hoogrendementketel economisch verantwoord is, evenals het toepassen van een regeling, die behalve van de buitentemperatuur, ook afhankelijk is van de windsnelheid.

Bij de probleemanalyse wordt het probleem in kleinere brokken verdeeld. Er wordt een structuur aangebracht, zodat het geheel overzichtelijk wordt. Het werkt erg goed om het geheel in kaart te brengen door schema's te tekenen, waarin duidelijk de relaties tussen de diverse blokken aangegeven zijn. Ook de ingangs- en uitgangssignalen zijn hierin opgenomen (zie fig. 8.19).

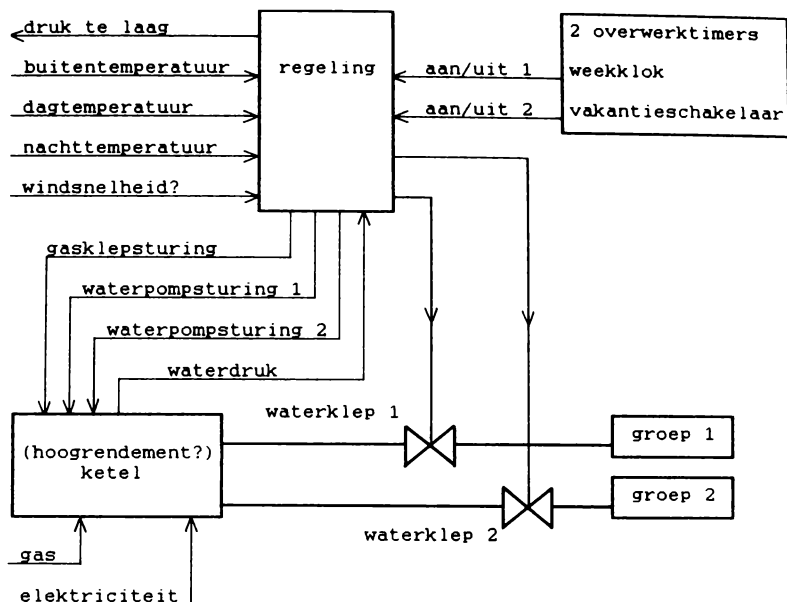


Fig. 8.19 Blokdiagram van CV-installatie

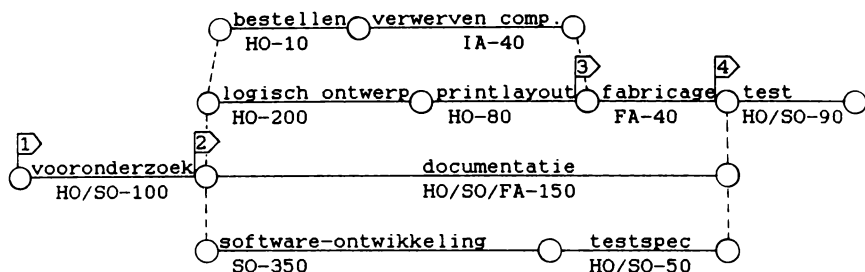
Ook tijdens de probleemanalyse is intensief contact met de klant/gebruiker noodzakelijk om misverstanden uit de weg te ruimen. Het is bovendien nuttig om vroegtijdig de (sluimerende) wensen van de klant/gebruiker boven tafel te krijgen, opdat niet kort na de aflevering van het produkt reeds wijzigingen zullen ontstaan, die bij beter overleg waren voorkomen. Zo kan het zijn dat de klant best bereid is meer voor de CV-installatie te betalen, als blijkt dat de meerprijs van een hoogrendementketel een zodanige energiebesparing oplevert, dat deze binnen een beperkt aantal jaren wordt terugverdiend. Ook kan het gebeuren dat een klant/gebruiker bepaalde zaken niet vraagt, omdat hij denkt dat deze (nog) niet realiseerbaar zijn of te duur.

8.7.4 Planning en kostenberekening

Uit de probleemanalyse kan een planning worden afgeleid. Het probleem is in kleinere stukken verdeeld. Deze kunnen eventueel verder verfiind worden en vervolgens plaatsen we de diverse activiteiten in een tijdschema, deels in serie en deels parallel. Activiteiten die in principe gelijktijdig kunnen

plaatsvinden, komen parallel te staan. Activiteiten die pas kunnen beginnen, nadat andere activiteiten zijn beëindigd, staan in serie met die activiteiten.

De diverse activiteiten worden tevens aan de verschillende afdelingen of personen worden toegewezen. Tenslotte wordt per activiteit het benodigde aantal uren bijgeschreven. In de zo ontstane planning is de ontwerpfase tot en met de vrijgave/overdracht opgenomen, inclusief bepaalde mijlpalen en data. Een mijlpaal is een karakteristiek punt, waarop een bepaalde activiteit wordt afgesloten.



- | | |
|---|-------------------|
| 1. start | HO=Hardware Ontw. |
| 2. vrijgave functionele specificaties | SO=Software Ontw. |
| 3. vrijgave hardware ontwerp | FA=Fabricage Afd. |
| 4. vrijgave software ontwerp en testspecs | IA=Inkoop Afd. |

Fig. 8.20 Gedeelte uit de planning van een project

Als bijvoorbeeld het hardware-ontwerp (logisch ontwerp + printlayout) gereed is, kan dit worden afgesloten door het ontwerp in een vrijgave-vergadering goed te keuren (mijlpaal 3, fig. 8.20). Hierna kan met de fabricage van het produkt worden begonnen. Bij zo'n moment moeten alle betrokkenen geraadpleegd worden en hun goedkeuring aan het tot dan toe bereikte resultaat hechten. In het genoemde voorbeeld zijn dat de ontwerpers, de klant/gebruiker, de fabrikanten en de testers.

Fig. 8.20 laat een voorbeeld zien van een planning. Het gaat hierbij om het leveren van een digitale regelaar voor een CV-installatie. Het gaat hierbij om een grove planning, die nog verdere detaillering behoeft binnen de deelnemende afdelingen. Hier wordt het werk per man/vrouw of per groepje medewerkers gepland, waarbij de taken in kleinere, beter te bewaken activiteiten zijn onderverdeeld.

Uit de probleemanalyse en de tijdsinvestering kan een kostenberekening worden afgeleid. Het is mogelijk dat hieruit een aanpassing van het eisenpakket uit voortvloeit en dat dientengevolge de probleemdefinitie en alles wat daarop volgt (probleemanalyse enzovoorts) moet worden aangepast.

8.7.5 Ontwerp

Nu kan het eigenlijke ontwerp beginnen. Deze fase is te splitsen in twee onderdelen, het functionele ontwerp en het technische ontwerp.

Bij het functionele ontwerp worden alle functies bepaald, die nodig zijn voor de realisatie. Er wordt nog niet gedacht over de uiteindelijke oplossing. Bepaald wordt bijvoorbeeld welk vermogen de CV-ketel moet hebben en of het al dan niet een hoogrendementketel moet zijn. Het gaat hierbij om het opstellen van de functionele specificaties van de diverse onderdelen van de te ontwerpen apparatuur en programmatuur. Vervolgens worden gedetailleerde blokdiagrammen en structogrammen opgesteld. Ook hierbij is het zinvol de klant/gebruiker te betrekken, opdat deze inzicht krijgt in de moeilijkheden en mogelijkheden die zich voortdoen. Hieruit kunnen weer wijzigingen voortvloeien, die effect hebben op een van de voorgaande fasen. Er moet dan telkens worden teruggekoppeld. Bovendien is het belangrijk de klant/gebruiker telkens te wijzen op de consequenties van de wijzigingen met betrekking tot kosten en tijd. Hierna wordt bepaald welke onderdelen zelf gemaakt worden en welke zullen worden ingekocht.

Bij het technisch ontwerp wordt, uitgaande van het functionele ontwerp, het te vervaardigen produkt daadwerkelijk ontwikkeld met bestaande onderdelen (plaatmateriaal, transistoren, IC's, instructies, formules, enzovoorts). Hierbij ontstaat een technische specificatie, die de technische eigenschappen van het ontwerp beschrijft. Denk bijvoorbeeld aan het opgenomen elektrische vermogen van een apparaat. Een aantal technische eisen, waaraan een ontwerp moet voldoen, liggen van tevoren al vast. Denk bijvoorbeeld aan toelaatbare straling en verstoring van de netspanning en aan de waarden van de netspanningen en omgevingstemperaturen, waarbij het apparaat nog correct moet functioneren.

Contact met de klant/gebruiker is bij de technische ontwerpfase niet nodig, tenzij deze bepaalde technische oplossingen voorschrijft.

In de ontwerpfase ontstaan tekeningen, programma-listings en dergelijke.

De modulariteit is van groot belang om het werk gemakkelijk te kunnen verdelen over meerdere afdelingen of personen en tevens om de voortgang van het project beter te kunnen bewaken.

8.7.6 Bestellen van onderdelen

De benodigde onderdelen moeten zo spoedig mogelijk worden besteld om te voorkomen dat de produktie stagneert. Hierbij

gaat het niet alleen om materiaal, maar ook om in te kopen software pakketten. Let hierbij op de soms lange levertijden. Van belang is zoveel mogelijk tweede en eventueel derde leveranciers te vinden, opdat de levering gewaarborgd is, als de eerste leverancier niet aan zijn verplichtingen kan voldoen.

8.7.7 Fabricage

Hier wordt het produkt gebouwd.

8.7.8 Debugging

In deze fase wordt het apparaat en/of softwarepakket getest en worden de fouten hersteld. Hier is het modulair opgebouwd zijn van het produkt van grote waarde, omdat de diverse modules eerst kunnen worden getest en daarna pas de grotere onderdelen en tenslotte het geheel. Het testen van het produkt in zijn geheel is aanzienlijk moeilijker dan het testen van de kleinere onderdelen. Het debuggen van het produkt geschiedt derhalve vaak via de bottom-up methode.

Bij het testen moet men, behalve de doorgaans voorkomende situaties, ook de randgevallen en de normaal niet optredende toestanden meenemen. Denk bij programma's bijvoorbeeld aan positieve en negatieve getallen en aan (delen door) 0.

8.7.9 Documentatie

Diverse documenten horen te ontstaan tijdens alle fasen, vanaf de probleemdefinitie tot en met het onderhoud. Een goede documentatie is van wezenlijk belang bij het snel en betrouwbaar opsporen en oplossen van problemen. Ook bij het aanbrengen van wijzigingen is een goede documentatie onontbeerlijk.

8.7.10 Vrijgave/overdracht

Aan het einde van het project wordt het produkt vrijgegeven en overgedragen aan de klant/gebruiker. Vaak moet er apparatuur en/of programmatuur bij de klant worden geïnstalleerd en moet de klant worden opgeleid om het produkt te gebruiken.

Er vindt een nacalculatie plaats om te bepalen wat het produkt werkelijk heeft gekost aan arbeidsuren en aan materiaal. Lang niet altijd heeft deze nacalculatie invloed op de prijs, die de afnemer betaalt. Het is wel nuttig deze calculatie te doen, teneinde van eventuele misrekeningen te leren.

8.7.11 Onderhoud

Hieronder wordt niet alleen verstaan het verlenen van service bij storingen, maar ook het blijvend ondersteunen van de klant/gebruiker tijdens de levensduur van het produkt. Denk aan het aanbrengen van bepaalde verbeteringen, waarover natuurlijk wel vooraf afspraken met de klant/gebruiker dienen te worden gemaakt.

Het aanpassen van het produkt aan nieuwe wensen van de klant/gebruiker valt eveneens onder onderhoud (maintenance).

Een produkt is vele malen gemakkelijker te wijzigen als het modulair is opgebouwd. De invloed van veranderingen op andere delen van het produkt is veel beter te overzien. Hierdoor is de kans om nieuwe fouten te introduceren, veel geringer.

8.8 Samenvatting

Module: zelfstandig onderdeel van een programma.

Subroutine: programma-onderdeel, dat vanuit elke willekeurige plaats in een sourceprogramma kan worden aangeroepen; na uitvoering wordt door de instructie RTS automatisch teruggesprongen naar het aanroepende programma.
De executietijd van het programma neemt iets toe bij gebruik van subroutines.
Worden de subroutines meerdere malen aangeroepen, dan worden source- en objectprogramma korter.

Macro: zelfstandig onderdeel van een programma, dat door de assembler op elke willekeurige plaats in het objectprogramma kan worden tussengevoegd; er wordt hierbij niet gesprongen.
De executietijd neemt niet toe bij gebruik van macro's en het objectprogramma wordt niet korter.
Het sourceprogramma wordt korter, indien de macro's meerdere keren worden aangeroepen.
Bij een macro kunnen parameters worden meegegeven, die instructies wijzigen. Hierdoor kan een macro stukken software vervangen, die niet helemaal gelijk zijn.

Parameteroverdracht tussen modules kan op de volgende manieren geschieden:

1. via registers
2. via vaste geheugenplaatsen
3. via de stack
4. via een pointer

- als pointer kan fungeren:
 - a. een 16-bits register
 - b. het direct page register
 - c. 2 vaste geheugenlocaties
 - d. 2 stacklocaties

8.9 Opgaven

- 1 Waarom is het bij subroutines, in tegenstelling tot bij macro's, niet mogelijk parameters toe te wijzen, die instructies veranderen?
- 2 Is het mogelijk om met behulp van een subroutine een wachttijd te realiseren, die exact n ms bedraagt, waarbij n variabel is?
Motiveer uw antwoord.
- 3 Ontwerp de subroutine ADD4 van voorbeeld 8.7, waarbij de optelling in een programma lus plaatsvindt, zonder gebruik te maken van een macro of subroutine.
- 4 Wat moet er in de subroutine RAMMAP van voorbeeld 8.9 veranderen, indien het testprogramma meer dan één pagina van het geheugen omvat?
Geef de veranderde en/of toegevoegde instructies.
- 5 Ontwerp een subroutine, die elke bit van een bepaalde geheugenpagina test.
Het paginanummer staat in het direct page register. Indien alle geheugenlocaties van die pagina in orde zijn, moet $Z=1$ zijn. In het andere geval moet $Z=0$ zijn en tevens moet X het adres bevatten van de eerste locatie waar een fout optreedt. Ontwerp eerst een structogram.
- 6 Ontwerp een subroutine, die nagaat of een bepaalde naam voorkomt in een lijst.
Stel u hebt een lijst van (maximaal 127) type-aanduidingen van elektronische bouwstenen (transistoren, dioden, IC's, enzovoorts). U wilt nagaan of een bepaalde component in die lijst voorkomt. De lijst bestaat uit een aantal namen, strings van karakters. Deze strings zijn onderling gescheiden door een $\langle ; \rangle$. Het einde van de lijst wordt aangegeven door $\langle CR \rangle$. Het beginadres van de gezochte string staat in register Y. De gezochte string wordt afgesloten door $\langle CR \rangle$. Het beginadres van de lijst staat in register U. Indien de gezochte type-aanduiding niet in de lijst voorkomt, moet $Z=0$ worden. In het andere geval moet $Z=1$ worden en moet X het beginadres bevatten van de gezochte string in de lijst.

- 7 Schrijf een subroutine om twee 1-byte unsigned getallen op elkaar te delen door na te gaan hoeveel keren de deler kan worden afgetrokken van het deeltal.
Ga ervan uit dat het deeltal in A staat en de deler in B.
Het quotiënt moet in A worden gezet en de rest in B.
- 8 Wat moet er in de subroutine FDIVU van voorbeeld 8.11 worden veranderd om als resultaat een 16-bits getal te krijgen met de binaire komma in het midden?

9 INTERRUPTS

9.1 Specifieke doelstelling

Na het bestuderen van hoofdstuk 9 moet u tot het volgende in staat zijn:

- * De functie van interrupts kennen.
- * Weten hoe een interruptafhandeling geschiedt.
- * De verschillen kennen tussen hardware interrupts, software interrupts en traps.
- * De eigenschappen van de diverse hardware en software interrupts van de MC6809-microprocessor kennen.
- * Weten wat de microprocessor automatisch doet en waar de programmeur voor moet zorgen bij het optreden van interrupts.
- * Interrupt Service Routines kunnen schrijven.
- * Weten wat er met polling en vectoring wordt bedoeld.

9.2 Begripsvorming

Een interrupt is een onderbreking van de huidige bezigheden ten gevolge van een externe gebeurtenis, zodanig dat de bezigheden na de onderbreking weer kunnen worden hervat. Essentieel is hierbij, dat de onderbreking moet worden veroorzaakt van buitenaf.

Veronderstel u leest een boek. Na enige tijd besluit u met het lezen te stoppen, omdat u vindt dat u lang genoeg hebt gelezen. Er is hier geen sprake van een interrupt, omdat u zelf besluit tot het beëindigen van uw activiteit. U wordt immers niet gestoord door een externe gebeurtenis. Na enige tijd kunt u eventueel weer besluiten verder te lezen.

Het kan echter ook zijn, dat u gestoord wordt door een externe gebeurtenis, bijvoorbeeld door het overgaan van de telefoonbel. Nu is er sprake van een interrupt. Het opgebeld worden geschiedt geheel buiten u om. U kunt op elk moment, buiten uw wil om, worden opgebeld.

Doorgaans gaat u nu over tot het afhandelen van de interrupt. U legt een papiertje in uw boek om te weten waar u gebleven bent met lezen. Vervolgens legt u het boek terzijde en neemt u de hoorn van de haak. Na het beëindigen van het gesprek legt u de hoorn neer, neemt uw boek op en gaat verder met het lezen op de plaats waar het papiertje ligt.

Nu kan in principe een telefoongesprek ook plaatsvinden zonder gebruik te maken van interrupts. Dit is echter bijzonder onhandig. Als uw telefoontoestel geen bel (of andere

Voorbeeld 9.1 (beheer)

Een computersysteem werkt doorgaans aan diverse taken "tegelijkertijd" (multitasking), zodat bijvoorbeeld verschillende gebruikers "tegelijkertijd" (multi-user) programma's kunnen laten uitvoeren. Dit kan geschieden doordat het operating system aan elke taak een bepaalde tijd toewijst. Er wordt bij toerbeurt gedurende de toegewezen tijd aan de diverse taken gewerkt (time-sharing). Is die tijd verstreken, dan wordt het lopende programma onderbroken en wordt er doorgegaan met het uitvoeren van de volgende taak. Zo wordt taak 1 na de toegewezen tijd t1 onderbroken en taak 2 wordt voortgezet. Na t2 wordt taak 2 onderbroken en wordt gedurende t3 aan taak 3 gewerkt. Als alle taken aan de beurt zijn geweest, wordt taak 1 weer voortgezet gedurende een tijd t1, enzovoorts.

Voorbeeld 9.2 (service)

De operator onderbreekt het systeem om bijvoorbeeld een testprogramma uit te voeren, teneinde na te gaan of een bepaald deel van het systeem nog correct functioneert.

Voorbeeld 9.3 (beveiliging)

Bij een apparatuurfout (bijvoorbeeld een niet-functionerende printer) moet het lopende programma worden onderbroken en moet er een melding worden gegeven aan de operator, die dan actie kan ondernemen (bijvoorbeeld de printer inschakelen of van papier voorzien).

Bij een programmatuurfout (bijvoorbeeld bij delen door 0) moet het programma worden afgebroken en moet er als output een foutmelding worden gegenereerd.

Voorbeeld 9.4 (communicatie)

Een proces kan het lopende programma onderbreken ten behoeve van onmiddellijke actie als bijvoorbeeld het vloeistofniveau in een tank te laag wordt.

Ten behoeve van interrupts moeten er enige voorzieningen worden getroffen:

1. Er moet een mogelijkheid zijn om de interrupt kenbaar te maken (denk aan de telefoonbel).
2. Naar aanleiding van het optreden van een interrupt moet er een bepaalde activiteit uitgevoerd kunnen worden (denk aan het opnemen van de telefoonhoorn).
3. De onderbroken activiteit moet zinvol voortgezet kunnen worden (bladwijzer in het boek leggen bij het stoppen met lezen).
4. Bepaalde interrupts moeten geblokkeerd kunnen worden (telefoonbel uitschakelen).
5. Bij meerdere interrupts moet nagegaan kunnen worden waar de interrupt vandaan komt (bijvoorbeeld de deurbel en de telefoonbel hebben een verschillend geluid).

Er zijn drie soorten interrupts:

1. hardware interrupts
2. software interrupts
3. traps

Hardware interrupts

Voordat een interrupt kan worden afgehandeld, moet deze eerst aan de microprocessor worden gemeld. Dit geschiedt via een aparte ingang van de microprocessor. Via deze ingang wordt het interruptsignaal aan de microprocessor doorgegeven. We hebben hier te maken met zogenaamde hardware interrupts, omdat externe hardware via zo'n interruptingang het lopende programma kan onderbreken. Zo'n onderbreking kan op elk willekeurig moment optreden en is derhalve asynchroon ten opzichte van het programma.

Een interruptingang van de microprocessor wordt meestal gerekend tot de controlbus.

Software interrupts

Naast een of meerdere hardware interrupts kennen microprocessoren bovendien een of meerdere software interrupts. Een software interrupt is een interrupt, die wordt uitgevoerd bij een bepaalde instructie, die op een door de programmeur bepaalde plaats in het programma voorkomt. Een software interrupt is dus synchroon ten opzichte van het programma. Software interrupts worden vaak gebruikt om functies van de systeem software aan te roepen, die ook door de applicatieprogrammeurs gebruikt kunnen worden. In dit geval wordt ook wel gesproken van system calls.

Traps

Sommige microprocessoren veroorzaken zelf intern op de chip een interrupt bij het optreden van een bepaalde situatie, die ontstaat ten gevolge van het programma zelf. Zo genereert de MC68000-microprocessor bijvoorbeeld een interne interrupt indien er een niet-bestaande opcode wordt gedetecteerd door de processor of wanneer er bij een divide-instructie wordt gedeeld door 0. Zo'n interne interrupt wordt ook wel trap genoemd. Ook een trap is synchroon ten opzichte van het programma, omdat hij optreedt als direct gevolg van het uitvoeren van een instructie. Hij treedt echter in tegenstelling tot een software interrupt niet altijd op bij het uitvoeren van bepaalde instructies, maar alleen in geval van een uitzonderingstoestand, zoals bij het delen door 0.

9.3 Interruptafhandeling

Interrupts detecteren

De interruptaanvraag moet voldoende lang aanwezig zijn, opdat de microprocessor de aanvraag kan detecteren. Is de

interruptaanvraag te kort, dan wordt deze niet gedetecteerd en wordt er geen actie ondernomen. Het programma wordt zonder onderbreking voortgezet.

De interruptaanvraag moet tijdig worden weggehaald (door de aanvrager). Is de interrupt afgehandeld, met andere woorden zijn de gewenste acties uitgevoerd, dan wordt het onderbroken programma weer voortgezet. Is de interruptaanvraag nog aanwezig, dan wordt de interruptafhandeling opnieuw gestart en wordt de door de (oude) interrupt gevraagde actie nogmaals uitgevoerd. Het is dus nodig de interruptaanvraag weg te halen, voordat de afhandeling beëindigd is, dus voordat er wordt teruggekeerd naar het onderbroken programma.

In het algemeen is er een terugmelding van de microprocessor aan de interruptaanvrager nodig, opdat deze weet wanneer hij de interruptaanvraag kan weghalen. Hierop komen we terug in hoofdstuk 10.

Programma onderbreken

Om het onderbroken programma na de interrupt zinvol te kunnen voortzetten, moet de microprocessor eerst de instructie afmaken, waar hij mee bezig is. De interrupt vraagt om een bepaalde activiteit, dus om het uitvoeren van instructies. Hierdoor gaat de oorspronkelijke inhoud van het instructieregister verloren. Derhalve mag de processor niet halverwege een instructie stoppen.

Om terug te kunnen keren naar het programma moet de processor weten, waar hij gebleven was, met andere woorden, op welke geheugenlocatie de volgende instructie van het onderbroken programma staat. Het adres van die geheugenlocatie bevindt zich in de program counter (zie paragraaf 3.5). Door nu de inhoud van de program counter (net als bij een subroutine-aanroep, zie paragraaf 8.3) op te bergen op de stack, wordt de plaats waar het programma later dient te worden voortgezet, onthouden.

Een interrupt vraagt om acties, instructies. Deze beïnvloeden de bits van het condition code register. Daar een interrupt in principe op elk willekeurig moment kan optreden, kan de programmeur hiertegen geen maatregelen treffen. Bij het voortzetten van het programma mag de inhoud van het condition code register echter niet zijn gewijzigd. Derhalve moet de processor ook de inhoud van het condition code register bewaren. Ook dit gebeurt op de stack.

Veelal zet de microprocessor ook de inhoud van de overige registers van het programmeermodel automatisch op de stack bij het optreden van een interrupt.

Vanzelfsprekend moet de stack pointer een adres bevatten, waaronder zich voldoende vrije geheugenruimte bevindt, die als stack kan worden gebruikt. Pas als dit het geval is, mag een

interrupt optreden. Hiervoor zijn er in de hardware van de microprocessor bepaalde maatregelen getroffen, zoals in paragraaf 9.7 zal blijken.

Het is in principe ook mogelijk de registerinhouden van de microprocessor te redden in aparte, daarvoor bedoelde MPU-registers. In dit geval bevindt de stack zich als het ware op de microprocessorchip. Het nadeel hiervan is dat deze stackruimte erg beperkt is.

Interrupt Service Routine starten

Nu de maatregelen om het onderbroken programma later te kunnen voortzetten, zijn getroffen, kan met het uitvoeren van de door de interrupt gewenste activiteit worden begonnen. Deze activiteit wordt gerealiseerd door een aantal instructies, dus door een stukje programmatuur. Deze programmatuur wordt aangeduid met Interrupt Service Routine (ISR). De Interrupt Service Routine moet ergens in het werkgeheugen van de computer staan. Vervolgens dient de processor te weten op welk adres de ISR begint. Hiervoor zijn er diverse methoden:

1. De plaats waar de ISR begint, is in de hardware van de microprocessor vastgelegd. Dit betekent dat de ISR altijd op dezelfde plaats in het geheugen moet beginnen.
2. De microprocessor haalt het startadres van de ISR op een vaste geheugenlocatie op. Dit is het geval bij de MC6809. De programmeur moet ervoor zorgen dat het startadres van de ISR op die geheugenlocatie wordt opgeslagen, alvorens er een interrupt mag optreden. De locatie waar het startadres van de ISR staat, wordt interruptvector genoemd.
3. De interruptaanvrager levert zelf vectorinformatie via de databus. Dit gebeurt pas als de microprocessor daarom vraagt middels een Interrupt Acknowledge signaal (na het detecteren van het interruptsignaal van de aanvrager). Uit die vectorinformatie bepaalt de microprocessor vervolgens de interruptvector.
4. Het is ook mogelijk om met behulp van externe hardware het vectoradres van de microprocessor, dan wel de inhoud van de interruptvector te vertalen in een andere waarde. We spreken in dit geval van hardware vectoring (zie paragraaf 9.6).

ISR uitvoeren

Zodra de microprocessor de inhoud van de interruptvector (het startadres van de ISR) heeft opgehaald, wordt met het uitvoeren van de ISR begonnen. Dit gaat op dezelfde wijze als het uitvoeren van een gewoon programma. Telkens wordt de volgende instructie uit het geheugen opgehaald en uitgevoerd.

Een interrupt verstoort de normale flow van het programma. Hierbij wordt het programma trager, omdat de ISR tussendoor wordt uitgevoerd. Vandaar dat het van belang is om een ISR zo kort mogelijk te houden. Anders bestaat het gevaar dat programmadelen, die binnen een bepaalde tijd uitgevoerd moeten

zijn, foutief aflopen. Dit kan eventueel worden voorkomen door de interrupts tijdens het uitvoeren van deze programmadelen te blokkeren. Er kunnen echter ook zeer belangrijke interrupts voorkomen, die niet geblokkeerd mogen worden. Het blokkeren van interrupts komt in paragraaf 9.7 aan de orde.

In de ISR moeten de gewenste acties worden uitgevoerd. Er moet service worden verleend aan de interruptaanvrager. Bovendien moet de interruptaanvrager de melding krijgen dat zijn aanvraag gehonoreerd wordt. Hij moet dan de aanvraag laten vervallen. Zou de terugmelding niet plaatsvinden, dan zal de aanvrager de interrupt blijven activeren. Dit heeft tot gevolg dat er opnieuw naar het begin van de ISR wordt gesprongen, zodra deze is afgehandeld. Zodoende zou één interrupt diverse malen achter elkaar worden afgehandeld. Dit is ontoelaatbaar. Heeft een input device bijvoorbeeld één byte data gereed staan voor de microprocessor, dan zou deze byte diverse keren worden opgehaald en op meerdere geheugenlocaties worden opgeborgen, indien de interruptaanvraag te lang wordt geactiveerd. Een interruptaanvraag moet vervallen zijn, voordat de processor het uitvoeren van de ISR heeft voltooid.

Onderbroken programma voortzetten

Aan het einde van de ISR moet worden teruggesprongen naar het hoofdprogramma. Dit geschiedt met behulp van een return-instructie. De instructie RTS is hiervoor niet geschikt, omdat hierbij alleen het terugkeeradres van de stack wordt gehaald. Bij een interrupt is tenminste ook de inhoud van het condition code register op de stack geplaatst. Derhalve bestaat er voor een ISR een aparte return-instructie RTI (ReTurn from Interrupt). Hierbij worden automatisch alle registers, die bij de interrupt op de stack zijn gezet, weer van de stack teruggelezen. Zodoende kan het programma worden vervolgd met dezelfde informatie als voor de interrupt.

Merk op dat de informatie op de stack niet door de ISR moet worden overschreven en dat de stack pointer bij de RTI-instructie weer dezelfde inhoud moet hebben als aan het begin van de ISR, opdat het onderbroken programma kan worden voortgezet alsof er geen interrupt is geweest. Iets soortgelijks is reeds opgemerkt bij subroutines (zie paragraaf 8.3).

De volgorde waarin de registerinhouden bij een interrupt op de stack worden gezet, is dezelfde als bij een push-instructie. Bij de RTI-instructie worden de registers in omgekeerde volgorde van de stack gehaald, net als bij de pull-instructie.

Fig. 9.2 toont het structogram, behorende bij het afhandelen van een interrupt. De bijbehorende programmaflow en de inhoud van de stack ziet u in fig. 9.3.

huidige instructie afmaken
registerinhouden op de stack opbergen
program counter laden met het startadres van de ISR
ISR uitvoeren
(bij RTI-instructie) registerinhouden vanaf de stack vullen
onderbroken programma voortzetten

Fig. 9.2 Structogram van een interruptafhandeling

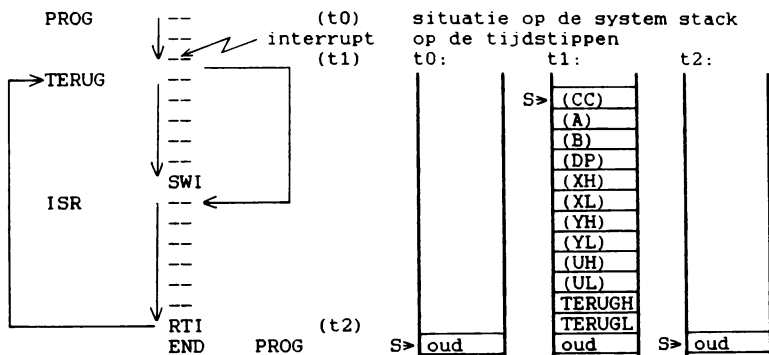


Fig. 9.3 Programmafloop bij een interrupt

Merk op dat op het moment van optreden van de interrupt geldt: (PC) = TERUG

Net als subroutines kunnen ook Interrupt Service Routines worden genest (zie ook paragraaf 8.3). Het enige verschil is dat er bij elke interrupt meerdere registers op de stack worden gezet, dus niet alleen de inhoud van de program counter.

9.4 Meerdere interrupts

Eén interrupt per interruptingang

Microprocessoren hebben doorgaans meerdere interruptingangen. Dit betekent dat er meerdere interruptaanvragers aangesloten kunnen worden. Bij elke interruptingang hoort een aparte interruptvector, zodat elke interrupt zijn eigen ISR heeft. Wordt er nu op elke ingang slechts één interruptaanvrager aangesloten, dan wordt er automatisch naar de bijbehorende ISR gesprongen, bij het optreden van een interrupt.

Zo heeft een microprocessor met vier interruptingangen (Interrupt ReQuest ingangen) IRQ1, IRQ2, IRQ3 en IRQ4 vier interruptvectoren VEC1, VEC2, VEC3 en VEC4, die de respectievelijke startadressen ISR1, ISR2, ISR3 en ISR4 van de vier Interrupt Service Routines bevatten. Sluiten we op deze ingangen vier interruptaanvragers aan, dan krijgen we de situatie van fig. 9.4.

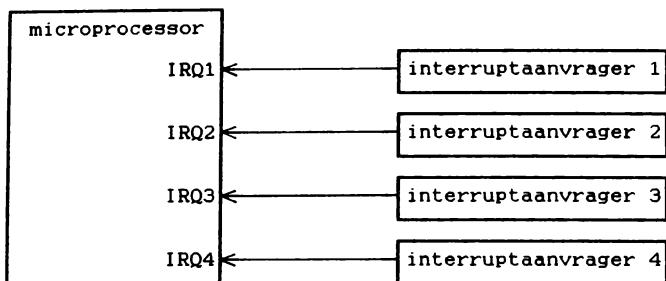


Fig. 9.4 Microprocessor met vier interruptingangen

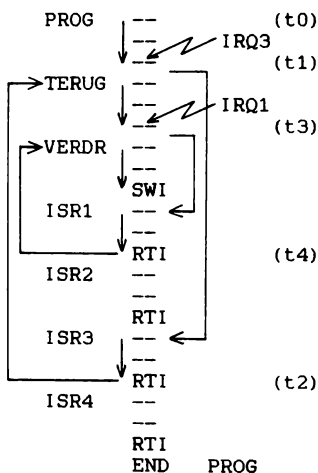


Fig. 9.5 Programmaflow bij interrupts die na elkaar optreden

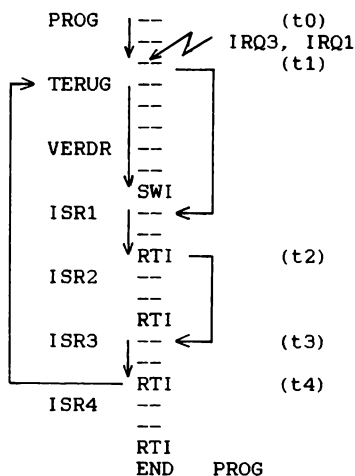


Fig. 9.6 Programmaflow bij ongeveer gelijktijdig optredende interrupts met verschillende prioriteiten

Bij het na elkaar optreden van interrupts zal de programmaflow er uit zien als weergegeven in fig. 9.5. Treden er interrupts ongeveer gelijktijdig op, dan wordt de situatie als weergegeven in fig. 9.6.

Fig. 9.7 laat de bij fig. 9.5 horende stackinhouden zien op de diverse tijdstippen.

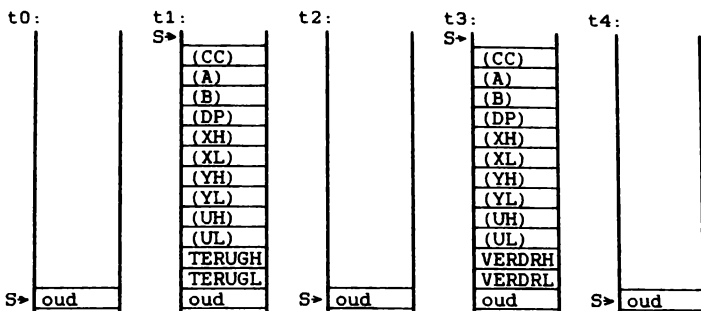


Fig. 9.7 Stackinhoud horende bij fig. 9.5

Prioriteit

Nu kan een microprocessor slechts één ISR tegelijk uitvoeren, zodat er een probleem ontstaat bij het optreden van meerdere interrupts tijdens het uitvoeren van een instructie. Dit is in de microprocessoren opgelost door de diverse interrupts een verschillende prioriteit te geven. Bij gelijktijdig optreden van verschillende interrupts wordt eerst de interrupt met de hoogste prioriteit afgehandeld en daarna die met een lagere prioriteit. Dit gaat door totdat alle interrupts afgehandeld zijn.

Stel dat IRQ1 de hoogste en IRQ4 de laagste prioriteit heeft. Treden IRQ3 en IRQ1 nu tegelijk op, dan krijgen we de programmafloop van fig. 9.6 en de bijbehorende stackplaatjes van fig. 9.8.

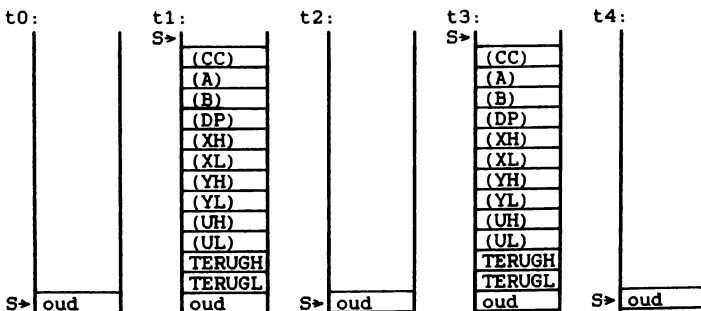


Fig. 9.8 Stackinhoud horende bij fig. 9.7

De programmafloop bij ongeveer gelijktijdig optredende interrupts kunnen we ook in een tijddiagram laten zien (zie fig. 9.9).

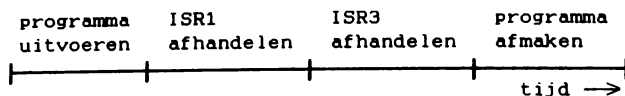


Fig. 9.9 Tijddiagram bij ongeveer gelijktijdig optredende interrupts met verschillende prioriteit

Treedt IRQ1 korte tijd na IRQ3 op, dan wordt ISR3 onderbroken door interrupt IRQ1. Deze heeft immers een hogere prioriteit dan IRQ3. Nadat ISR1 is uitgevoerd, wordt ISR3 afgemaakt en daarna wordt het programma weer voortgezet. Het tijddiagram wordt weergegeven door fig. 9.10. U ziet hier dat Interrupt Service Routines (evenals subroutines) genest kunnen zijn. De bijbehorende programmaflow en stackinhouden zijn gegeven in fig. 9.11, respectievelijk fig. 9.12.

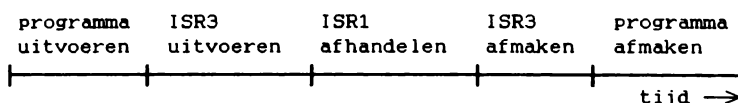


Fig. 9.10 Tijddiagram bij geneste interrupt service routines

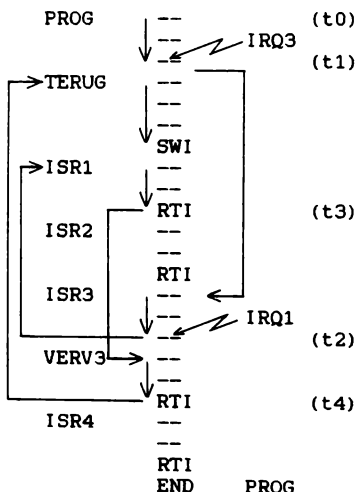


Fig. 9.11 Programmaflow bij geneste interrupts

Het last-in/first-out karakter van de stack zorgt ervoor dat telkens de uitvoering van het laatst onderbroken programma wordt hervat (zie fig. 9.12).

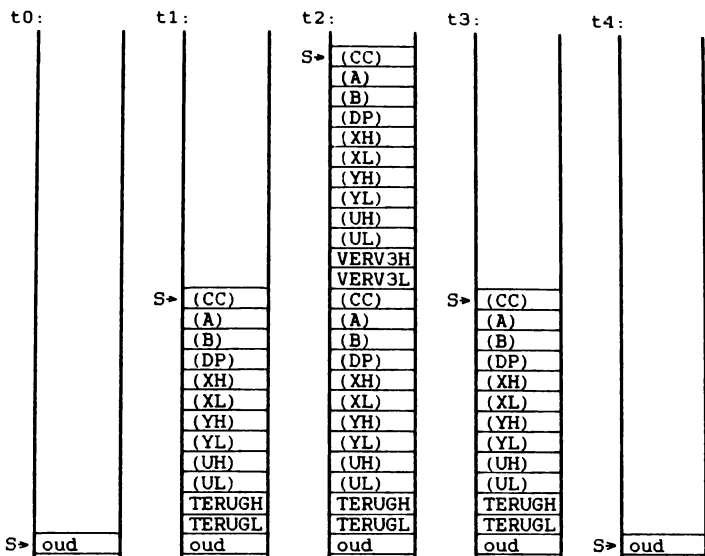


Fig. 9.12 Stackinhoud horende bij fig. 9.11

Meerdere interruptaanvragers op dezelfde interruptingang

Zolang elke interruptaanvrager op een andere interruptingang van de processor is aangesloten, treden er geen problemen op voor wat betreft de afhandeling van de interrupts. Doorgaans zijn er meer interruptaanvragers dan interruptingangen. Dit betekent dat er meerdere interruptaanvragers moeten worden aangesloten op eenzelfde interruptingang. Dit geschiedt meestal met behulp van een zogenaamde wired-AND (zie fig. 9.13).

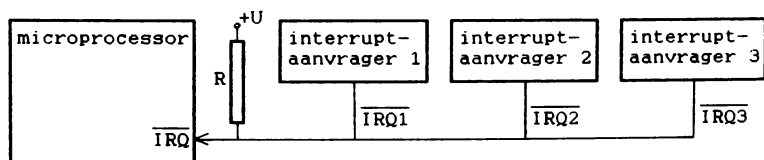


Fig. 9.13 Wired-AND ten behoeve van het aansluiten van meerdere interruptaanvragers op één interruptingang

Er is een interruptaanvraag als $\overline{\text{IRQ}}=1$, ofwel als de (geïnverteerde) ingang $\overline{\text{IRQ}}$ de waarde 0 aanneemt. Als een van de interruptaanvragers de $\overline{\text{IRQ}}$ -lijn laag maakt (door zijn $\overline{\text{IRQ}}$ -uitgang bijvoorbeeld via een transistor met aarde = 0 V te verbinden), dan krijgt de $\overline{\text{IRQ}}$ -lijn de waarde 0, ongeacht of de andere aanvragers hun $\overline{\text{IRQ}}$ -uitgang laag maken of niet. In het laatste geval laten de aanvragers hun $\overline{\text{IRQ}}$ -uitgang zweven, dus

is deze als het ware afgeschakeld van de buslijn. Dit betekent dat de $\overline{IRQ}$ -buslijn alleen de logische waarde 1 kan aannemen als alle interruptaanvragers hun $\overline{IRQ}$ -uitgang afgeschakeld hebben. Er is dan geen enkele interruptaanvraag. De zogenaamde pull-up weerstand R zorgt er dan voor dat de buslijn een hoog niveau (+U) krijgt, dus de logische waarde 1. Dan geldt $\overline{IRQ}=1$, ofwel $IRQ=0$, zodat de microprocessor geen interrupt detecteert.

Er wordt hier een and-functie van de $\overline{IRQ}$ -uitgangen van de interruptaanvragers gerealiseerd, immers de buslijn is 0 als een of meerdere $\overline{IRQ}$ -uitgangen 0 zijn en de buslijn is 1 als alle $\overline{IRQ}$ -uitgangen 1 zijn. De and-functie wordt hier niet gerealiseerd met behulp van een digitale bouwsteen, maar met behulp van de bedrading, vandaar dat er wordt gesproken van een wired-AND. Er geldt dus:

$$\overline{IRQ} = \overline{IRQ1} \cdot \overline{IRQ2} \cdot \overline{IRQ3}$$

Het signaal $\overline{IRQ}$ wordt in de microprocessor geïnverteerd, zodat

$$IRQ = \overline{\overline{IRQ1} \cdot \overline{IRQ2} \cdot \overline{IRQ3}} = IRQ1 + IRQ2 + IRQ3$$

Zo ontstaat er functioneel een OR-schakeling van de diverse interruptaanvragen. Zodra een of meerdere aanvragen worden gedaan, zal de microprocessor een interruptaanvraag detecteren en de Interrupt Service Routine uitvoeren. Om de wired-AND in een functionele OR te veranderen, zijn de interruptaanvragen inverse signalen. Op deze manier zijn de diverse aanvragen functioneel opgenomen in een OR-schakeling. Vandaar dat in de literatuur ook wel gesproken wordt van een wired-OR, hoewel dit strict genomen niet correct is. De OR-functie ontstaat pas na inverteren. Hiervoor is een elektronische schakeling (inverter) nodig. Er kan dus niet worden volstaan met alleen bedrading.

Een wired-AND wordt doorgaans gerealiseerd met behulp van schakelingen, die een open-collector of open-drain uitgang hebben (zie fig. 9.14). Als de transistor geleidt, is de uitgang vrijwel gelijk aan 0 V (= 0) en als de transistor niet geleidt, is de uitgang hoogimpedant (zwevend) en kan deze door de pull-up weerstand naar +U_v (= 1) worden getrokken. Overigens is het mogelijk om (bijvoorbeeld met schakelingen met open-emitter of open-source uitgangen) een echte wired-OR te maken.



Fig. 9.14 Open-collector uitgang en open-drain uitgang

Als de microprocessor in het geval van fig. 9.13 een interrupt detecteert, weet hij op dat moment niet van welke aanvrager deze afkomstig is. Er moet nu een methode gevonden worden om de aanvrager op te sporen. Dit kan gebeuren met behulp van

1. polling
2. vectoring

9.5 Polling

Werkwijze

Polling is het beurtelings aftasten van de diverse aanvragers om te achterhalen wie om een actie verzoekt. Hiertoe moet elke aanvrager een register bezitten, waarvan de zogenaamde interruptbit geset wordt, zodra een interrupt gegenereerd wordt door de betreffende aanvrager. Door dit register van de verschillende aanvragers beurtelings te adresseren via de adresbus en de inhoud via de databus binnen te halen, kan de processor door de interruptbit te testen, nagaan of de betreffende aanvrager een interrupt heeft gegenereerd. Dit gebeurt zodra er een interrupt door de processor is gedetecteerd. De ISR is dan inmiddels als gestart. Dit betekent dat de polling zich binnen de ISR moet afspelen en wel aan het begin. De diverse interruptaanvragers verzoeken om verschillende acties. Afhankelijk van wie de aanvrager is, moet er dus een bepaalde actie worden gestart. Daar de polling in dit geval met behulp van instructies (in de ISR) wordt uitgevoerd, spreekt men van software polling. Fig. 9.15 laat structogrammen zien voor deze software polling.

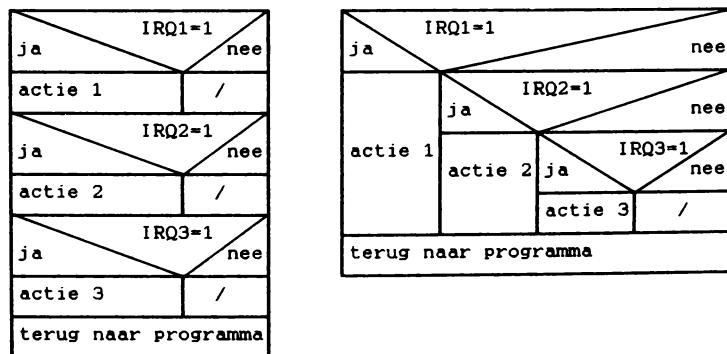


Fig. 9.15 Structogrammen voor software polling

Bij software polling krijgen de interrupts tevens een verschillende prioriteit. In fig. 9.15 ziet u dat er eerst wordt nagegaan of aanvrager 1 een interrupt heeft gegenereerd. Als dat zo is, wordt de bijbehorende actie uitgevoerd. De

eventuele overige interruptaanvragen zullen nog even onbeantwoord blijven. Deze hebben klaarblijkelijk een lagere prioriteit. De prioriteit wordt hier bepaald door de programmeur.

Bij het linker structogram van fig. 9.15 worden binnen dezelfde ISR alle aanwezige aanvragen afgehandeld. Pas daarna wordt er naar het programma teruggekeerd.

Bij het rechter structogram wordt er telkens naar het programma teruggesprongen, nadat er één interrupt is afgehandeld. Bij meerdere aanvragen tegelijk wordt er onmiddellijk weer teruggesprongen naar de ISR, zonder dat er een instructie van het programma is uitgevoerd. Het programma wordt voortgezet als alle aanvragen zijn afgehandeld.

Het nadeel van deze laatste methode is dat er telkens uit de ISR wordt teruggekeerd naar het programma om vervolgens weer onmiddellijk terug te keren naar de ISR en daar van voren af aan te beginnen. Dit kost extra tijd. Denk aan het telkens op de stack zetten en van de stack halen van de registerinhouden en het telkens helemaal opnieuw doorlopen van de polling routine.

Het voordeel is dat een aanvrager met een hoge prioriteit niet hoeft te wachten totdat alle acties behorende bij interrupts van een lagere prioriteit zijn uitgevoerd. Deze situatie kan optreden als een aanvrager met een hoge prioriteit een interrupt genereert, terwijl de ISR al uitgevoerd wordt ten gevolge van interrupts van aanvragers met een lagere prioriteit.

Bij het linker structogram wordt deze interrupt pas afgehandeld nadat alle lopende interrupts afgehandeld zijn. Bij het rechter structogram hoeft slechts te worden gewacht op het voltooiën van de lopende actie behorende bij een van de interrupts.

Het is sterk aan te raden om elke interruptaanvrager te pollen en niet aan te nemen dat de laatste aanvrager een interrupt heeft gegenereerd, als alle voorgaande dat niet gedaan hebben. Bij een storing op de interruptlijn wordt er ten onrechte naar de ISR gesprongen. Dan zou actie n, behorende bij aanvrager n (met de laagste prioriteit) worden uitgevoerd, als de laatste test ($IRQn=1$) zou worden weggelaten.

Voordeel van polling: grote flexibiliteit

De prioriteitsvolgorde wordt door de programmeur bepaald, evenals de beginadressen van de feitelijk uit te voeren acties. Dit alles is eenvoudig te veranderen.

Nadeel van polling: traagheid

Er gaat vrij veel tijd verloren met het opsporen van de interruptaanvrager. Dit probleem is groter naarmate het aantal aanvragers groter is. Dit tijdverlies kan bij snelle besturingssystemen ontoelaatbaar zijn. In dat geval moet men zijn toevlucht zoeken tot hardware vectoring.

9.6 Vectoring

Principe van hardware vectoring

Bij hardware vectoring worden de diverse interruptaanvragers niet aangesloten op de interruptingang van de microprocessor, maar op de verschillende interruptingangen van een externe hardwareschakeling. Verder worden de adresuitgangen van de microprocessor via deze schakeling met de adresbus verbonden. De procedure is nu als volgt:

1. Zodra een van de interruptingangen actief wordt, geeft deze schakeling een interrupt af naar de processor.
2. Als de processor hierop reageert door de bijbehorende interruptvector te adresseren, vertaalt de externe schakeling dit vectoradres in een ander adres of levert het startadres (of een gedeelte ervan) van de bijbehorende ISR. Deze informatie is afhankelijk van de interruptaanvraag, die binnen is gekomen op de interruptschakeling.
3. Is er geen interruptaanvraag of wordt er een ander adres dan de interruptvector door de processor afgegeven, dan wordt het adres zonder vertaling doorgegeven aan de adresbus, respectievelijk zet de interrupt hardware geen vectorinformatie op de databus.

Elke interruptaanvrager wordt uitgerust met een eigen vectoradres, hetgeen verwijst naar de bijbehorende interrupt service routine. Dit maakt polling overbodig. Er wordt nu bij een interrupt automatisch naar de juiste ISR gesprongen. Hierbij treedt geen tijdverlies op met zoeken naar de aanvrager. De externe hardware geeft aan waar zich de juiste interruptvector bevindt of waar de betreffende ISR begint.

Men spreekt van hardware vectoring, omdat het vectoradres of de vectorinformatie wordt geleverd door externe hardware.

Functies van de interrupt hardware

De interrupt hardware bevat drie functies:

1. De encoder
Deze zorgt ervoor dat slechts één interruptaanvraag wordt doorgelaten en voorziet de vectorvertaler, respectievelijk vectorgenerator van informatie ten behoeve van het leveren van de juiste informatie.
Het probleem van gelijktijdige interrupts kan hier eenvoudig worden opgelost door de interrupt hardware uit te rusten met een zogenaamde priority encoder, die alleen de aanvraag met de hoogste prioriteit doorlaat.
2. De vectoradresdecoder
Deze detecteert of de microprocessor de interruptvector adresseert en geeft een signaal ten teken dat het adres moet worden vertaald, respectievelijk dat er een interruptvector of vectorinformatie moet worden gegenereerd.
3. De vectoradresvertaler, respectievelijk vectorgenerator

De eerste vertaalt (op bevel van de vectoradresdecoder) het adres van de interruptvector, zoals die door de microprocessor wordt afgegeven, in een nieuw vectoradres, afhankelijk van de aanvrager.
De tweede levert de interruptvector (het startadres van de ISR) of de vectorinformatie, waaruit de microprocessor het startadres van de ISR afleidt.

Er bestaan diverse hardware methoden voor het opstarten van de juiste Interrupt Service Routine:

1. De interrupt hardware vertaalt het interruptvectoradres.
2. De interrupt hardware levert het startadres van de ISR.
3. De interruptaanvrager levert de vectorinformatie.
4. Hardware polling.
5. Software polling gecombineerd met hardware vectoring.
6. Interrupt levels (eventueel in combinatie met software polling of hardware vectoring).

Vectoradresvertaling

Bij deze methode wordt het interruptvectoradres door de externe hardware vertaald in een ander adres, dat afhangt van de interruptaanvrager. Dit vertaalde adres wordt aangeboden aan het geheugen, dat op zijn beurt de interruptvector levert via de databus (zie fig. 9.16 en 9.17).

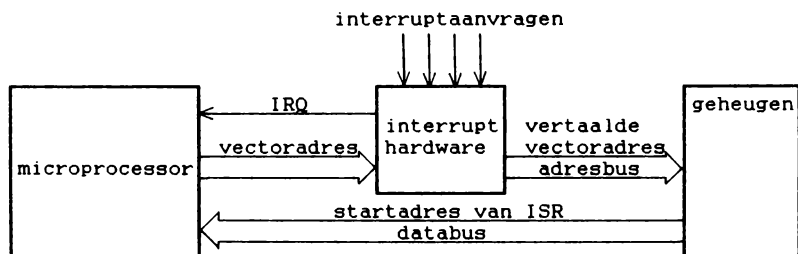


Fig. 9.16 Principe van vectoradresvertaling

Het is niet nodig om alle (16) adressignalen te vertalen. Er kan worden volstaan met een beperkt aantal, bijvoorbeeld drie. De bijbehorende adresuitgangen van de microprocessor zijn niet rechtstreeks, maar via de interrupt hardware met de adresbus verbonden.

Bedenk dat een adres uit 16 bits bestaat en dat een geheugenlocatie (bij 8-bits microprocessoren) slechts 8 bits bevat. Dit betekent dat er twee adressen nodig zijn voor het ophalen van het startadres van een interrupt service routine. Nu staat een adres op twee opeenvolgende geheugenlocaties, zodat bit A0 buiten beschouwing kan worden gelaten bij het vertalen.

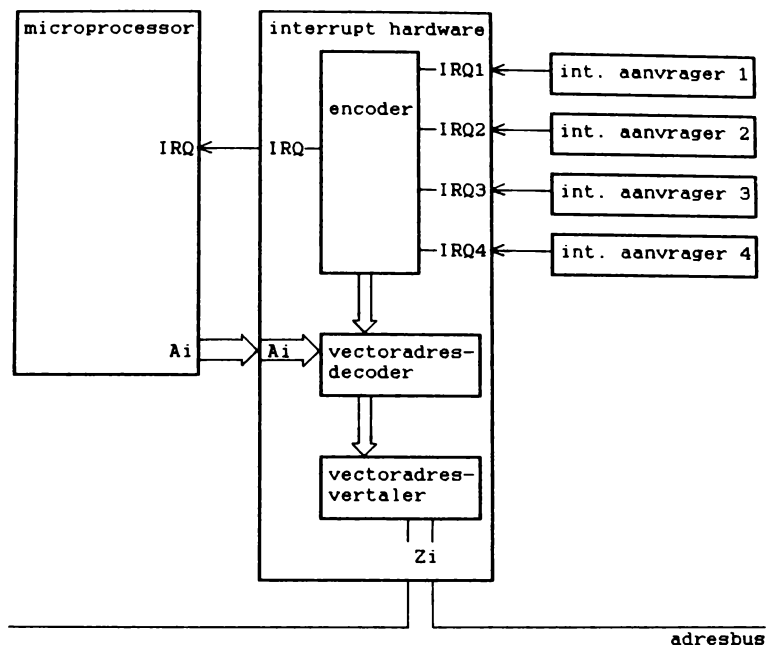


Fig. 9.17 Hardware vectoring met vectoradresvertaling

Bij vectoradresvertaling moeten de oude vectoren gehandhaafd blijven en moeten nieuwe vectoren worden toegevoegd. De vectoradresvertaling mag natuurlijk geen locatie opleveren waar zich een andere interruptvector bevindt! Dit betekent dat de nieuwe interruptvectoren elders in het geheugen moeten staan.

Voorbeeld 9.5

Stel dat de interruptvector zich bevindt op adres \$FFF8 en \$FFF9. We beïnvloeden slechts drie adresbits en laten de overige 13 ongewijzigd. Door de bits A4, A2 en A1 te vertalen van 100 in de waarden 000, 001, 010 of 011, afhankelijk van de interruptaanvrager, hebben we het ene vectoradres vertaald in vier nieuwe. De nieuwe interruptvectoren bevinden zich op geheel andere plaatsen in het geheugen, namelijk op \$FFE8 en \$FFE9, op \$FEEA en \$FEEB, op \$FEEC en \$FEED en op \$FEEF en \$FEEF.

Interrupt hardware kan als digitale bouwsteen worden gekocht. Zo'n component wordt wel Priority Interrupt Controller (PIC) genoemd (zie paragraaf 11.4).

Bij vectoradresvertaling moet de programmeur de startadressen

van zijn interrupt service routines opbergen in de geheugenlocaties, die (mede) door de externe hardware zijn aangegeven.

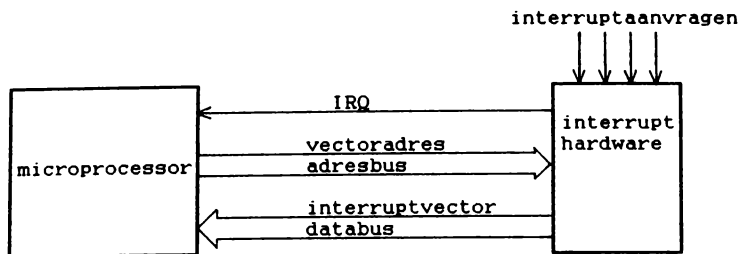


Fig. 9.18 Principe van het genereren van de interruptvector

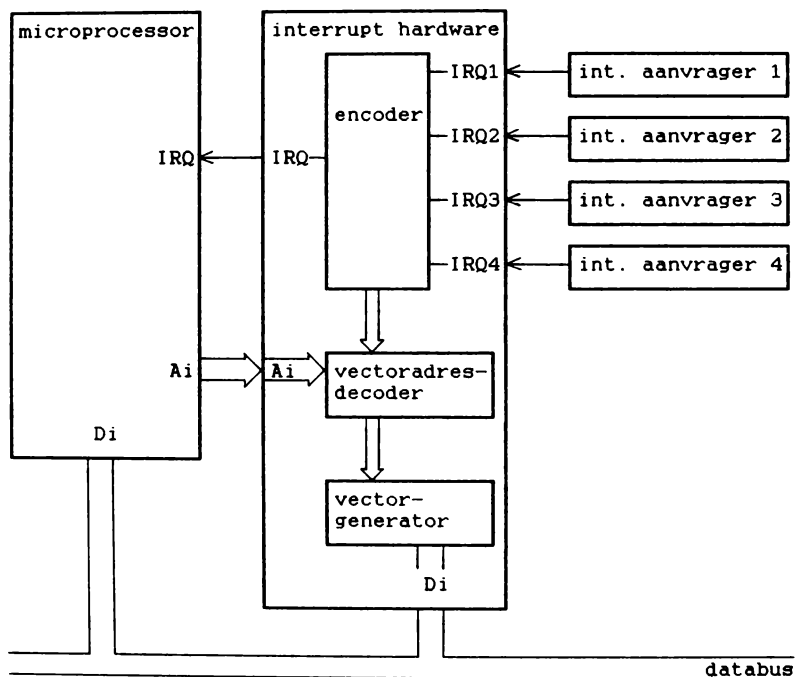


Fig. 9.19 Hardware vectoring, waarbij de interruptvector wordt geleverd door de externe interrupt hardware

Het genereren van de interruptvector

Het is ook mogelijk om in plaats van het vectoradres de inhoud van de interruptvector (of een gedeelte daarvan, bijvoorbeeld 1 byte) door externe hardware te laten vertalen. Dit kan worden

bereikt door met het vectoradres niet het geheugen, maar de externe hardware te activeren. Afhankelijk van de interruptaanvrager levert deze hardware dan het bijbehorende startadres (of een gedeelte daarvan) van de betreffende interrupt service routine. Dit wordt via de databus door de microprocessor binnengehaald (zie ook fig. 9.18 en 9.19). In dit geval bepaalt de interrupt hardware (tenminste voor een deel) op welke geheugenlocatie de ISR moet beginnen. Bij deze methode mag het geheugen niet worden geadresseerd door het vectoradres. Niet het geheugen, maar de interrupt hardware zet de interruptvector op de databus.

Interruptaanvrager levert vectorinformatie

Bij sommige processoren moet de interruptaanvrager zelf vectorinformatie leveren. In dit geval zendt de microprocessor na het detecteren van een interrupt een IACK-sigitaal (Interrupt ACKnowledge) naar de interruptaanvrager. Deze antwoordt hierop met het op de databus zetten van de vectorinformatie (zie fig. 9.20). De microprocessor haalt deze vectorinformatie vervolgens binnen en bepaalt hieruit het vectoradres. Dit betekent dat het vectoradres direct wordt aangewezen door de interruptaanvrager.

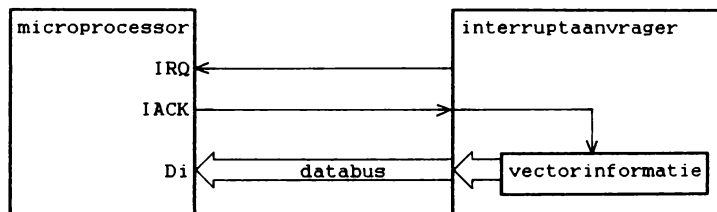


Fig. 9.20 De interruptaanvrager levert zelf vectorinformatie

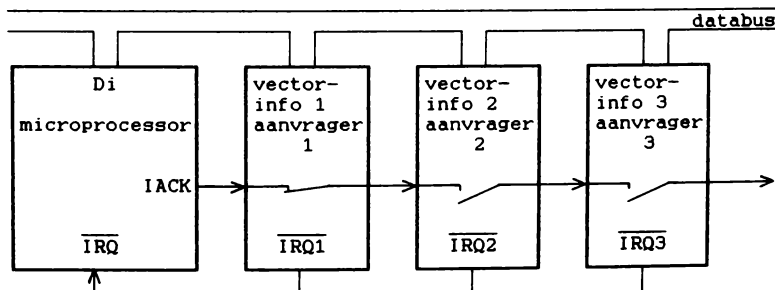


Fig. 9.21 Daisy-chain van interruptaanvragers

Hardware polling

Worden er meerdere interruptaanvragers op dezelfde interruptingang van de processor aangesloten, dan zijn er extra

voorzieningen nodig. Het IACK-sigitaal is slechts voor één aanvrager tegelijk geldig. De processor kan slechts één interrupt tegelijk afwerken. Er moet nu een schakeling zijn, die bepaalt naar welke aanvrager het sigitaal IACK moet gaan, met andere woorden, welke aanvraag het eerst behandeld zal worden. Dit gebeurt door de interruptaanvragers voor wat betreft het IACK-sigitaal in een zogenaamde daisy-chain te schakelen (zie fig. 9.21).

Bij deze daisy-chain wordt het sigitaal IACK in elke interruptaanvrager doorgelust naar de volgende aanvrager. Op deze wijze ontstaat er een keten van aanvragers. Nu kan elke aanvrager de IACK-keten verbreken. Dit gebeurt, indien de betreffende aanvrager een interrupt genereert. Deze interrupt wordt (via een wired-AND) doorgegeven aan de microprocesor, die hierop het sigitaal IACK afgeeft. Dit sigitaal bereikt aanvrager 1. Indien deze geen interrupt gegenereerd heeft, reageert deze hier niet op. De keten is hier niet verbroken, zodat het sigitaal IACK ook aanvrager 2 bereikt. Als deze wel een aanvraag heeft gedaan, reageert deze hierop door zijn vectorinformatie op de databus te zetten. Daar de interruptketen hier verbroken is, bereikt het sigitaal IACK aanvrager 3 (en eventuele volgende) niet meer. Ook al heeft aanvrager 3 eveneens een interrupt gegenereerd, hij zal zijn vectorinformatie niet op de databus kunnen zetten, omdat hij het sigitaal IACK niet ontvangen heeft. Pas als de interrupt van aanvrager 2 is afgehandeld, kan aanvrager 3 aan de beurt komen. Kennelijk heeft aanvrager 2 een hogere prioriteit dan aanvrager 3. U ziet in fig. 9.21, dat de prioriteit bepaald wordt door de plaats in de keten.

De benaming "daisy-chain" is ontstaan in verband met de analogie met een ketting van madeliefjes, waarbij elk madeliefje in de steel van zijn voorganger is gestoken.

Bij de daisy-chain kunnen we spreken van hardware polling. Hier wordt elke aanvrager afgetast door een sigitaal, in plaats van met behulp van instructies.

Hardware polling is snel, software polling is langzaam.

Daarentegen is software polling flexibel. Bij hardware polling ligt de prioriteit vast in de schakeling.

Hardware polling kan ook worden toegepast bij processoren die zelf geen IACK-uitgang hebben. In dit geval moet het sigitaal IACK extern worden gegenereerd. IACK kan bijvoorbeeld worden afgeleid uit het vectoradres (op de adresbus).

Software polling in combinatie met hardware vectoring

Er kan ook een combinatie worden gemaakt van hardware vectoring en software polling. Dit gebeurt wanneer de vector niet behoort bij een enkele aanvrager, maar bij een groep van aanvragers.

Met behulp van software wordt dan nagegaan om welke aanvrager

van de betreffende groep het gaat. Er hoeft nu slechts in de betreffende groep te worden gekeken, in plaats van bij alle aanvragers.

Interrupt levels

Er zijn microprocessoren met gecodeerde interruptingangen. Dit is bijvoorbeeld het geval bij de MC68000. Men spreekt hier van interrupt levels.

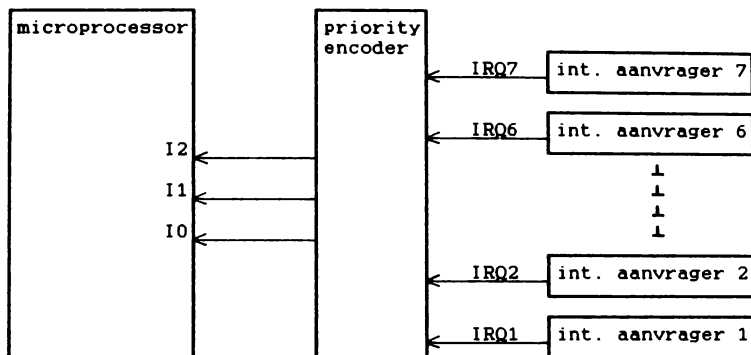


Fig. 9.22 Interrupt levels

IRQ7	IRQ6	IRQ5	IRQ4	IRQ3	IRQ2	IRQ1	I2	I1	I0	
0	0	0	0	0	0	0	0	0	0	geen interrupt
0	0	0	0	0	0	1	0	0	1	
0	0	0	0	0	1	X	0	1	0	X = don't care
0	0	0	0	1	X	X	0	1	1	
0	0	0	1	X	X	X	1	0	0	
0	0	1	X	X	X	X	1	0	1	
0	1	X	X	X	X	X	1	1	0	
1	X	X	X	X	X	X	1	1	1	

Fig. 9.23 Functietabel van priority encoder

Een interruptaanvraag wordt niet rechtstreeks aangesloten op zo'n gecodeerde interruptingang, maar via een priority encoder (zie fig. 9.22). Is er geen interruptaanvraag, dan is $I2=I1=I0=0$. De microprocessor gaat in dit geval gewoon door met het uitvoeren van het programma. Zodra er een interruptaanvraag is, wordt er een van 0 afwijkende code gegenereerd door de priority encoder. Deze code komt overeen met het nummer van de hoogste interruptaanvraag. Aanvraag 7 heeft de hoogste prioriteit en aanvraag 1 de laagste. Genereren bijvoorbeeld de aanvragers 1, 3 en 6 ongeveer gelijktijdig een interrupt, dan wordt $I2I1I0$ gelijk aan 110 (= 6), zodat de interrupt van aanvrager 6 het eerst wordt afgehandeld. De microprocessor

haalt de bijbehorende interruptvector op uit het geheugen.

Elk interrupt level heeft een eigen vectoradres. Zo kunnen zonder meer 7 verschillende ISR's worden uitgevoerd, zonder polling of vectoring (dus zonder tijdverlies en zonder externe interrupt hardware).

Zijn er meer interruptaanvragers dan interrupt levels, dan kunnen op de diverse ingangen van de priority encoder groepen van interruptaanvragers in wired-AND schakelingen worden opgenomen. Welke aanvrager binnen zo'n groep actief is, kan worden bepaald met behulp van software polling of hardware vectoring.

Bij elk level hoort dan een eigen IACK-sigitaal en er kan dan per IACK-sigitaal een daisy-chain worden gemaakt. Hierbij worden de interruptuitgangen van de aanvragers aangesloten op een (priority) encoder, die deze interrupts vertaalt in het juiste interrupt level.

Nu de principes van interrupts behandeld zijn, gaan we eens kijken welke soorten interrupts er zijn en waarvoor ze gebruikt kunnen worden. Hiertoe worden de interrupts van de MC6809 beschouwd.

9.7 MC6809-interrupts

9.7.1 Soorten interrupts

De MC6809 kent vier hardware interrupts en drie software interrupts:

- hardware interrupts:
 - restart interrupt, RESET
 - non-maskable interrupt, NMI
 - fast interrupt request, FIRQ
 - interrupt request, IRQ
- software interrupts:
 - SWI
 - SWI2
 - SWI3

9.7.2 Interrupts blokkeren

Tijdkritische programmadelen

Zoals reeds eerder werd opgemerkt, moeten interrupts kunnen worden geblokkeerd. Het kan immers zijn dat een bepaald gedeelte van het programma niet onderbroken mag worden in verband met de timing. Het blokkeren van de interrupts FIRQ en

IRQ gebeurt bij de MC6809 met behulp van de bits F, respectievelijk I van het condition code register (bit 6 en 4, zie paragraaf 3.6). Deze bits worden interrupt mask bits genoemd. Wordt een interrupt mask bit geset, dan wordt er als het ware voor de betreffende interruptingang een masker gezet, zodat de microprocessor een eventuele interruptaanvraag op die ingang niet ziet en er derhalve niet op reageert. De interrupts, die op die ingang binnenkomen, zijn dus geblokkeerd.

Voorkomen van continu herstarten van ISR (stack overflow)

Dit blokkeren van interrupts is ook noodzakelijk vanwege een andere reden. Stel er komt een interruptaanvraag binnen op de ingang IRQ. Op een bepaald moment wordt er met de uitvoering van de IRQ-ISR begonnen. Er worden dus wederom instructies uitgevoerd. De interruptaanvraag op de IRQ-ingang zal echter nog steeds aanwezig zijn. Dit betekent dat het programma (nu de IRQ-ISR) onmiddellijk weer wordt onderbroken door deze (zelfde) aanvraag. Er wordt opnieuw naar de IRQ-ISR gesprongen, maar deze wordt wederom onmiddellijk onderbroken. Het resultaat is dat er slechts achter elkaar wordt gesprongen naar het begin van de IRQ-ISR zonder dat er instructies worden uitgevoerd. Wel worden telkens de registerinhouden op de stack geplaatst, met het gevolg dat in zeer korte tijd de stack overschreden wordt. Er treedt een zogenaamde stack overflow op. Wordt dit niet door de hardware van de computer gedetecteerd en worden hierop geen maatregelen getroffen, dan zal er doorgedaan worden het gehele geheugen te overschrijven. Hierbij gaat alle informatie verloren.

Om dit te voorkomen maakt de microprocessor zelf de I-bit 1 bij het optreden van een IRQ-interrupt. Zo wordt bij het optreden van een FIQR-interrupt de F-bit automatisch geset.

Dit gebeurt nadat het condition code register op de stack is gezet, zodat bij de RTI-instructie, waarbij het condition code register vanaf de stack wordt geladen, de oude waarde van de interrupt mask bits weer hersteld wordt.

9.7.3 Restart interrupt

Inleiding

Bij het inschakelen van de voedingsspanning van een computersysteem komt er in het RAM-geheugen en in de flipflops van de microprocessor willekeurige informatie te staan. Dit betekent dat ook de inhoud van de program counter met willekeurige informatie wordt gevuld. De processor zal hierdoor op een willekeurige plaats in het geheugen een programma starten. Aangezien daar vrijwel zeker geen programma begint, zal dit leiden tot chaotische tafereelen. Er zal een mogelijkheid geschapen moeten worden om dit te voorkomen. Dit betekent dat de computer bij het inschakelen van de spanning moet worden voorzien van de juiste aanvangsinformatie, zodat er

een zinvol programma kan worden gestart (in het ROM-geheugen), bijvoorbeeld het operating system. Dit kan geschieden met behulp van de restart interrupt RESET en de bijbehorende interrupt service routine.

Werking

Wanneer de ingang RESET van de microprocessor (gedurende tenminste 1 klokcyclus) 0 wordt gemaakt, worden de activiteiten van de microprocessor onmiddellijk onderbroken. De inhoud van het direct page register wordt 0 gemaakt. Voorts wordt de NMI-flipflop gereset en geblokkeerd (zie paragraaf 9.7.4) en worden de bits I en F van het condition code register geset. Hiermee zijn alle andere hardware interrupts geblokkeerd.

De lopende instructie wordt niet afgemaakt bij een RESET. Terugspringen naar het onderbroken programma heeft geen enkele zin. De inhoud van het instructieregister, de informatie in de ALU, de toestand van de controller/sequencer, enzovoorts gaan verloren, omdat er naar een interrupt service routine wordt gesprongen, waar instructies uitgevoerd worden. Deze maken eveneens gebruik van instructieregister, ALU, controller/sequencer, enzovoorts. Een afgebroken instructie kan derhalve niet worden voortgezet. Vandaar dat het op de stack zetten van registers geen nut heeft en derhalve wordt nagelaten bij een RESET-interrupt. Terugkeren naar het onderbroken programma kan dus niet meer plaatsvinden, zodat de RESET-ISR niet wordt afgesloten met een RTI-instructie. Dit alles is minder dramatisch dan het lijkt, omdat het kwaad al heeft plaatsgevonden, voordat de RESET-interrupt optreedt. Denk bijvoorbeeld aan het niet aanwezig zijn van de juiste voedingsspanning, zodat de computer niet correct kan werken. Het kan ook zijn dat ten gevolge van een programmeerfout verkeerde dingen gebeuren, zodat het programma moet worden afgebroken om de zinloze acties te stoppen en om vervolgens het programma te vervolmaken.

De microprocessor staakt zijn activiteiten zolang de RESET-ingang 0 is.

Zodra RESET=1 wordt, haalt de microprocessor de RESET-interruptvector op (uit de geheugenlocaties \$FFFE en \$FFFF) en plaatst deze in de program counter. Hierdoor wordt er automatisch naar het beginadres van de RESET-ISR gesprongen, mits ervoor gezorgd is, dat op de locaties \$FFFE en \$FFFF het adres van deze ISR aanwezig is. Dit impliceert dat de twee genoemde geheugenlocaties van het ROM-type moeten zijn, omdat bij RAM willekeurige informatie aanwezig is na het inschakelen van de voedingsspanning.

In een RESET-ISR moet in ieder geval de stack pointer worden geïnitialiseerd. Bedenk hierbij dat bij het schrijven op de stack de stack pointer eerst met 1 wordt verlaagd. Dit betekent

dat de stack pointer moet worden geladen met het adres net één locatie boven de stack.

Vervolgens moet er naar een programma (in ROM) worden gesprongen. Dit kan zijn het operating system, maar ook een applicatieprogramma, bijvoorbeeld een verkeersregelprogramma.

Voorbeeld 9.6 (Operating system opstarten)

Stel dat na een RESET-interrupt het operating system moet worden gestart. Dit begint op adres \$F000. De stack moet beginnen op \$7FFF.

De RESET-ISR ziet er dan (in assembly) als volgt uit:

```
10          ORG      $FFFE
20 RESVEC   FDB      $FF00          initialiseer de RESET-vector
30 *
40          ORG      $FF00          beginadres van de RESET-ISR
50 RESISR   LDS      #$8000          initialiseer de stack pointer
60          LBRA     $F000          ga naar het operating system
70 *
80 *Einde van de RESET-ISR
```

Opstarten van computersysteem

Van de voedingsspanning wordt een RESET-signaal afgeleid, dat 0 is wanneer de voedingsspanning niet de juiste waarde heeft en dat 1 is als de voedingsspanning wel aan de voorgeschreven waarden voldoet. Op deze wijze wordt bij het inschakelen van de voeding automatisch een RESET-interrupt gegenereerd. In voorbeeld 9.6 heeft dit tot gevolg dat automatisch het operating system wordt opgestart.

Herstarten van computersysteem

Het kan gebeuren dat een programma door een storing of door een programmeerfout in een oneindige lus terecht komt. Dan kan zo'n programma worden onderbroken door met de hand een RESET-interrupt te veroorzaken. Hiertoe moet de RESET-lijn met behulp van een schakelaar 0 gemaakt kunnen worden. Doordat de RESET-ISR de besturing teruggeeft aan het operating system, kan de fout opgespoord worden, bijvoorbeeld met behulp van systeemprogramma's. Hierna kan het onderbroken programma opnieuw worden opgestart. Uitschakelen van de voedingsspanning zou tot gevolg hebben dat alle informatie in het RAM-geheugen vernield wordt, waardoor het opsporen van de fout bemoeilijkt kan worden.

De RESET-interrupt is uiterst belangrijk en niet maskeerbaar. Bovendien heeft hij de hoogste prioriteit. Hij onderbreekt derhalve alle andere interrupts. De RESET-ISR kan zelf niet worden onderbroken door een FIRQ- of een IRQ-interrupt, omdat de bits F en I geset zijn. Een NMI-interrupt kan de RESET-ISR wel onderbreken, echter pas nadat de stack pointer is geïnitieerd (zie paragraaf 9.7.4).

Fig. 9.24 toont het structogram van een RESET-ISR.

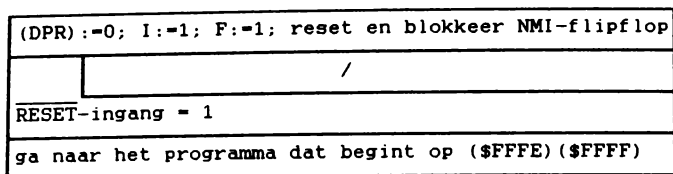


Fig. 9.24 Structogram van de processoracties bij een RESET

9.7.4 Non-maskable interrupt

Zoals de naam al aangeeft is ook de NMI-interrupt niet maskeerbaar. Deze interrupt heeft na de RESET-interrupt de hoogste prioriteit.

Werking

Wanneer de microprocessor een neergaande flank (een overgang van 1 naar 0) op zijn NMI-ingang detecteert, wordt de interne NMI-flipflop geset om deze flank te onthouden. De afhandeling van de interrupt vindt pas plaats nadat de executie van de huidige instructie is voltooid.

Na een RESET-interrupt gebeurt het zetten van de NMI-flipflop alleen als de stack pointer met behulp van een instructie van informatie is voorzien. Dit laatste is noodzakelijk omwille van het volgende. Bij een interrupt (behalve bij RESET) worden registerinhouden op de stack gered alvorens naar de interrupt service routine te springen. Hiertoe moet de stack pointer naar de stack wijzen. Na het inschakelen van de voedingsspanning bevat de stack pointer willekeurige informatie. Alvorens een interrupt te kunnen afhandelen, moet de stack pointer geïnitieerd zijn. Bij het opstarten (door RESET) worden de flags I en F geset, zodat de interrupts IRQ en FIRQ zijn geblokkeerd. De interrupt NMI is niet maskeerbaar, zodat de registerinhouden op willekeurige locaties zouden worden opgeborgen bij een NMI-interrupt die kort na het inschakelen van de voedingsspanning optreedt. Als op die locaties geen RAM aanwezig is, kan er niet worden teruggesprongen naar het onderbroken programma, omdat het terugkeeradres verloren is gegaan. Om dit te voorkomen, wordt de NMI-flipflop gereset en geblokkeerd door de RESET. Deze blokkering wordt pas opgeheven nadat de stack pointer is geladen met informatie (bijvoorbeeld met behulp van de instructie LDS).

NMI-interrupts die optreden voordat de stack pointer is geïnitieerd, gaan verloren. Het opstartprogramma moet de stack pointer laden met het topadres van de stack (+1).

Als de NMI-interruptvector niet in ROM staat, moet deze worden geïnitieerd voordat de stack pointer wordt geladen, omdat

er anders naar een willekeurig adres wordt gesprongen bij een NMI-interrupt.

Wat er bij een negatieve NMI-flank achtereenvolgens gebeurt, wordt getoond in het structogram van fig. 9.25.

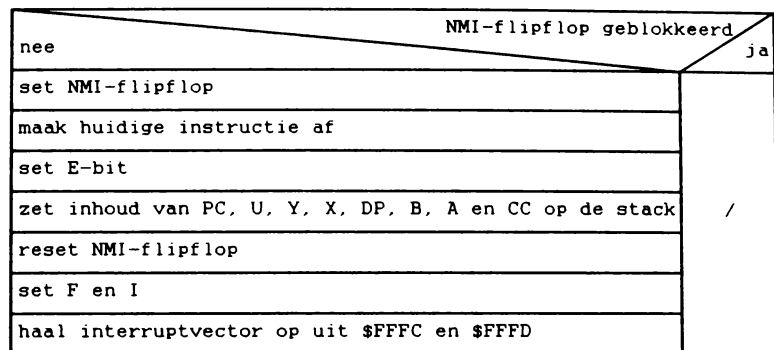


Fig. 9.25 Structogram van de processoractiviteit bij een NMI-interrupt

De E-bit is bit 7 van het condition code register. Deze Entire bit geeft aan dat alle microprocessorregisters op de stack opgeslagen zijn en dat ze derhalve bij de RTI-instructie weer van de stack moeten worden gehaald.

Door de bits F en I te setten voorkomt de microprocessor dat de NMI-ISR wordt onderbroken door een FIRQ- of door een IRQ-interrupt.

De ontwerper van de NMI-ISR kan natuurlijk besluiten om binnen deze interrupt service routine FIRQ- en/of IRQ-interrupts toe te laten. Dit kan hij verwezenlijken door de betreffende interrupt mask bit(s) te resetten.

F en I worden geset nadat het conditie code register op de stack is gezet. Bij terugkeer in het onderbroken programma (dus na het herstellen van de registerinhouden) hebben F en I hun oorspronkelijke waarde. Zouden F en I geset zijn voordat CC op de stack is gezet, dan zouden F en I bij terugkeer in het onderbroken programma 1 zijn. De interrupts FIRQ en IRQ zouden altijd geblokkeerd zijn na een NMI-interrupt.

Nadat de interruptvector in de program counter is opgeborgen, wordt het programma voortgezet met de eerste instructie van de NMI-ISR. Hiertoe moet de interruptvector van de juiste informatie zijn voorzien.

1. Stack overflow

De NMI-interrupt kan zichzelf niet onmiddellijk onderbreken, doordat deze alleen reageert op een flank op de NMI-ingang. Na de neergaande flank zal het signaal eerst weer omhoog moeten gaan, alvorens er een volgende neergaande flank kan ontstaan. Een interruptaanvrager mag niet zeer snel achter elkaar flanken naar de microprocessor sturen, omdat er dan stack overflow kan optreden.

Remedie:

Het achter elkaar genereren van NMI-interrupts door dezelfde aanvrager, voordat zijn vorige aanvraag is afgehandeld, kan worden voorkomen door de aanvrager terug te melden, wanneer hij een volgende interrupt mag genereren.

2. Missing interrupts

Zolang een aanvrager de NMI-lijn 0 houdt, worden alle NMI-interrupts van andere aanvragers geblokkeerd. Deze interrupts gaan bovendien verloren, omdat de negatieve flanken niet kunnen optreden, zolang de NMI-lijn 0 is. Om de kans klein te maken interrupts te missen bij meerdere NMI-aanvragers, moet de NMI-lijn niet te lang 0 worden gehouden.

De processor kan een volgende NMI-interrupt pas opslaan in de NMI-flipflop, zodra hij de registerinhouden op de stack heeft gezet. Immers dan wordt de NMI-flipflop gereset, zodat deze opnieuw kan worden geset. Hierdoor voert de processor na het ophalen van de NMI-interruptvector wederom de NMI-activiteit uit (zie fig. 9.25).

Gemist worden alle NMI-interrupts die optreden tussen het detecteren van de eerste NMI-interrupt en het resetten (door de processor) van de NMI-flipflop. De tijd gedurende welke de NMI-interrupts worden gemist, hangt af van de duur van de instructie, tijdens welke de eerste NMI-interrupt optrad.

Remedie:

Een NMI-interruptaanvrager mag pas opnieuw een interrupt genereren, nadat hij van de processor toestemming heeft gekregen.

Bij meerdere NMI-aanvragers moet de duur van de aanvraag korter zijn dan de tijd die de processor nodig heeft om de NMI-activiteit uit te voeren (registers redden, enzovoorts). Dan worden alle actieve aanvragen gedetecteerd bij de polling. Er moet hierbij gebruik worden gemaakt van software polling en wel zodanig dat alle actieve interrupts in één ISR-doorgang worden afgehandeld (linker structogram van fig. 9.15). Na terugkeer in het onderbroken programma is de NMI-flipflop gereset (tenzij er inmiddels een nieuwe aanvraag is), zodat de niet-afgehandelde oude aanvragen verloren zijn gegaan. Bij hardware polling en bij hardware vectoring wordt slechts één interruptaanvraag per ISR-doorgang afgehandeld, zodat de overige aanvragen verloren gaan.

3. Softwareproblemen bij meerdere NMI-aanvragers
Wordt er een NMI-interrupt gegenereerd tijdens het uitvoeren van de NMI-interrupt service routine, dan wordt de NMI-ISR onderbroken en opnieuw opgestart vanaf het begin. Hierbij ontstaan er softwareproblemen, tenzij er speciale maatregelen zijn getroffen. In dit geval moet de NMI-ISR reentrant zijn. Reentrant routines worden behandeld in paragraaf 12.5.
Remedie: Reentrant ISR.

4. Dubbel afhandelen van interrupt

Bij meerdere NMI-aanvragers ontstaat het probleem, dat een bepaalde interrupt meerdere malen wordt afgehandeld. Stel dat aanvrager 1 een interrupt genereert. Bij software polling wordt ontdekt wie de interrupt heeft gegenereerd. Juist voordat actie 1 (zie fig. 9.15) wordt uitgevoerd, wordt een tweede NMI-interrupt gedetecteerd (afkomstig van bijvoorbeeld aanvrager 2). De NMI-ISR wordt onderbroken en opnieuw opgestart. Weer wordt ontdekt dat aanvrager 1 een interrupt heeft gegenereerd, zodat actie 1 (en eventueel actie 2) wordt uitgevoerd. Er wordt teruggekeerd naar het onderbroken programma (de NMI-ISR), zodat actie 1 nogmaals wordt uitgevoerd.

Het is dus van belang dat de interruptaanvrager zo snel mogelijk de opdracht krijgt zijn interruptbit te resetten, opdat dit niet twee keer achter elkaar wordt gedetecteerd bij software polling. De kans dat dit gebeurt, is bij software polling niet 0 te maken.

Bij hardware vectoring heeft elke aanvrager een eigen interruptvector en dus een eigen ISR, zodat dezelfde interrupt nooit meerdere malen wordt afgehandeld. Nu treedt echter het probleem van de missing interrupts weer op.

Remedie:

Semaforen toepassen, zodat een bepaalde activiteit binnen een ISR pas opnieuw kan worden gestart, nadat de vorige uitvoering voltooid is. Semaforen komen ter sprake in paragraaf 11.3.

Toepassing

Een NMI-interrupt kan gebruikt worden voor zeer belangrijke aangelegenheden, die geen uitstel mogen hebben. Denk bijvoorbeeld aan het adresseren van locaties waar zich geen geheugen of I/O bevindt. Als dit wordt gedetecteerd door een externe schakeling, moet er onmiddellijk worden gestopt, omdat voortzetten van het programma tot ernstige fouten kan leiden. Door het binnenhalen van verkeerde gegevens kan het programma geheel in de war raken.

9.7.5 Fast interrupt request

Toepassing

Bij het uitvoeren van interrupts hebben we te maken met een hoeveelheid overhead. De processor voert allerlei acties uit, voordat er met de eigenlijke interrupt service routine wordt

begonnen. Zo worden diverse bits van het condition code register gewijzigd, registerinhouden worden op de stack geplaatst en de interruptvector wordt uit het geheugen opgehaald. De naam fast interrupt request suggereert dat het hier gaat om een snelle interrupt. Deze verhoging van snelheid wordt gerealiseerd door alleen de inhoud van program counter en condition code register op de stack te zetten. Dit betekent dat bij de RTI-instructie alleen deze twee registers van de stack moeten worden gehaald. Er treedt dus twee keer een tijdswinst op.

De FIRQ-interrupt wordt gebruikt, indien er een zeer snelle interruptafhandeling moet plaatsvinden, of indien er interrupts van verschillende prioriteit voorkomen.

Werking

Er wordt een FIRQ gegenereerd door de $\overline{\text{FIRQ}}$ -ingang van de microprocessor gelijk aan 0 te maken. Deze FIRQ wordt alleen gedetecteerd indien F=0.

Wat er hierna achtereenvolgens gebeurt, wordt getoond in het structogram van fig. 9.26.

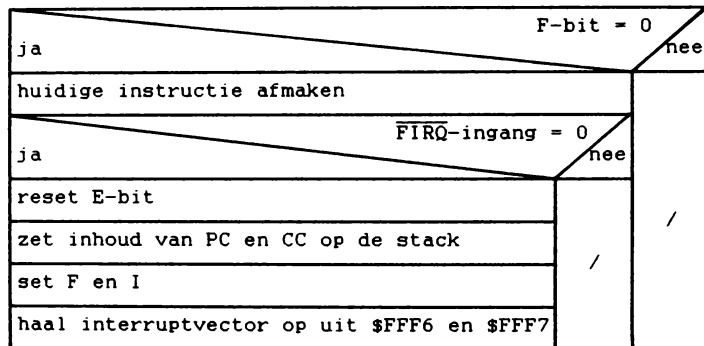


Fig. 9.26 Structogram van de processoractiviteit bij een FIRQ-interrupt

U ziet dat de FIRQ-interrupt alleen wordt afgehandeld, als de ingang $\overline{\text{FIRQ}}$ nog 0 is na het uitvoeren van de huidige instructie.

De E-bit wordt gereset om aan te geven dat alleen (PC) en (CC) op de stack worden opgeborgen, zodat alleen deze twee registers bij de RTI-instructie van de stack moeten worden gehaald.

Worden binnen de FIRQ-Interrupt Service Routine andere registers dan CC en PC gebruikt, dan moeten die alsnog met behulp van een push-instructie worden gered aan het begin van de ISR. Aan het einde van de ISR moeten ze worden hersteld met behulp van een pull-instructie. De registerinhouden mogen niet

worden gewijzigd door de FIRQ-ISR, omdat een interrupt in principe op elke willekeurig moment kan optreden. Het onderbroken programma kan derhalve zelf niet voor het redden van registerinhouden zorgdragen.

De F-bit wordt geset om te voorkomen dat de FIRQ-interrupt onmiddellijk weer een interrupt zou veroorzaken, enzovoorts. Hierdoor zou stack overflow ontstaan.

De I-bit wordt geset om te voorkomen dat een IRQ-interrupt (die een lagere prioriteit heeft) de FIRQ-ISR onderbreekt.

Natuurlijk kan de programmeur een IRQ-interrupt toelaten door de I-flag te resetten.

Zo ook kan hij een tweede FIRQ-interrupt toestaan door de F-bit te resetten. Dit laatste mag echter pas gebeuren nadat de FIRQ-ingang weer 1 is geworden, anders zou er stack overflow optreden! Wanneer de FIRQ-ISR door een tweede FIRQ wordt onderbroken, wordt de FIRQ-ISR opnieuw opgestart, zonder dat deze volledig is uitgevoerd. Opdat hierbij geen softwarefouten optreden, moeten er speciale maatregelen worden getroffen. Men zegt dat de routine "reentrant" moet zijn. Hierover echter meer in paragraaf 12.5.

Voorbeeld 9.7 (Real Time Clock)

Als voorbeeld van het gebruik van een interrupt service routine tonen we een zogenaamde Real Time Clock.

Veronderstel dat van de 50 Hz netspanning een digitaal signaal is afgeleid, dat aangesloten is op de FIRQ-ingang van de microprocessor. Zo ontstaat er telkens na 20 ms een neergaande flank op deze ingang en derhalve een interrupt. Hierop moet de microprocessor ervoor zorgen dat de tijd bijgewerkt wordt. Deze tijd wordt uitgedrukt in uren, minuten, seconden en tienden van seconden. Deze staan op bepaalde geheugenlocaties en kunnen door andere programma's worden gebruikt, bijvoorbeeld voor het doen van tijdmetingen. Het bijwerken van de tijd gaat geheel buiten de andere programma's om en speelt zich af binnen de FIRQ-ISR.

Het structogram voor deze ISR wordt gegeven in fig. 9.27, gevolgd door de bijbehorende FIRQ-ISR RTC.

Eventueel kan het structogram worden verkort door een herhalingsstructuur toe te passen. Dit is omwille van de overzichtelijkheid niet gedaan.

Let op dat bij de FIRQ-ISR alleen (PC) en (CC) gered worden. (A) moeten we in de FIRQ-ISR zelf redden (regel 250) en herstellen (regel 500).

Aangenomen is dat het operating system gezorgd heeft voor het instellen van de juiste tijd op de geheugenlocaties TIENDE, SEC, MIN en UUR en voor het opbergen van de waarde 5 op de locatie TELLER. Voorts nemen we aan dat het operating system de FIRQ-interruptvector van de juiste waarde (RTC = \$B000, zie regel 180) heeft voorzien.

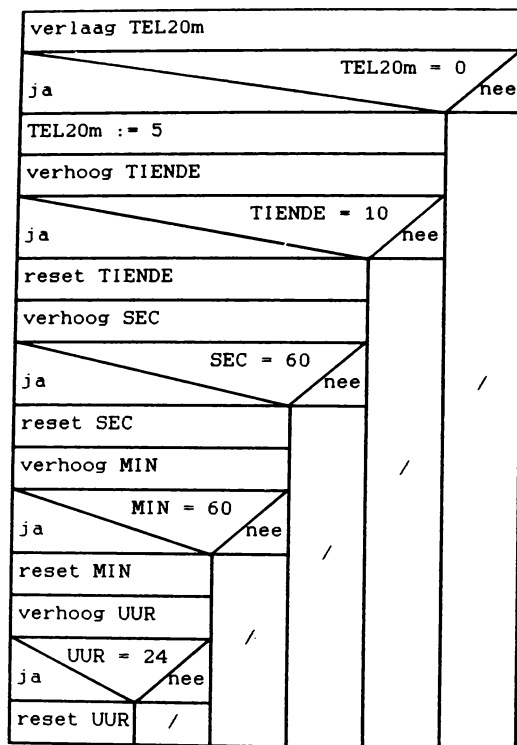


Fig. 9.27 Structogram voor Real Time Clock

```

10 *****
20 * FIRQ-ISR Real Time Clock *
30 *****
40 *
50 Progr.: J.G. Rouland
60 Revisie 0      Datum: 08-08-1986
70 *
80 *Op de FIRQ-ingang van de MC6809 is een afgeleide van de
90 *netspanning aangesloten, die elke 20 ms een interrupt
100 *veroorzaakt.
110 *De ISR werkt hierop de tijd bij in uren (tot maximaal 24),
120 *minuten, seconden en tienden van seconden.
130 *Deze staan achtereenvolgens in de geheugenlocaties met de
140 *adressen $A000, $A001, $A002 en $A003.
150 *Locatie $A004 wordt door de ISR als teller gebruikt.
160 *
170 *
180      ORG      $B000      beginadres van FIRQ-ISR

```

190	UUR	EQU	\$A000	
200	MIN	EQU	\$A001	
210	SEC	EQU	\$A002	
220	TIENDE	EQU	\$A003	
230	TELLER	EQU	\$A004	
240	*			
250	RTC	PSHS	A	red register A
260		DEC	TEL20m	er zijn 20 ms verstreken
270		BNE	EINDE	spring als geen 100 ms voorbij
280		LDA	#5	initialiseer de
290		STA	TELLER	20ms-afteller
300		INC	TIENDE	er is 0.1 sec verstreken
310		LDA	TIENDE	
320		CMPA	#10	
330		BNE	EINDE	spring als geen sec. voorbij is
340		CLR	TIENDE	reset tiende-secondenteller
350		INC	SEC	er is 1 sec verstreken
360		LDA	SEC	
370		CMPA	#60	
380		BNE	EINDE	spring als geen min. voorbij is
390		CLR	SEC	reset secondenteller
400		INC	MIN	er is 1 min verstreken
410		LDA	MIN	
420		CMPA	#60	
430		BNE	EINDE	spring als geen uur voorbij is
440		CLR	MIN	reset minutenteller
450		INC	UUR	er is 1 uur verstreken
460		LDA	UUR	
470		CMPA	#24	
480		BNE	EINDE	spring als geen etmaal voorbij
490		CLR	UUR	er zijn 24 uur verstreken
500	EINDE	PULS	A	herstel (A)
510		RTI		terug naar onderbroken programma

9.7.6 Interrupt request

Door de $\overline{\text{IRQ}}$ -ingang gelijk aan 0 te maken, terwijl $\text{I}=0$ is, wordt een interrupt request gedetecteerd. Fig. 9.28 laat zien welke activiteiten de microprocessor vervolgens onderneemt.

De IRQ-interrupt wordt alleen afgehandeld als de $\overline{\text{IRQ}}$ -ingang van de microprocessor nog 0 is, nadat de huidige instructie is uitgevoerd.

De E-bit wordt geset om aan te geven dat alle registers op de stack worden gezet.

De I-bit wordt geset om te voorkomen dat er stack overflow optreedt.

De IRQ-interrupt wordt veelal gebruikt voor communicatie met externe apparatuur (zie hoofdstuk 10).

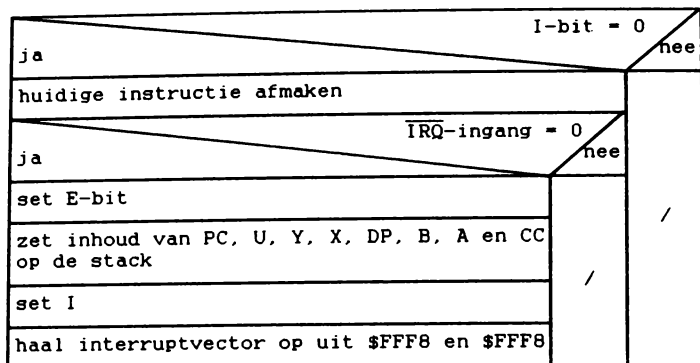


Fig. 9.28 Structogram voor de processoractiviteit bij een IRQ-interrupt

9.7.7 Software interrupts

Software interrupts worden gebruikt voor het aanroepen van speciale programma's, vaak routines die deel uitmaken van het operating system. Ze worden in het laatste geval ook wel system calls genoemd.

Een software interrupt ontstaat bij het uitvoeren van de instructie SWI, SWI2 of SWI3. De figuren 9.29 t/m 9.31 tonen de processoractiviteit bij de diverse software interrupts.

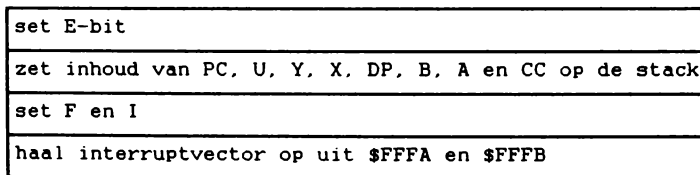


Fig. 9.29 Structogram voor de processoractiviteit na het detecteren van een SWI-interrupt

U ziet dat de bits F en I geset worden. Dit betekent dat een SWI-interrupt een hogere prioriteit heeft dan een FIRQ- en een IRQ-interrupt, immers deze interrupts kunnen een SWI-ISR niet onderbreken.

Overigens kan de programmeur binnen de SWI-ISR de interrupt mask bits resetten en zodoende de bijbehorende interrupts toch doorlaten.

set E-bit
zet inhoud van PC, U, Y, X, DP, B, A en CC op de stack
haal interruptvector op uit \$FFF4 en \$FFF5

Fig. 9.30 Structogram voor de processoractiviteit na het detecteren van een SWI2-interrupt

set E-bit
zet inhoud van PC, U, Y, X, DP, B, A en CC op de stack
haal interruptvector op uit \$FFF2 en \$FFF3

Fig. 9.31 Structogram voor de processoractiviteit na het detecteren van een SWI3-interrupt

De interrupts SWI2 en SWI3 hebben de laagste prioriteit. Zij zetten geen van de interrupt mask bits.

In feite is er, op het vectoradres na, geen verschil tussen SWI2 en SWI3. De fabrikant (Motorola Inc.) zegt echter dat SWI2 bedoeld is voor de eindgebruiker en niet moet worden toegepast in softwarepakketten, die door anderen worden gebruikt.

Voorbeeld 9.8 (Het terugkeren naar het operating system)

In dit boek is SWI steeds gebruikt om een programma te beëindigen. In feite werd hiermee de besturing teruggegeven aan het operating system. De bijbehorende SWI-interrupt service routine ziet er als volgt uit, aangenomen dat het operating system op adres \$E000 begint.

```

STOP    LEAS    12,S      (S) := (S) + 12
        JMP     $E000    start het operating system op
*Einde van SWI-ISR

```

Het beginadres van de SWI-ISR (STOP) moet natuurlijk op de geheugenplaatsen \$FFFA en \$FFFB staan.

Er wordt niet teruggekeerd naar het programma, omdat we dit juist willen stoppen aan het einde. Derhalve zal de SWI-ISR in dit geval niet eindigen met RTI, maar met een jump- of branch-instructie naar het operating system. Zodoende worden de registerinhouden niet van de stack gehaald. Ze zijn trouwens niet meer nodig. Het programma is immers beëindigd. Wel moet de stack pointer worden hersteld. Hij moet wijzen naar dezelfde stacklocatie als vlak voor het uitvoeren van de instructie SWI. Dit betekent dat de stackpointer met 12 moet worden verhoogd. Bij SWI zijn 12 bytes op de stack opgeborgen (zie ook fig. 9.3). Zou dit verhogen van de stack pointer achterwege blijven, dan zal de stack steeds voller raken, naarmate er meer

programma's achter elkaar worden uitgevoerd en gestopt met behulp van deze SWI-ISR.
 Bij dit alles is er van uit gegaan dat het operating system een applicatieprogramma start met behulp van een jump of branch naar het beginadres van het programma.

Wordt een applicatieprogramma gestart met behulp van een JSR- of BSR-instructie, dan kan eenvoudig met de instructie RTS worden teruggekeerd naar het operating system. Er wordt dan teruggekeerd in het operating system op de locatie na de subroutine call (van het applicatieprogramma).
 Willen we het operating system in dit geval opnieuw opstarten, dan moet het applicatieprogramma eindigen met het verhogen van de stack pointer met 2 in verband met het terugkeeradres (2 bytes), dat op de stack is gezet bij de subroutine call.
 Vervolgens wordt er met een JMP- of BRA-instructie gesprongen naar het begin van het operating system.

System calls

Om met één software interrupt verschillende functies te kunnen aanroepen, moet er informatie omtrent de gewenste functie meegegeven worden, bijvoorbeeld een functienummer. Dit kan geschieden via

- een MPU-register
- een vaste geheugenlocatie
- een stacklocatie
- de geheugenlocatie (in het programmeergeheugen) die volgt op de betreffende SWI-instructie

Overdracht via een pointer is niet zinvol, omdat het hier meestal om één byte gaat. Met één byte kunnen 256 verschillende functies worden onderscheiden!

De meest eenvoudige wijze van de overdracht van het functienummer vindt plaats via een register. Stel het functienummer bevindt zich in accumulator A. Door een tabel FUNTAB te maken met de startadressen van de functies, kan de SWI-ISR er als volgt uit zien:

```

100 *****
110 * SWI-ISR FUNCTIE CALL *
120 *****
130 *
140 *SWI wordt gebruikt om een functie aan te roepen.
150 *A moet het nummer van de gewenste functie bevatten.
160 *
170 *
180 FUNTAB   FDB      FUN0,FUN1,FUN2,...   tabel met startadressen
190 *
200 CALFUN   LDX      #FUNTAB              initialiseer de adrespointer
210         ASLA                      bepaal de offset
220         BCC      LOWADR              spring als nummer < 128
    
```

```

230          LEAX    128,X          (X) := (X) + 128
240 LOWADR   BSR     [A,X]         roep de gewenste functie aan
250          RTI                    terug naar onderbroken progr.
260 *
270 FUN0     --
280          --
290          RTS
300 FUN1     --
310          --
320          RTS
enzovoorts

```

Daar een adres uit 2 bytes bestaat, moet (A) eerst met twee worden vermenigvuldigd (regel 210) om het juiste adres aan te wijzen. Het startadres van de gewenste functie bevindt zich op de locatie met het adres (A)+(X). Vandaar dat met een indexed indirecte call de gewenste subroutine wordt aangeroepen.

Staat het functienummer op de geheugenlocatie FUNNUM, dan moet regel 205 worden tussengevoegd:

```

205          LDA     FUNNUM         laad het functienummer

```

Is het functienummer in het aanroepende programma op de stack gezet (met behulp van een PSHS-instructie), dan wordt regel 205 als volgt:

```

205          LDA     12,S          laad het functienummer

```

Bij de software interrupt zijn 12 bytes (van de registerinhouden) op de stack gezet. Het functienummer is in de SWI-ISR derhalve te bereiken via offset 12 ten opzichte van de stack pointer (zie ook fig. 9.3).

Teruggekeerd in het aanroepende programma moet de stack pointer met 2 worden verhoogd, opdat de stack nietodeloos vol raakt. De stack pointer moet telkens zijn oorspronkelijke waarde terugkrijgen.

Plaatsen we het functienummer als een constante in het programma, direct achter de instructie SWI, dan wordt de zaak complexer. Stel we willen een systeemfunctie aanroepen om een karakter vanaf het toetsenbord in te voeren. Het nummer van deze functie is bijvoorbeeld 31. Dit gaat dan als volgt:

```

10          --
20 INCHAR   EQU     31             functienummer van INCHAR
30          --
40 PROGR    --
50          --
60          SWI     INCHAR         lees karakter van keyboard
70 VERDER   --                   vervolg het programma
80          --

```

In de SWI-ISR wordt regel 205 nu:

```

205          LDA     [10,S]

```

immers de locaties (S)+10 en (S)+11 bevatten de geredde inhoud van de program counter, zijnde het adres van de locatie direct achter SWI, dus de locatie van het functienummer. Bedenk dat (S)+0 het adres van de geredde inhoud van CC is (zie ook fig. 9.3).

Aan het einde van de ISR wordt via de instructie RTI (regel 250) teruggekeerd naar het aanroepende programma PROGR. De program counter wordt vanaf de stack gevuld. Daar staat in dit geval het adres van het functienummer. De microprocessor zal dit als een opcode interpreteren, zodat het programma foutief zal aflopen. De eerstvolgende instructie na SWI staat één locatie verder. Dit betekent dat het terugkeeradres met 1 verhoogd moet worden. Dit moet gebeuren voordat de program counter van de stack wordt geladen. Derhalve moet de inhoud van de betreffende stacklocatie met 1 worden verhoogd, voordat de instructie RTI wordt uitgevoerd.

Dit kan geschieden met behulp van de instructies:

206	INC	11,S	verhoog low-byte terugkeeradres
207	BNE	MAAL2	spring als C=0
208	INC	10,S	verhoog high-byte terugkeeradres
210	MAAL2	ASLA	bepaal de offset

Nu wordt PROGR voortgezet met de instructie, die volgt op SWI (op het adres VERDER).

In regel 207 kan niet de instructie BCC worden toegepast, omdat de increment-instructie de carry ongewijzigd laat (zie appendix B). Er ontstaat alleen een carry als het getal \$FF wordt verhoogd met 1. De uitkomst is dan \$00, zodat er een carry is als de uitkomst 0 wordt.

Het functienummer is te vergelijken met de vectorinformatie bij de hardware interrupts. In de SWI-ISR wordt (softwarematig) uit het functienummer bepaald om welke functie het gaat.

Indien de betreffende functie gegevens nodig heeft voor de uitvoering, dan kunnen deze op diverse wijzen aan de ISR worden doorgegeven. Zie hiervoor paragraaf 9.8.

9.7.8 Return from interrupt

Elke interrupt service routine, uitgezonderd de RESET-ISR, moet worden beëindigd met de RTI-instructie. Zoals u in voorbeeld 9.8 hebt gezien, kan een interrupt (bijvoorbeeld SWI) worden gebruikt om terug te keren naar het operating system. Het programma wordt hierbij voorgoed verlaten. In zo'n geval ontbreekt de instructie RTI. Wel moet de stack pointer worden aangepast voor het vrijgeven van de ruimte, waar de geredde registerinhouden staan. Dit betekent dat S moet worden verhoogd met 3 bij een FIRQ en met 12 bij elke andere interrupt, die

wordt gebruikt om het programma voorgoed te verlaten.

Bij de RTI-instructie worden de MPU-registers hersteld en wordt er teruggekeerd naar het onderbroken programma (zie fig. 9.32).

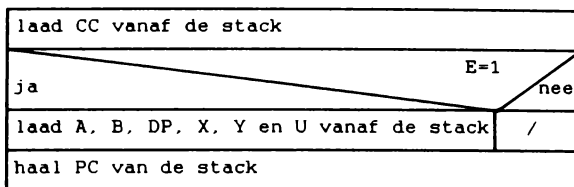


Fig. 9.32 Structogram behorende bij de RTI-instructie

Eerst wordt het conditie code register vanaf de stack geladen. De inhoud van dit register moet altijd worden gered en wel om de volgende reden:

In een interrupt service routine wordt de inhoud van CC gewijzigd, omdat vrijwel elke instructie een of meerdere bits van CC wijzigt (zie appendix B). Bovendien worden de bits I en/of F bij een aantal interrupts geset om stack overflow te voorkomen of om interrupts met een lagere prioriteit te blokkeren. Na terugkeer in het onderbroken programma moet CC dezelfde informatie bevatten als op het moment van de interrupt. Het programma kan (CC) niet redden, omdat de interrupt op elk moment kan optreden. Vandaar dat de processor (CC) automatisch op de stack zet bij het optreden van een interrupt (behalve bij RESET).

Na het laden van CC wordt de Entire-bit getest.

Als E=1 is, zijn bij de laatste interrupt alle registers gered en moeten ook alle registerinhouden weer worden hersteld.

Als E=0 moet alleen de inhoud van de program counter worden hersteld. Het condition code register is inmiddels al geladen! De program counter moet bij elke interrupt (behalve bij RESET) worden gered, omdat deze het terugkeeradres bevat.

De E-bit wordt bij een interrupt eerst van de juiste waarde voorzien en daarna wordt (CC) op de stack gezet. Zodoende blijft deze waarde van E gehandhaafd bij het uitvoeren van de RTI-instructie.

Zou eerst (CC) zijn opgeborgen op de stack en daarna E aangepast, dan wordt de waarde van E (die aangeeft welke registerinhouden moeten worden hersteld) aan het begin van de uitvoering van RTI overschreven en zou E in een aparte flipflop in de processor moeten worden opgeborgen. Vandaar dat eerst E wordt ingesteld en daarna (CC) op de stack wordt gezet.

9.7.9 Interruptvectoren

Fig. 9.33 toont de interruptvectoren van de MC6809-microprocessor.

interrupt	vectoradres	
reserved	\$FFF0, \$FFF1	(gereserveerd voor eventuele uitbreiding in de toekomst)
SWI3	\$FFF2, \$FFF3	
SWI2	\$FFF4, \$FFF5	
FIRQ	\$FFF6, \$FFF7	
IRQ	\$FFF8, \$FFF9	
SWI	\$FFFA, \$FFFB	
NMI	\$FFFC, \$FFFD	
RESET	\$FFFE, \$FFFF	

Fig. 9.33 Overzicht van de interruptvectoradressen

In paragraaf 9.7.3 hebt u gezien dat de interruptvector van de RESET-interrupt in ROM moet staan, omdat deze meteen na het inschakelen van de voedingsspanning de juiste waarde moet hebben. Aangezien een ROM-IC meer dan 2 bytes bevat, bestaat hierdoor het bovenste deel van een MC6809-systeem uit ROM. Dit houdt in dat ook de andere interruptvectoren in ROM staan. De waarde van deze vectoren moet door de applicatieprogrammeur kunnen worden veranderd, omdat deze zijn eigen interrupt service routines moet kunnen schrijven. De lengte van deze routines is niet van tevoren bekend. Vandaar dat de plaats waar deze routines worden opgeborgen, eveneens onbekend is. Dit betekent dat de interruptvectoren in RAM moeten staan. Aan deze strijdige eisen kan op twee manieren worden voldaan.

1. Hardware oplossing:

RESET-vector in ROM en andere vectoren in RAM

In de bovenste adresruimte van het geheugen bevindt zich zowel ROM als RAM. Deze zijn als het ware parallel geschakeld. Met behulp van de adresdecoders wordt ervoor gezorgd dat de ROM alleen wordt geselecteerd voor de adressen \$FFFE en \$FFFF, terwijl de RAM alleen voor de overige adressen wordt geselecteerd (zie fig. 9.34).

Zoals u in fig. 9.34 ziet, wordt de ROM alleen geadresseerd als A1 t/m A15 gelijk aan 1 zijn. Alleen in dat geval is de ingang CS (Chip Select) van de ROM 1 en zijn de uitgangen D0 t/m D7 van de ROM geactiveerd. Is CS=0, dan zijn de uitgangen D0 t/m D7 hoogimpedant (zwevend) en derhalve van de databus afgeschakeld.

De CS-ingang van de ROM is dus alleen 1 voor de adressen \$FFFE en \$FFFF. Op deze locaties bevindt zich de RESET-interruptvector. De waarde van A0 bepaalt of de high-byte (A0=0) van de interruptvector wordt opgehaald of de low-byte (A0=1).

De CS-ingang van de RAM is 1 als A13=A14=A15=1, mits niet tevens geldt dat A1 t/m A15 gelijk aan 1 zijn. Dit geldt dus voor de adressen \$E000 t/m \$FFFD. De waarde van de RAM-ingangen A0 t/m A12 bepaalt welke van de 8k bytes wordt geselecteerd. Met behulp van de adresdecoders wordt voorkomen dat meerdere schakelingen tegelijk data op de databus gaan zetten. Dit zou leiden tot ongedefinieerde niveaus en tot hoge stromen in de schakelingen. De ene schakeling zou een hoog niveau (1) op een bepaalde buslijn kunnen zetten, terwijl een andere schakeling deze lijn laag (0) wil maken.

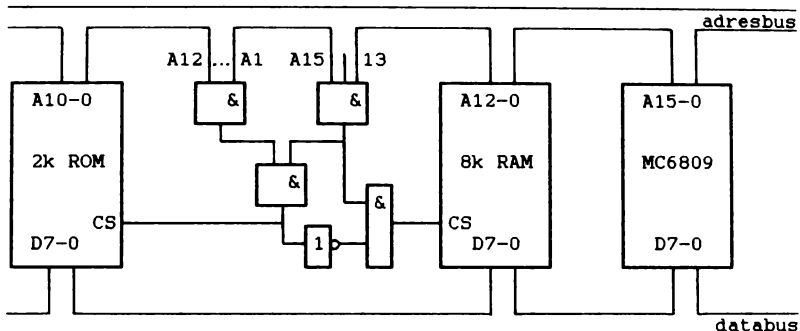


Fig. 9.34 ROM en RAM voor de interruptvectoren

In plaats van een ROM toe te passen voor het opslaan van de RESET-interruptvector, kunnen 16 schakelaars worden gebruikt. De adresdecoders blijven hierbij gelijk aan die van fig. 9.34. Met A0=0 wordt de high-byte van de schakelaars geselecteerd en met A0=1 de low-byte, mits tegelijkertijd A1 t/m A15 gelijk aan 1 zijn.

Het voordeel van deze werkwijze is dat de RESET-interruptvector nu ingesteld kan worden op elke gewenste waarde.

2. Software oplossing:

Interrupt vectoren verplaatsen naar RAM

In dit geval is de hoogste adresruimte gevuld met ROM, waarin zich doorgaans het operating system of het monitorprogramma bevindt. Vanzelfsprekend bevinden zich de interruptvectoren boven in. Dit is immers in de hardware van de MC6809-microprocessor zo vastgelegd.

Bij het opstarten wordt via de RESET-interruptvector naar het begin van het operating system gesprongen. Het operating system moet de mogelijkheid scheppen dat de gebruiker zelf interrupt service routines kan schrijven met een willekeurig startadres. Om dit te bewerkstelligen is het eerste deel van elke ISR in het operating system opgenomen. Hierin wordt gesprongen naar het RAM, waar de ISR wordt voortgezet. Door nu een indirecte spronginstructie te kiezen, is het beginadres van de routine in

RAM een variabele, die door de gebruiker kan worden veranderd.

Veronderstel dat het operating system zich bevindt in een 8k ROM (op de adressen \$E000 t/m \$FFFF). De interruptvectoren staan op \$FFF0 t/m \$FFFF (zie fig. 9.33).

Direct onder de ROM bevindt zich RAM. Laten we de startadressen van de "user-ISR's" boven in het RAM zetten, dus op de locaties \$DFF0 t/m \$DFFF, in dezelfde volgorde als in de ROM (op \$FFF0 t/m \$FFFF).

De stack zetten we direct onder de locaties met de startadressen. Het hoogste stackadres is dan \$DFEF.

Het eerste deel van elke ISR, te weten de sprong naar het vervolg van de ISR in RAM, zetten we boven in de ROM, onder de interruptvectoren. Een extended indirect jump-instructie kost 4 bytes (zie appendix B). Voor de 8 interruptvectoren (inclusief de gereserveerde voor de toekomst) hebben we 32 geheugenlocaties nodig, dus van \$FFD0 t/m \$FFEF. Op deze ROM-locaties moeten nu de volgende instructies staan:

adres	instructie	commentaar
\$FFD0	JMP [\$DFF0]	naar beginadres reserved ISR in RAM
\$FFD4	JMP [\$DFF2]	naar beginadres SWI3-ISR in RAM
\$FFD8	JMP [\$DFF4]	naar beginadres SWI2-ISR in RAM
\$FFDC	JMP [\$DFF6]	naar beginadres FIRQ-ISR in RAM
\$FFE0	JMP [\$DFF8]	naar beginadres IRQ-ISR in RAM
\$FFE4	JMP [\$DFFA]	naar beginadres SWI-ISR in RAM
\$FFE8	JMP [\$DFFC]	naar beginadres NMI-ISR in RAM

Het startadres van de restart-interrupt (RESET) moet niet worden verplaatst, omdat dit het beginadres is van het operating system (in ROM).

Opdat de juiste spronginstructie wordt uitgevoerd bij een interrupt, moet de volgende informatie in ROM staan:

adres	inhoud	commentaar
\$FFF0/\$FFF1	\$FFD0	beginadres reserved ISR
\$FFF2/\$FFF3	\$FFD4	beginadres SWI3-ISR
\$FFF4/\$FFF5	\$FFD8	beginadres SWI2-ISR
\$FFF6/\$FFF7	\$FFDC	beginadres FIRQ-ISR
\$FFF8/\$FFF9	\$FFE0	beginadres IRQ-ISR
\$FFFA/\$FFFB	\$FFE4	beginadres SWI-ISR
\$FFFC/\$FFFD	\$FFE8	beginadres NMI-ISR
\$FFFE/\$FFFF	\$E000	beginadres RESET-ISR

Het is sterk aan te raden voor elke hardware interrupt een ISR te schrijven, ook al wordt niet elke interrupt gebruikt. Bij een storing op een interrupt-ingang wordt er naar de bijbehorende interrupt service routine gesprongen. Is de betreffende interruptvector niet geïnitieerd, dan wordt er naar een willekeurig adres gesprongen, alwaar het programma (de ISR) wordt voortgezet. Het gebruikersprogramma loopt nu

verkeerd af, omdat op die willekeurige locatie geen ISR staat. Om dit probleem te voorkomen, moet u alle interruptvectoren initialiseren. Voor de niet-gebruikte interrupts kan worden verwezen naar hetzelfde startadres, alwaar zich slechts de instructie RTI hoeft te bevinden. Nu wordt er teruggekeerd naar het onderbroken programma, als er een storing optreedt op een niet-gebruikte interruptingang.

Het operating system zal meestal zelf ook gebruik maken van interrupts. Hiertoe moet het operating system de interruptvectoren initialiseren. De bijbehorende "system-ISR's" maken deel uit van het operating system en staan in ROM. Het begin van het operating system kan als volgt uit zien:

10	NMIVEC	EQU	\$DFFC	
20	SWIVEC	EQU	\$DFFA	
30	IRQVEC	EQU	\$DFF8	
40	FIVEC	EQU	\$DFF6	
50	SW2VEC	EQU	\$DFF4	
60	SW3VEC	EQU	\$DFF2	
70	*			
80	*			
90		ORG	\$E000	beginadres van operating system
100		LDX	#NMISYS	zet startadres van system-
110		STX	NMIVEC	NMI-ISR in interruptvector
120		LDS	#\$DFF0	initialiseer stackpointer en
130	*			enable NMI-interrupt
140		LDX	#SWISYS	zet startadres van system-
150		STX	SWIVEC	SWI-ISR in interruptvector
160		LDX	#IRQSYS	zet startadres van system-
170		STX	IRQVEC	IRQ-ISR in interruptvector
180		LDX	#FISYS	zet startadres van system-
190		STX	FIVEC	FIRQ-ISR in interruptvector
200		LDX	#SW2SYS	zet startadres van system-
210		STX	SW2VEC	SWI2-ISR in interruptvector
220		LDX	#SW3SYS	zet startadres van system-
230		STX	SW3VEC	SWI3-ISR in interruptvector
240		ANDCC	#0	enable IRQ- en FIRQ-interrupts
250		--		
260		--		
270	NMISYS	--		
280		--		
290		RTI		
300	SWISYS	--		
310		--		
320		RTI		

enzovoorts

De stackpointer wordt geladen met het adres \$DFF0 (regel 120). Dit is de locatie net boven de stack, hetgeen te hoog lijkt. Bij een push-instructie wordt eerst de stack pointer verlaagd en daarna wordt op de stack geschreven, zodat de eerste byte

boven in de stack komt te staan (zie paragraaf 7.3). Door het laden van de stack pointer wordt de NMI-interrupt ge-enabled (zie paragraaf 9.7.4). Alvorens een interrupt kan worden afgehandeld, moet de interruptvector zijn geïntialiseerd. Derhalve mag de stack pointer pas worden geladen, nadat het beginadres van de NMI-ISR is opgeborgen in de NMI-interruptvector (regel 100 en 110).

Zo ook mogen de interrupts IRQ en FIRQ pas worden ge-enabled (regel 240), nadat de bijbehorende interruptvectoren zijn geïntialiseerd.

9.7.10 De instructie CWAI

De CWAI-instructie (Clear CC-bits and WAit for Interrupt) is een immediate addressing instructie en heeft de volgende vorm:

CWAI #M

Het masker M bepaalt welke bits van het conditie code register worden gereset. Het masker is bedoeld om FIRQ- en/of IRQ-interrupts door te laten.

Het gevolg van de instructie CWAI kan wordt weergegeven in fig. 9.35.

(CC) := (CC) AND M	
set E	
zet inhoud van PC, U, Y, X, DP, B, A en CC op de stack	
/	
er is een niet-gemaskeerde interrupt	
voer bijbehorende ISR uit	
vervolg het programma achter de CWAI-instructie	

Fig. 9.35 Structogram van de CWAI-instructie

De instructie CWAI kan worden gebruikt om op een (niet-gemaskeerde) interrupt te wachten. Alle registerinhouden worden alvast op de stack gezet, ook indien er op een FIRQ wordt gewacht. Er kan (eventueel ongewild) een NMI optreden. Vandaar dat alle registers op de stack worden gered. Het afhandelen van de interrupt gaat nu snel. De registerinhouden staan immers al op de stack. Er kan direct worden begonnen met het ophalen van de interruptvector, zodra er een (niet-gemaskeerde) interrupt wordt gedetecteerd.

9.7.11 De instructie SYNC

De instructie SYNC (SYNChronize) is een inherent addressing instructie. Fig. 9.36 laat het effect van deze instructie zien.

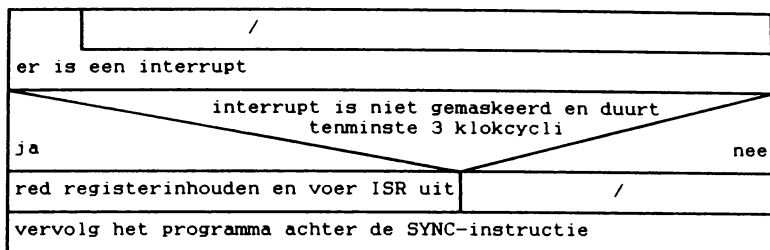


Fig. 9.36 Structogram van de SYNC-instructie

De SYNC-instructie verzorgt de synchronisatie tussen de software en een hardware proces. Het programma stopt bij een SYNC-instructie en wacht totdat er een interrupt komt. Zodra er een interrupt wordt gedetecteerd, wordt het programma voortgezet. Dit betekent dat het programma wordt doorgestart met behulp van een hardware interrupt. Het programma wordt dus als het ware gesynchroniseerd met een hardware signaal.

De SYNC-instructie kan ook worden toegepast voor snelle communicatie. Door immers de interrupt te maskeren (of IRQ, respectievelijk FIRQ korter te laten duren dan 3 klokcycli, maar langer dan 1 klokcyclus), wordt de interrupt service routine niet uitgevoerd, maar wordt het programma onmiddellijk voortgezet. Door de interrupt service routine niet te laten uitvoeren, wordt de tijdrovende overhead (registerinhouden redden, interruptvector ophalen enzovoorts) overgeslagen. Op deze manier is het mogelijk data binnen te halen of weg te sturen, zodra het randapparaat gereed is en dit meldt via een interruptsignaal. Dit is met name van belang bij snelle randapparaten, zoals disk units.

Zonder de SYNC-instructie zou de data met behulp van een fast interrupt FIRQ moeten worden uitgewisseld, hetgeen een overhead met zich meebrengt van minimaal

- 1 klokcyclus voor het resetten van de E-bit
- + 3 cycli voor het op de stack zetten van (PCL), (PCH) en (CC)
- + 1 cyclus voor het zetten van F en I
- + 2 cycli voor ophalen van de FIRQ-interruptvector
- + 1 cyclus voor gereed zetten van interruptvector op adresbus
- + 6 cycli voor het uitvoeren van de RTI-instructie aan het einde van de FIRQ-ISR

Dit zijn totaal 14 klokcycli aan overhead. Bij de maximale MC6809-klokkrequentie van 1 MHz kost deze overhead dus minstens

14 μ s. Hierbij komt nog de tijd om de huidige instructie af te maken in het geval dat de interrupt optreedt tijdens het uitvoeren van een instructie. De FIRQ-ISR bevat verder alleen de noodzakelijke load- en store-instructies ten behoeve van de gewenste communicatie. Dit is derhalve geen overhead.

Door toepassen van de instructie CWAI hoeft alleen de interruptvector te worden opgehaald bij het optreden van de interrupt. Dit betekent dat er dan in eerste instantie 5 klokcycli worden gespaard. Echter de instructie RTI duurt nu 9 cycli langer, omdat alle registerinhouden van de stack moeten worden gehaald.

Beter is het te wachten op de FIRQ-interrupt in de wachtlus:

```
LUS      BRA      LUS
```

Natuurlijk moet in de wachtlus een test opgenomen zijn om de lus te verlaten als de communicatie wordt beëindigd. Hierdoor wordt er trager gereageerd op de interrupt.

In plaats van te communiceren op basis van interrupt kan het meldingssignaal van het randapparaat worden opgevangen in een register, bij voorkeur in bit 7, omdat deze bit eenvoudig en snel getest kan worden. Bit 7 is de tekenbit. Deze wordt gekopieerd in bit N van het conditie code register en kan rechtstreeks worden getest met de instructie BPL (branch if plus) of BMI (branch if minus):

TEST	TST	GEREED	wacht tot randapparaat gereed
	BPL	TEST	is voor communicatie
	LDA	DATA	lees data van randapparaat
	BRA	TEST	wacht op nieuwe data

De overhead bestaat nu uit de tijd die het uitvoeren van de instructies TST en BPL kost. Deze bedraagt minimaal 6 klokcycli (bij direct addressing) + 2 klokcycli = 8 klokcycli, ofwel 8 μ s.

Komt de melding van het randapparaat net nadat de test-instructie is uitgevoerd, dan moet er minstens 2 μ s langer worden gewacht.

Het bovenstaande programma is niet geheel compleet. Er moet in de lus worden getest of alle informatie is uitgewisseld. Indien dit het geval is, moet niet meer worden teruggesprongen. Deze test doet de reactietijd op het meldingssignaal toenemen.

Voorbeeld 9.9 laat zien hoe de communicatie kan plaatsvinden met behulp van de SYNC-instructie.

Voorbeeld 9.8 (snelle communicatie met behulp van SYNC)

We willen 128 bytes van disk lezen. Dit kan zeer snel gebeuren door het signaal van de disk, dat aangeeft dat er een byte

gereed staat, te verbinden met bijvoorbeeld de ingang IRQ van de microprocessor. Het volgende programma zorgt dan voor de synchronisatie en de snelle communicatie.

10	ORCC	##00010000	maskeer de IRQ-interrupt
20	LDX	#BUFFER	initialiseer datapointer
30	LDB	#128	initialiseer byteteller
40	INPUT	SYNC	wacht op data
50	LDA	DISC	lees data en berg data op in
60	STA	,X+	geheugen; verhoog datapointer
70	DECB		
80	BNE	INPUT	spring als geen 128 bytes gelezen

Bij regel 10 wordt ervoor gezorgd dat er geen ISR wordt uitgevoerd, zodat er geen overhead is. Bij regel 40 wordt er gewacht op een interrupt. Zodra deze aankomt, wordt er verder gegaan met regel 50, waar de data wordt binnengehaald. Er gaat hierbij geen tijd verloren met het redden en herstellen van registerinhouden.

Realiseer u wel dat u er zeker van moet zijn, dat er geen andere interrupts kunnen optreden. Deze verstoren immers de synchronisatie, want elke interrupt heft de wachttoestand van de microprocessor op! Zonodig moeten alle andere interrupts buiten de processor worden geblokkeerd. Dit kan gebeuren met externe hardware of door de interruptaanvragers te melden, dat ze tot nader order geen interrupts mogen genereren.

9.8 Parameteroverdracht bij ISR

Net als bij gewone subroutines kan er overdracht van parameters nodig zijn. Bij interrupt-gestuurde communicatie moeten de uitvoergegevens aan de ISR worden doorgegeven, dan wel de invoergegevens vanuit de ISR worden doorgegeven aan het verwerkingsprogramma.

Dit kan, evenals bij subroutines (zie paragraaf 8.5), geschieden via

1. registers
2. vaste geheugenplaatsen
3. een pointer

De pointer kan zijn:

- a. een 16-bits register
- b. het direct page register
- c. twee vaste geheugenplaatsen
- d. twee stacklocaties

Van programma naar ISR

Parameteroverdracht van een programma naar een interrupt service routine kan zonder meer geschieden op elk van de bovengenoemde wijzen.

Van ISR naar programma

Parameteroverdracht van een interrupt service routine naar het onderbroken programma gaat niet zonder meer via registers. Dit komt doordat de registerinhouden bij de instructie RTI worden overschreven met hun oude waarden, die op de stack zijn bewaard. Willen we de parameters via registers overdragen, dan moeten de betreffende stacklocaties met de waarden van de parameters worden geladen. Aangekomen in het onderbroken programma zijn de betreffende registers met de parameterwaarden gevuld.

In de FIRQ-ISR moeten de registers zelf worden geladen met de parameters, omdat bij de FIRQ-interrupt alleen (CC) en (PC) worden gered op de stack, tenzij de FIRQ-ISR is voorafgegaan door een CWAI-instructie.

Parameteroverdracht via registers is doorgaans niet erg zinvol bij ISR's, omdat niet bekend is wanneer de interrupt optreedt. Het is niet duidelijk wanneer de parameters door de ISR overgenomen zijn, respectievelijk wanneer de parameters door de ISR in de registers zijn opgeborgen.

Worden de parameters op geheugenlocaties gezet, dan kunnen deze locaties in het gehele programma worden vrijgehouden ten behoeve van de overdracht. Er is immers een overvloed aan geheugenlocaties.

Met behulp van een flag (in het geheugen) kan worden aangegeven of de parameters zijn verwerkt, respectievelijk geldig zijn.

Bij de interrupt service routine van voorbeeld 9.7 hebben we parameteroverdracht via vaste geheugenplaatsen gezien. Door bijvoorbeeld op de geheugenlocaties \$A010 en \$A011 het adres \$A000 op te bergen, kunnen de gewenste parameters, zowel in het programma als in de ISR, worden bereikt middels het initialiseren van een pointer, bijvoorbeeld met

LDX #\$A010

en vervolgens de vijf parameters aan te wijzen met behulp van index addressing. Hierbij is de operand respectievelijk

X	voor UUR
1,X	voor MIN
2,X	voor SEC
3,X	voor TIENDE
4,X	voor TEL20m

Het overdragen van parameters met behulp van de stack is in het algemeen riskant, omdat in principe de mogelijkheid bestaat, dat de betreffende ISR binnen een andere ISR voorkomt. In dit geval wijst de stack pointer naar een andere stacklocatie, dan wanneer de ISR niet genest was.

Een pointer is eveneens te gebruiken voor het overdragen van parameters, in het bijzonder als de pointer op vaste

geheugenlocaties wordt opgeborgen.
Wordt de pointer via een register of via de stack overgedragen,
dan gelden de hiervoor genoemde opmerkingen betreffende
parameteroverdracht via registers of stack.

9.9 Samenvatting

Bij een interrupt (niet de RESET-interrupt) voert de
microprocessor automatisch de achtereenvolgende activiteiten
uit:

- huidige instructie afmaken
- Entire bit zetten dan wel resetten
- registerinhouden op de stack opbergen
- eventueel F-bit en/of I-bit zetten
- interrupt vector ophalen

Alvorens een interrupt (behalve de RESET-interrupt) zinvol
uitgevoerd kan worden, moet de programmeur ervoor zorgen dat

- de interrupt vector de juiste waarde heeft
- de stack pointer is geïnitieerd
- de betreffende mask bits zijn gereset

Binnen de ISR moet achtereenvolgens het volgende gebeuren:

- Bij meerdere interruptaanvragers per ingang moet de
aanvrager worden opgespoord.
- Er moet worden gezorgd dat de aanvrager zijn interrupt
weghaalt.
- De gewenste actie moet worden uitgevoerd.
- Er moet (middels de RTI-instructie) worden teruggekeerd
naar het hoofdprogramma.

Bij meerdere interruptaanvragers per interruptingang kan de
aanvrager worden achterhaald door

- software polling
- hardware vectoring door
 - * vectoradresvertaling
 - * het genereren van het vectoradres
 - * het verstrekken van vectorinformatie door de
aanvrager
 - * hardware polling
- een combinatie van software polling en hardware vectoring
- gecodeerde interruptingangen op de microprocessor

De MC6809 kent de volgende interrupts, in volgorde van
afnemende prioriteit:

- RESET
- NMI
- SWI
- FIRQ
- IRQ
- SWI2, SWI3

Overzicht van de MC6809-interrupts:

	RESET	NMI	SWI	FIRQ	IRQ	SWI2	SWI3
mask bit	-	-	I	F	-	-	-
set	F,I	E,F,I	E,F,I	F,I	E	E	E
reset	-	-	-	E	-	-	-
saved reg.	geen	alle	alle	CC,PC	alle	alle	alle
(DP) :=	0	-	-	-	-	-	-
vectoradres	\$FFFE	\$FFFC	\$FFFA	\$FFF6	\$FFF8	\$FFF4	\$FFF2

9.10 Opgaven

- Geef het structogram van een polling routine voor drie interrupts A, B en C. Interrupt C moet de hoogste prioriteit hebben en interrupt B de laagste.
- Ontwerp een RESET-ISR in assembly, zodanig dat
 - de interrupt vectors van NMI, SWI en IRQ worden geïnitieerd
 - De startadressen van de bijbehorende routines zijn respectievelijk \$4000, \$4A00 en \$4DF0.
 - de IRQ-interrupt ge-enabled wordt en de FIRQ-interrupt geblokkeerd blijft
 - de stack op adres \$A7A7 begint
 - naar een procesbesturingsprogramma wordt gesprongen, dat begint op adres \$2000
 Neem aan dat zich RAM bevindt op de vectoradressen van NMI, SWI en IRQ.

10 INPUT/OUTPUT

10.1 Specifieke doelstelling

Na het bestuderen van hoofdstuk 10 wordt u geacht in staat te zijn tot het volgende:

- * Het onderscheid kennen tussen memory-mapped I/O en I/O-mapped I/O.
- * Programmed I/O kunnen toepassen.
- * De voordelen en nadelen kennen van parallel datatransport en serieel datatransport.
- * Weten op welke wijzen de synchronisatie tussen processor en Input/Output-apparaten kan worden gerealiseerd.
- * Het principe van een handshake kennen.
- * De toepassing van een startbit en een stopbit kennen bij asynchrone communicatie.
- * I/O op basis van interrupt kunnen toepassen.
- * De functies van PIA (MC6821), ACIA (MC6850) en PTM (MC6840) kennen.
- * PIA, ACIA en PTM op de juiste wijze kunnen initialiseren.
- * Interrupts van PIA, ACIA en PTM kunnen afhandelen.
- * De procedure kennen van direct memory access.
- * De functies van de DMAC (MC6844) kennen.
- * DMA met behulp van de DMAC kunnen toepassen.

10.2 Algemeen

In paragraaf 6.3.4 hebt u reeds gezien dat de microprocessor via zijn registers kan communiceren met input/output-apparaten (periferie). Dit kan geschieden middels memory-mapped I/O of middels I/O-mapped I/O.

Memory mapped I/O

Bij memory-mapped I/O geschiedt de gegevensuitwisseling tussen microprocessor en I/O met behulp van dezelfde instructies als bij de communicatie tussen microprocessor en geheugen. Dit betekent dat de processor geen onderscheid maakt tussen geheugenadressen en I/O-adressen. Het lijkt alsof de I/O-apparaten deel uitmaken van het geheugen. Elk adres mag slechts corresponderen met één device, omdat slechts één device tegelijk informatie op de databus mag zetten. Zouden meerdere devices tegelijk informatie op de databus zetten, dan kan het voorkomen dat het ene device een 0 op een bepaalde datalijn zet, terwijl een ander device een 1 op diezelfde datalijn zet. Dit levert een ongedefinieerde waarde op. Bovendien gaan er grote stromen lopen, omdat het ene device de spanning op de lijn laag (0) probeert te maken, terwijl het andere device de

lijn van een hoge spanning (1) tracht te voorzien. Vandaar dat bij elk adres een geheugenlocatie behoort, of een I/O-apparaat, of geen van beiden. Het aansluiten van I/O-apparaten gaat hier ten koste van de adresruimte van geheugenschakelingen. Hoe meer I/O-apparaten aangesloten worden, des te minder geheugenlocaties er kunnen worden aangesloten. Doorgaans is dit geen probleem, omdat elk I/O-apparaat slechts enkele geheugenplaatsen opeist. Dit betekent dat er ten gevolge van I/O-aansluitingen meestal niet meer dan enkele tientallen tot enkele honderdtallen geheugenlocaties verloren gaan op het totaal van 64k. Bovendien is het geheugen eventueel uitbreidbaar. Hiertoe moeten speciale hardwarevoorzieningen worden aangebracht.

I/O-mapped I/O

Bij I/O-mapped I/O zijn er aparte instructies voor het communiceren met I/O-apparaten. Bij het ene type instructie wordt het adres opgevat als een geheugenadres, bij het andere type als een I/O-adres. Zodoende bestaat er onderscheid tussen geheugenadressen en I/O-adressen. Om dit onderscheid kenbaar te maken aan de geheugenschakelingen en aan de I/O-schakelingen, is er een aparte controllijn nodig. Deze geeft aan of het adres bedoeld is voor het geheugen of voor een I/O-device.

Naast dit onderscheid in memory-mapped I/O en I/O-mapped I/O kan de data-uitwisseling tussen processor en I/O geschieden op basis van programmed I/O en op basis van interrupt-I/O.

Programmed I/O

Bij programmed I/O wordt op bepaalde, door de programmeur vastgestelde plaatsen in het programma data binnengehaald, dan wel weggestuurd. Hier vindt de communicatie synchroon met het programma plaats.

Interrupt-I/O

Vindt de communicatie plaats nadat het I/O-apparaat een interrupt heeft gegenereerd, dan spreekt men van interrupt-I/O. Daar een interrupt in principe op een willekeurig moment kan optreden, is deze vorm van gegevensuitwisseling asynchroon ten opzichte van het programma. Een input device kan een interrupt genereren, als het gereed is voor het afgeven van data aan de microprocessor. Een output device genereert een interrupt, indien het gereed is om data van de microprocessor te ontvangen. Ook kunnen deze devices een interrupt afgeven als er een fout optreedt tijdens het datatransport. Het lopende programma wordt bij de interrupt onderbroken, de interrupt wordt door de processor afgehandeld en het onderbroken programma wordt voortgezet. De afhandeling van de interrupt betekent in dit geval dat er data-uitwisseling tussen processor en I/O device plaats vindt. Dit gebeurt in de interrupt service routine.

Direct memory access

Een heel andere vorm van communicatie is direct memory access. Hierbij vindt rechtstreekse uitwisseling van gegevens plaats tussen geheugen en I/O. Dit gebeurt zonder tussenkomst van de processor, hetgeen inhoudt dat er geen instructies nodig zijn ten behoeve van het datatransport. Er is extra hardware nodig om ervoor te zorgen dat het datatransport in goede banen wordt geleid. Deze hardware moet onder andere geheugenadressen genereren, opdat het I/O-device met de juiste geheugenlocaties communiceert.

Serieel en parallel transport

Wanneer er data over slechts één lijn wordt getransporteerd, spreekt men van serieel transport. De informatie moet immers bit voor bit, dus achter elkaar worden vervoerd. Worden er meerdere bits tegelijkertijd (over meerdere lijnen) vervoerd, dan is er sprake van parallel transport. Serieel transport is veel langzamer dan parallel transport. Bij parallel transport zijn er echter meer verbindingssdraden nodig, zodat dit duurder is, vooral bij grote afstanden.

Synchronisatie

Een probleem dat bij elke vorm van data-uitwisseling plaats vindt, is de synchronisatie. Als de microprocessor data binnenhaalt (bijvoorbeeld met behulp van een load-instructie), moet deze data op de databus aanwezig zijn, met andere woorden het randapparaat moet weten, op welk moment het zijn data op de databus moet zetten. Zo ook moet het randapparaat weten wanneer de microprocessor gegevens op de databus zet (bijvoorbeeld bij een store-instructie).

Wordt dit probleem opgelost door het kloksignaal van de processor mee te zenden, dan spreekt men van synchrone transmissie. Hier dient de data op de datalijn(en) geldig te zijn in een bepaald gedeelte van de klokpulsperiode, bijvoorbeeld bij de neergaande flank van de klokpuls. Zodoende weet elk aangesloten apparaat wanneer de data op de datalijn(en) gezet moet zijn, respectievelijk wanneer de data gelezen kan worden. Het datatransport verloopt hier synchroon (gelijktijdig) met een kloksignaal.

Bij asynchrone transmissie verloopt het datatransport asynchroon ten opzichte van het kloksignaal. Het gelijklopen van het aanbieden van informatie door de zender met het opnemen van informatie door de ontvanger, dient nu op een andere wijze te worden gerealiseerd.

Op al deze aspecten van gegevensuitwisseling wordt in dit hoofdstuk nader ingegaan.

10.3 Programmed I/O

Programmed I/O is de meest eenvoudige vorm van communicatie. De microprocessor neemt hierbij het initiatief voor het verzenden of binnenhalen van gegevens.

Voorbeeld 10.1 (Data verzenden)

Veronderstel er moet een signaal verzonden worden om de gasklep van een CV-ketel te besturen. Een logische 1 zorgt voor het open zijn, een 0 voor het gesloten zijn van de klep. Het stuursignaal kan bijvoorbeeld worden verzonden via datalijn D0. Nu is aan de databus ook het geheugen aangesloten en eventueel meerdere input/output schakelingen. Derhalve moet er kenbaar gemaakt worden of het om het besturingssignaal van de gasklep gaat of niet. Hiertoe wordt er een adres meegezonden (via de adresbus). Stel dat adres \$F0F0 aangeeft dat het om het genoemde stuursignaal gaat. Dan mag de informatie van lijn D0 de klep alleen bereiken, indien op de adresbus \$F0F0 staat. Hiertoe is er een adresdecoder nodig, die nagaat of dit adres aanwezig is.

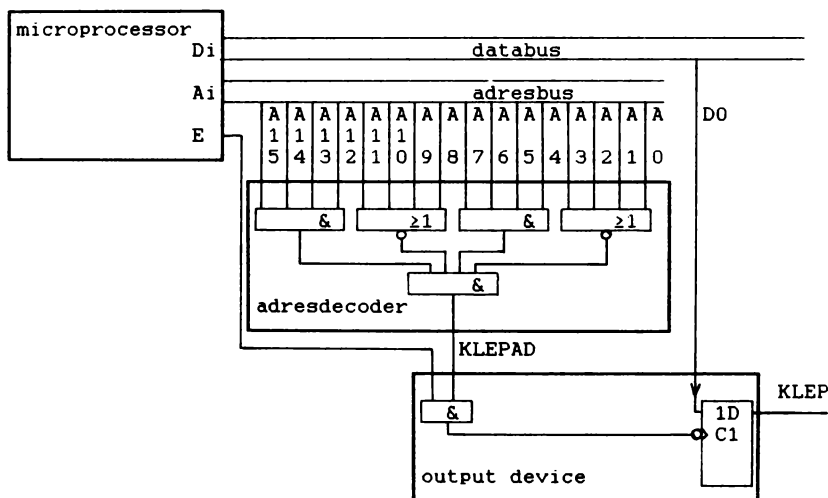


Fig. 10.1 Programmed output

Het openen van de gasklep kan geschieden met behulp van het volgende stukje programma:

```
KLEP    EQU    $F0F0
LDA     #1
STA     KLEP
```

Het sluiten van de gasklep kan worden bereikt met de volgende instructies:

Tussen deze groepjes instructies zal de microprocessor andere instructies uitvoeren, waardoor de informatie op de databus (evenals op de adresbus) voortdurend verandert. Dit betekent dat het stuursignaal voor de gasklep moet worden vastgehouden met behulp van een flipflop (zie fig. 10.1).

Het signaal KLEPAD is alleen 1 als de waarde \$FOF0 op de adresbus staat. Zodoende wordt het kloksignaal E alleen doorgelaten als het output device wordt geadresseerd. Alleen dan kan de informatie van D0 in de flipflop worden ingelezen. Dit gebeurt op de neergaande flank van de klokpuls E. Dit signaal wordt door de microprocessor afgegeven. Er is hier sprake van synchroon datatransport.

Op soortgelijke wijze kan de lijn D0 van fig. 10.1 worden gebruikt voor het verzenden van gegevens naar bijvoorbeeld een printer. Aangezien hier ASCII-karakters verzonden moeten worden, is er sprake van serieel datatransport. De diverse bits van een karakter moeten immers na elkaar worden overgestuurd.

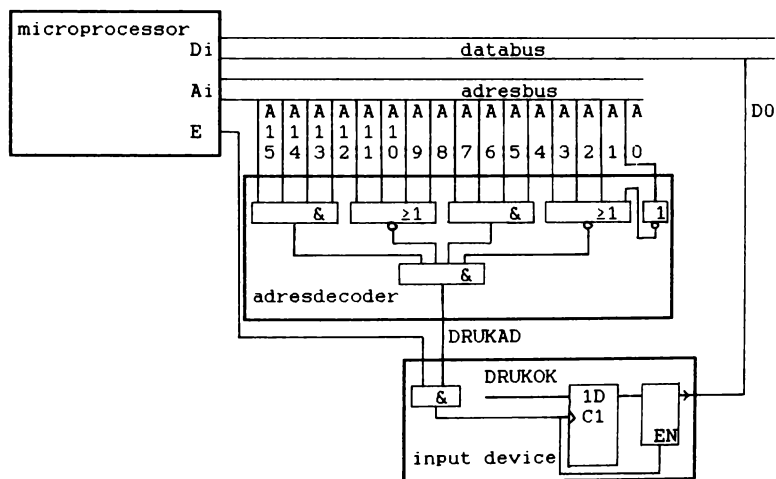


Fig. 10.2 Programmed input

Voorbeeld 10.2 (Data binnenhalen)

Stel dat we het signaal DRUKOK willen binnenhalen. Dit signaal geeft aan dat de waterdruk in de CV-installatie binnen de gewenste waarden ligt. Als DRUKOK gelijk is aan 0, is de druk niet goed en moeten er maatregelen worden getroffen (zoals de pomp stilzetten en de gasklep sluiten).

De CV-installatie mag dit signaal alleen op de databus zetten, als alle andere schakelingen geen informatie verzenden. Om dit te bereiken is er een adresdecoder nodig en een uniek adres, bijvoorbeeld \$F0F1. De zender wordt als het ware van de bus losgekoppeld als de enable ingang EN gelijk aan 0 is. Het ingangssignaal wordt doorgegeven aan de uitgang van de zender als EN=1 is. Hiervoor moet een zogenaamde open-collector zender of een tri-state zender worden gebruikt. Bij deze zenders is de uitgang hoogimpedant en dus als het ware losgekoppeld van de datalijn, als EN=0 is.

Wordt DRUKOK vervoerd via datalijn D0, dan krijgen we het schema van fig. 10.2. Ook hier zorgt het kloksignaal E voor de synchronisatie. Bij de opgaande flank van de klok E wordt de flipflop ingelezen en bij de neergaande flank van de klok leest de microprocessor de gegevens van de databus. Overigens moeten er in de CV-installatie maatregelen worden getroffen om te voorkomen dat de ingang van de flipflop verandert op het moment dat de flipflop wordt ingelezen. Anders zou de toestand van de flipflop ongedefinieerd kunnen worden.

Op soortgelijke wijze kan bit voor bit informatie worden binnengehaald, bijvoorbeeld vanaf een toetsenbord. Er is dan sprake van serieel datatransport. Worden de bits van een karakter tegelijk via de 8 datalijnen vervoerd, dan hebben we te maken met parallel transport.

Merk op dat een input device een apparaat is dat data naar de microprocessor zendt en dat een output device informatie van de processor ontvangt. De termen input en output hebben betrekking op de processor!

Voorbeeld 10.3 (Data zenden en ontvangen)

In fig. 10.2 kan niet zonder meer gebruik worden gemaakt van hetzelfde adres als in fig. 10.1, ondanks het feit dat de twee signalen op verschillende datalijnen zijn aangesloten. Dit komt doordat de microprocessor bij een store-instructie alle datalijnen bestuurt. Zou de CV-installatie op één adres aangesloten zijn, dan zou de zender van DRUKOK tegelijk worden geactiveerd bij het adresseren van KLEP. Hierbij bestuurt de microprocessor eveneens de lijn D0. Dit kan leiden tot een ongedefinieerd niveau op deze lijn en tot het optreden van grote stromen.

Door het signaal $R/\bar{W}$ te gebruiken kan wel worden volstaan met slechts één adres voor de beide CV-signalen. Bij het invoeren van informatie (lezen) maakt de microprocessor deze lijn 1 ($\text{Read} = \bar{\text{Write}} = 1$, zodat $\text{Write} = 0$). Bij het uitvoeren van informatie (schrijven) wordt de $R/\bar{W}$ -lijn 0 gemaakt ($\text{Read} = \bar{\text{Write}} = 0$, zodat $\text{Write} = 1$). Zorgen we er nu voor dat de zender van DRUKOK alleen geactiveerd wordt tijdens het lezen, dan ontstaat er geen probleem bij het aansturen van de klep. Dit wordt bereikt door het $R/\bar{W}$ -signaal in de adresdecoder van de klep op te nemen (zie fig. 10.3). Blokkeren we tijdens het

lezen de ontvanger van het stuursignaal van de klep, dan bestaat zelfs de mogelijkheid dit stuursignaal en het signaal DRUKOK te combineren, zoals fig. 10.3 laat zien. Lezen en schrijven kan immers nooit tegelijkertijd optreden. Als adres is hier \$F0F0 gekozen.

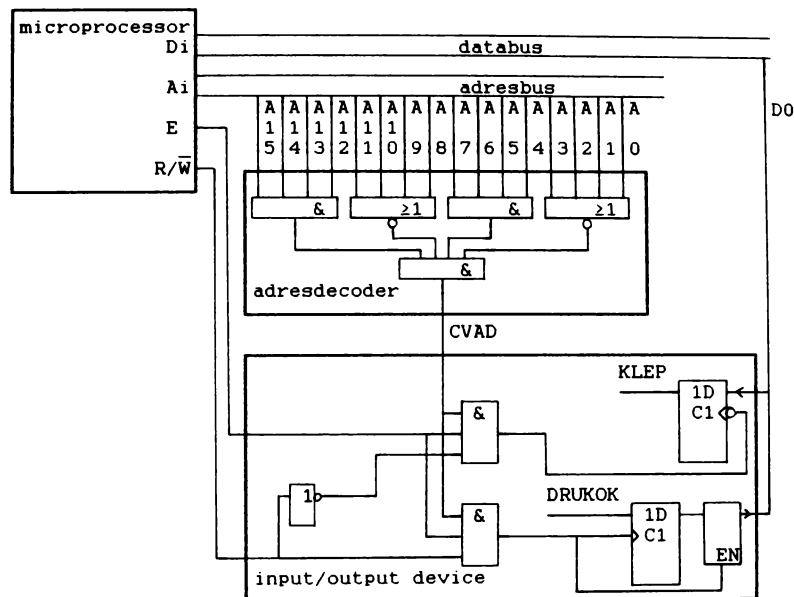


Fig. 10.3 Programmed input/output

In hoofdstuk 11 wordt dieper ingegaan op de timing van de signalen van de microprocessor.

In deze paragraaf hebt u gezien dat er voor de communicatie tussen de microprocessor en zijn omgeving extra digitale elektronica nodig is in de vorm van adresdecoders, flipflops en besturingsgates. Afhankelijk van de soort randapparaten (de periferie) zijn er meer of minder signalen nodig. Meestal is er zowel input als output. De verhouding tussen het aantal ingangssignalen en het aantal uitgangssignalen varieert nogal bij de diverse randapparaten. Er is echter een keur van IC's verkrijgbaar, die tamelijk universeel zijn voor wat betreft de signalen van en naar de periferie.

Men spreekt van U(S)ART's, Universal (Synchronous/)Asynchronous Receivers/Transmitters als het gaat om serietransport van data. Deze bouwstenen zijn universeel doordat ze programmeerbaar

zijn. Op deze wijze kan dezelfde communicatiechip voor diverse functies worden toegepast. Zoals de naam aangeeft kunnen via een USART zowel signalen worden binnengehaald als verzonden. Bovendien kan dit naar keuze op synchrone wijze dan wel op asynchrone wijze geschieden. Dit heeft betrekking op de communicatie met de periferie. De meeste microprocessoren werken synchroon. In dat geval verloopt het datatransport tussen de microprocessor en de bijbehorende USART's synchroon. De communicatie tussen USART en periferie kan zowel synchroon als asynchroon gebeuren. De meeste U(S)ART's worden softwarematig ingesteld. Dit geschiedt via een microprocessor. Aangezien de diverse families van microprocessoren verschillende signalen hebben, heeft elke microprocessorfamilie zijn eigen periferiechips. Deze zijn soms ook aansluitbaar op processoren van een andere familie. Hierbij kan extra externe hardware nodig zijn. Sommige communicatiechips zijn hardwarematig instelbaar. Deze kunnen nagedrag worden toegepast in randapparaten die niet zijn uitgerust met een microprocessor. Verder zijn er vele programmeerbare IC's voor parallelle communicatie.

10.4 Interrupt-I/O

Bij programmed I/O moet er worden gewacht tot het I/O-device gereed is om de data te verzenden of te ontvangen. Dit kan geschieden in een wachtlus of door te pollen. Dit is geen probleem als de reactietijd van het I/O-device klein is of als de processor niet verder kan met de uitvoering van het programma, voordat de communicatie is afgehandeld. In het andere geval is deze methode erg inefficiënt, omdat er veel CPU-tijd (tijd van de central processing unit) verloren gaat. Kan het programma worden voortgezet, dan dient regelmatig te worden nagegaan of de I/O is afgehandeld. De vraag is nu hoe vaak en waar dat moet gebeuren. Bovendien kan men het risico lopen om een terugmelding van een I/O-device te missen.

Het probleem van het moeten wachten op de terugmelding van een device, dan wel het regelmatig in het programma moeten pollen van de terugmeldingssignalen met het risico er een te missen, kan worden opgelost door gebruik te maken van interrupt-I/O. Hierbij wordt het terugmeldingssignaal van het device verbonden met een interruptingang van de microprocessor. Hierdoor wordt het lopende programma onderbroken, zodra het device een terugmelding geeft. Het initiatief ligt nu bij het device. De eigenlijke data-uitwisseling geschiedt nu in een interrupt service routine. Hierna zet de microprocessor het onderbroken programma weer voort. Er gaat bij interrupt-I/O geen CPU-tijd verloren met wachten op antwoord van een I/O-apparaat.

Voorbeeld 10.4

Een keyboard geeft een 0-puls STROBE af indien een toets wordt ingedrukt. De toetsinformatie is dan in ASCII beschikbaar op de parallelle uitgangspoort van het keyboard. Deze poort kan worden ge-enabled met het signaal EN en is via adres \$FF80 adresseerbaar. STROBE wordt aangesloten op de IRQ-ingang van de microprocessor (zie fig. 10.4). De ASCII-informatie wordt opgeslagen in de geheugenlocatie met als adres \$4000.

Het hoofdprogramma hoeft zich in eerst instantie niet te bemoeien met de communicatie. Deze vindt geheel plaats binnen de interrupt service routine.

Wel moet ervoor gezorgd zijn, dat de stack pointer is ingesteld. Dit kan in het hoofdprogramma geschieden met de instructie

```
LDS #STACK
```

waarin STACK het adres is, dat net boven de top van de stack wijst. Het is ook mogelijk de stack pointer te gebruiken zoals die is ingesteld door het operating system.

Het startadres van de interrupt service routine moet op de juiste geheugenlocatie staan:

```
LDX #KEYISR
```

```
STX IRQVEC
```

Hier is KEYISR het startadres van de interrupt service routine en IRQVEC is de locatie waar de microprocessor het startadres ophaalt.

Verder moet het hoofdprogramma ervoor gezorgd hebben, dat de IRQ-interrupt niet is geblokkeerd. De interrupt mask bit I van het condition code register moet gereset zijn, bijvoorbeeld met de instructie

```
ANDCC #%11101111
```

Uiteindelijk moet het programma de informatie van het keyboard verwerken.

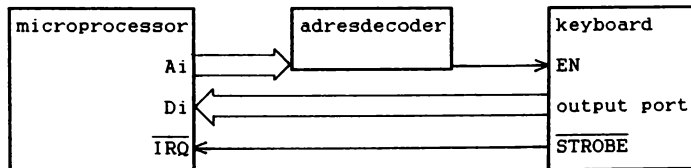


Fig. 10.4 Interrupt-I/O

De interrupt service routine ziet er als volgt uit:

```
100 *****
110 * AFHANDELING VAN KEYBOARD INTERRUPT *
120 *****
130 *
140 KEYOUT EQU $FF80
150 KEYDAT EQU $4000
160 *
```



```

170 KEYISR  LDA    KEYOUT
180          STA    KEYDAT
190          RTI

```

Komen er meerdere interrupts binnen op de $\overline{\text{IRQ}}$ -ingang van de microprocessor, dan moet via polling of vectoring worden nagegaan wie de veroorzaker van de interrupt is (zie paragraaf 9.5 en 9.6). Voor wat betreft polling moet de aanvrager een status bit hebben dat via de databus opgehaald kan worden en door de microprocessor getest.

10.5 Synchronisatie

10.5.1 Inleiding

Voor wat betreft de synchronisatie kan het datatransport op twee wijzen geschieden:

1. synchroon
2. asynchroon

Synchroon datatransport

Bij de synchrone communicatie hebben de zender en de ontvanger (aan weerszijden van de verbinding) een gemeenschappelijke klok (zie fig. 10.5). Dit houdt in dat het kloksignaal met de data moet worden meegezonden. Het maakt hierbij niet uit of de klokgenerator zich aan de zenzijde of aan de ontvangzijde bevindt.

Het aantal datalijnen is hierbij niet van belang. Zowel serie- als paralleltransport kan synchroon geschieden.

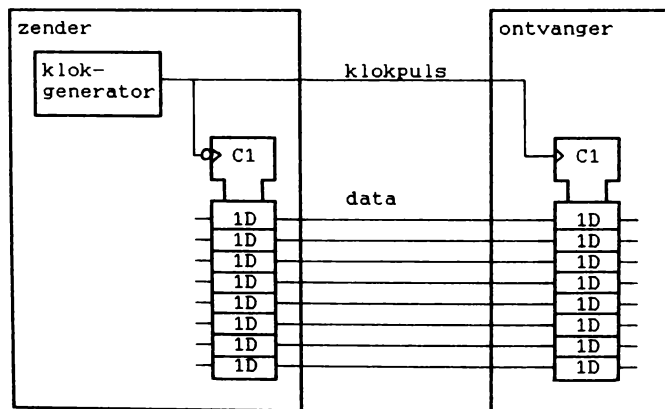


Fig. 10.5 Synchroon paralleltransport

De ontvanger mag de data op de verbindinglijn vanzelfsprekend pas gebruiken als deze geldig is. Bij de synchrone communicatie is dit eenvoudig te realiseren door de data bijvoorbeeld op de ene flank van de klokpuls te verzenden en op de andere flank van de klokpuls in te lezen. In fig. 10.5 wordt de data op de neergaande flank van de klok verzonden en op de opgaande flank ontvangen. Fig. 10.6 laat zien dat hierdoor de data wordt ingelezen op een moment dat deze inderdaad geldig is. Eventuele looptijdverschillen van de signalen over de verbinding hebben geen invloed, mits de pulsbreedte van de klok groot genoeg is. Op de neergaande flank veranderen de uitgangssignalen bij de zender. De data is dan ongeldig totdat alle signalen hun nieuwe waarde hebben aangenomen en bij de ontvanger zijn aangekomen.

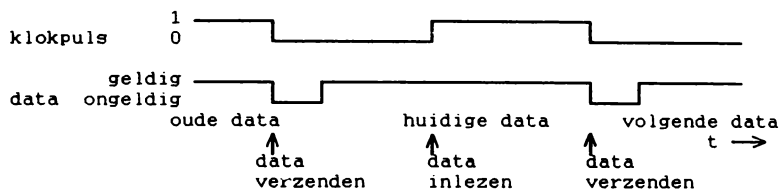


Fig. 10.6 Timing bij synchrone transmissie

De communicatie tussen de microprocessor en de PIA verloopt synchroon, immers de klokpuls E wordt met de data meegezonden. Het datatransport geschiedt via de bidirectionele databus. Het controlsignaal $R/\bar{W}$ geeft aan wat de richting van het transport is.

Asynchroon datatransport

De communicatie tussen de PIA en de randapparatuur verloopt meestal asynchroon. De randapparatuur heeft een eigen klokgenerator. Hierdoor lopen de klokken van PIA en randapparaat niet gelijk, dus asynchroon. Twee verschillende klokken hebben nooit exact dezelfde frequentie, zodat een transmissiesysteem dat met twee klokgeneratoren werkt, altijd asynchroon is (zie fig. 10.7).

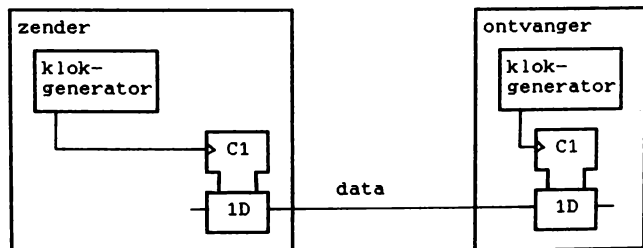


Fig. 10.7 Asynchroon serietransport

Ook asynchrone communicatie kan plaatsvinden met zowel serie- als paralleltransport.

Bij asynchrone communicatie bestaat het gevaar dat de data wordt ingelezen op het moment dat deze bezig is van waarde te veranderen. Dit betekent dat de ingelezen data ongeldig is. De toestand van de flipflops kan hierbij zelfs ongedefinieerd worden. Het opnemen van de data zal op het juiste moment moeten plaatsvinden. Er dient hier gesynchroniseerd te worden.

Overigens behoeven zender en ontvanger niet in alle gevallen te worden gesynchroniseerd. Het aansturen van signaleringslampjes en relais (bijvoorbeeld voor het bedienen van kleppen en motoren) hoeft niet synchroon te gebeuren. Zo kan ook de status van een aantal schakelaars, die een vaste instelling aangeven, asynchroon worden binnengehaald. Als de waarden van de ingangssignalen tijdens bedrijf kunnen veranderen, moet er in het algemeen synchronisatie plaatsvinden.

Het synchroniseren van het datatransport kan geschieden met behulp van:

1. strobing
2. synchronisatie-informatie binnen de data zelf
3. handshaking

Communicatie kan in principe in twee richtingen (bidirectioneel) plaatsvinden, eventueel via dezelfde lijnen. In het laatste geval is er een apart signaal nodig, dat aangeeft in welke richting de data getransporteerd moet worden. Dit wordt doorgaans aangegeven door het signaal read/write.

10.5.2 Strobing

Bij strobing wordt er een aparte lijn gebruikt ten behoeve van een signaal om de zender en de ontvanger te synchroniseren. De zender geeft een puls af om aan te geven dat de data geldig is (zie fig. 10.8).



Fig. 10.8 Het strobe-sigitaal

Deze puls, ook wel strobe genoemd, kan door de ontvanger worden gebruikt om de data in te klokken in een inputregister. Hiermee is de synchronisatie echter nog niet gereed. Het verwerken van

de ontvangen data geschiedt immers met behulp van een eigen klokpuls. De ontvangen data (in het inputregister) moet worden gesynchroniseerd met deze klokpuls. Indien de strobe-puls smal is, moet deze worden onthouden in een SR-flipflop (zie fig. 10.9).

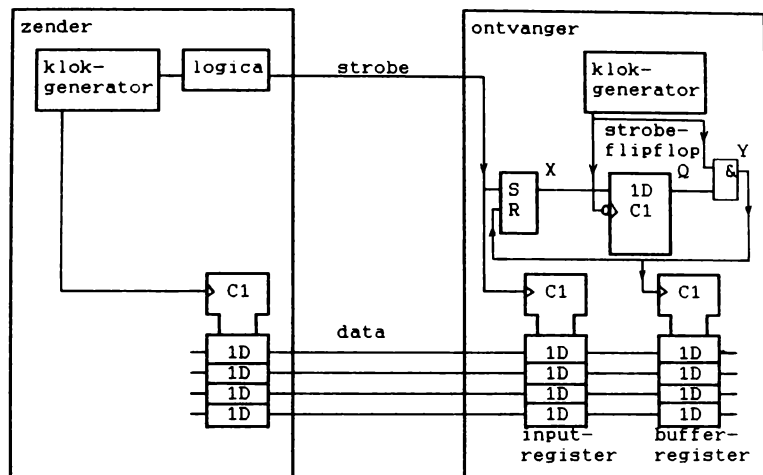


Fig. 10.9 Synchronisatie met behulp van een strobe-sigitaal

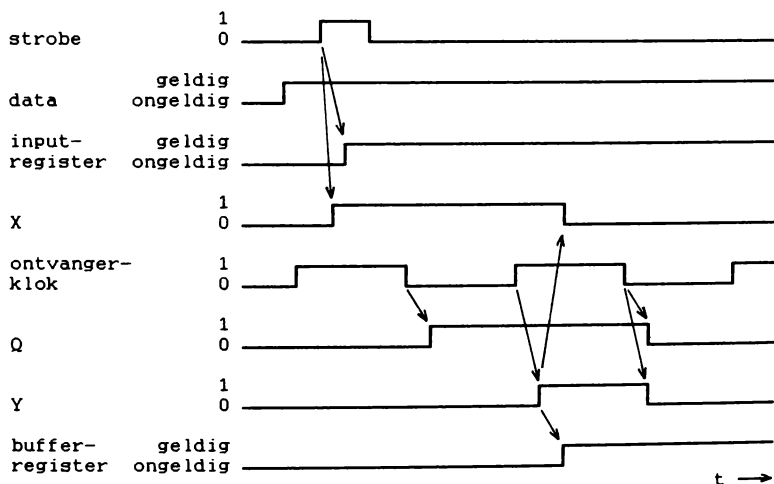


Fig. 10.10 Timing van het synchroniseren volgens fig. 10.9

De uitgang X van de SR-flipflop wordt ingelezen in de strobe-flipflop bij de neergaande flank van de ontvangerklok. De uitgang Q van deze strobe-flipflop wordt gebruikt om de (volgende) klokpuls door te laten. Met deze puls (op de uitgang Y van de AND-gate) wordt het bufferregister ingelezen en tegelijkertijd wordt de SR-flipflop gereset. De strobe-flipflop wordt op de neergaande flank van de klokpuls gereset (zie ook fig. 10.10). Het bufferregister wordt ingelezen met de ontvangerklok, hetgeen betekent dat de uitgangen van het bufferregister synchroon zijn met deze klok.

Is de ontvanger uitgerust met een microprocessor, dan kan de uitgang X van de SR-flipflop dienen als interruptsignaal. De signalen Q en Y en het bufferregister zijn dan niet nodig. Het inputregister wordt dan met de databus van de microprocessor van de ontvanger verbonden, zodat dit door de processor kan worden gelezen. Dit lezen gebeurt in de interrupt service routine, waarin tevens kan worden gezorgd voor het resetten van de SR-flipflop. Hiertoe moet de reset-ingang van deze flipflop aangesloten zijn op de databus. Bij meerdere interrupts wordt ook de uitgang X van de SR-flipflop op de databus aangesloten, zodat de microprocessor middels polling kan nagaan welke aanvrager de interrupt heeft gegenereerd.

In paragraaf 10.8 zult u zien dat strobing kan geschieden met behulp van de PIA.

10.5.3 Synchroniseren op de data

Worden er gewoon enen en nullen verzonden, dan is er geen synchronisatie-informatie aanwezig.

Indien de data synchronisatie-informatie bevat, kan de ontvanger hierop worden gesynchroniseerd. Er is dan geen afzonderlijk strobe-signaal nodig. Deze methode van synchronisatie wordt veelal toegepast bij seriële interfaces.

Indien er geen apart klok- of strobe-signaal wordt meegestuurd, is het vergelijken van de flanken van de ontvangerklok met de veranderingen in de ontvangen data de enige manier om te constateren of de ontvangerklok voor of achter loopt ten opzichte van de zenderklok. De signaalveranderingen zijn synchroon met de zenderklok. Treden er gedurende lange tijd geen veranderingen in het signaal op, dan gaat de synchronisatie verloren, immers de frequentie van de oscillator aan de ontvangerzijde is anders dan die van de oscillator van de zender. Dit betekent dat de ontvangerklok regelmatig moet worden bijgesteld. Vaak wordt gebruik gemaakt van hoogfrequent oscillatoren, zodat met een frequentiedeler de juiste zend-, respectievelijk ontvangfrequentie kan worden ingesteld. Hiertoe moet worden afgesproken hoe lang een bit duurt, zodat de frequentiedelers op de juiste waarde kunnen worden ingesteld.

Signaalcodering

Met de keuze van een juiste signaalcodering kan worden bereikt, dat het signaal voldoende vaak van niveau verandert. Hierbij wordt gewerkt met een vaste frequentie waarmee de informatie wordt verzonden.

1. Synchronisatie-informatie in de fase.

Een mogelijkheid is bijvoorbeeld te definiëren dat een 1 overeenkomt met een hoog niveau in de eerste helft van de bittijd en een laag niveau in de tweede helft van de bittijd. Bij een 0 is dit net andersom (zie fig. 10.11). De bittijd is de tijd gedurende welke een bit informatie aanwezig is.

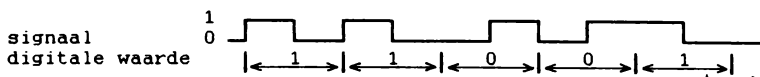


Fig. 10.11 Signaal, waarbij de positie van het hoge niveau in de bittijd bepalend is voor de binaire waarde van het signaal

Alvorens de eigenlijke communicatie kan beginnen, moet de ontvanger worden gesynchroniseerd. Hiertoe moeten er een aantal flanken worden gegenereerd door een aantal zogenaamde synchronisatiepatronen te zenden.

Bij de methode van fig. 10.11 ligt de opgaande flank nu eens in het begin van de bittijd en dan weer in het midden. Raakt de ontvangerklok ten gevolge van een storing een halve bittijd verschoven, dan worden de volgende bits allemaal geïnverteerd binnengehaald.

2. Synchronisatie-informatie in de aan/afwezigheid van flanken.

Een codering die dit nadeel niet heeft, is bijvoorbeeld de logische 0 te definiëren als een flank in het begin van de bittijd en een 1 als het ontbreken van een flank (zie fig. 10.12).

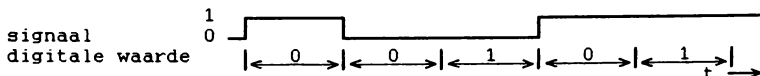


Fig. 10.12 Signaal, waarbij een 0 wordt weergegeven door een flank en een 1 door het ontbreken van een flank

Ook hier moet de ontvanger worden gesynchroniseerd, voordat de communicatie begint.

Bitsynchronisatie

Voordat de informatie-uitwisseling kan beginnen, moet de ontvanger weten op welk moment er een bit moet worden ingelezen. Hiertoe moet de ontvangerklok gelijk worden gezet met de zenderklok. Dit gebeurt door voldoende flanken te verzenden, zodat de ontvanger zijn klok kan gelijkzetten.

Wordt er enige tijd niet gezonden, dan ontbreken de flanken, zodat de ontvanger op een gegeven moment niet synchroon is met de zender. Wanneer er dan opnieuw gezonden wordt, moeten er eerst een aantal synchronisatieflanken worden gezonden, opdat de ontvanger opnieuw kan synchroniseren. Daarna kan de datatransmissie weer beginnen. Gedurende het gehele transport van een blok data blijft de ontvanger gesynchroniseerd.

Karaktersynchronisatie

Na het synchroniseren van de bits moet de ontvanger weten welke bit er wordt ingeklokt. Een groepje bits (meestal 7 of 8) vormt een karakter. Er moet dus bekend zijn wanneer een karakter begint. Dit geschiedt door een aantal zogenaamde synchronisatiekarakters te zenden, voordat de eigenlijke communicatie begint. Zodra de ontvanger een synchronisatiekarakter heeft gedetecteerd, weet hij dat hierna een volgend karakter begint. Men spreekt hier van karaktersynchronisatie.

Bij fig. 10.13 gebeurt dit door het zenden van synchronisatiekarakters, bijvoorbeeld %0111 1110. Om de synchronisatiekarakters te kunnen onderscheiden van data, mogen tijdens het datatransport niet 6 enen, gevolgd door een 0 voorkomen. Om dit te bereiken wordt door de zender telkens een 0 toegevoegd na een opeenvolging van 5 data-enen. De ontvanger laat elke 0 die wordt voorafgegaan door 5 enen, weg. Op deze manier wordt tevens bereikt, dat de ontvanger niet uit synchronisatie gaat door het ontbreken van flanken bij lange rijen enen. Er komen immers nooit meer dan 6 enen na elkaar voor.

Het nadeel van deze methode is dat er minder flanken voorkomen in het datatransport. Dit stelt iets hogere eisen aan de gelijkheid van de frequenties van de klokpulsen van zender en ontvanger.

Synchroniseren met start- en stopbits

Een andere methode van synchronisatie, die veel wordt toegepast bij seriële datatransmissie, is het gebruik van startbits en stopbits. Bij het zenden wordt elk karakter voorafgegaan door een zogenaamde startbit en afgesloten met een stopbit, die eventueel wordt voorafgegaan door een pariteitsbit (zie fig. 10.14). De start- en stopbits bieden de ontvanger de mogelijkheid om bij elk karakter opnieuw te synchroniseren. Hiertoe moet de ontvanger zijn uitgerust met een oscillator met een frequentie die bijvoorbeeld een factor 16 hoger is dan de frequentie van de databits.

Een factor 16 is handig in verband met een 16-deler, die bestaat uit een 4-bits teller. Deze doorloopt achtereenvolgens de standen 0, 1, 2, 3,, 14, 15, 0, enzovoorts. Er zijn 16 verschillende standen, zodat de oscillatorfrequentie door 16 wordt gedeeld en de ontvangerklok ongeveer dezelfde frequentie heeft als de binnenkomende data. Algemeen geldt dat een

frequentie op eenvoudige wijze kan worden gedeeld door een factor 2^n door een n-bits binaire teller toe te passen als frequentiedeler.

In rust heeft de transmissielijn de waarde 1. De startbit wordt gerealiseerd door de datalijn gedurende de bittijd 0 te maken. De ontvanger kan nu synchroniseren op de neergaande flank, die door de startbit wordt veroorzaakt. Bij die neergaande flank wordt een zestienstandenteller gestart, die wordt gestuurd door de oscillator van de ontvanger. De teller wordt gereset in stand 8. Dit is ongeveer in het midden van de startbit. De teller begint nu opnieuw te tellen, maar wordt pas gestopt bij de stopbit. Echter na stand 15 volgt telkens stand 0 bij een zestienstandenteller. Steeds als de tellerstand 0 wordt, wordt de data ingeklokt. Dit is ongeveer in het midden van de bittijd.

Bij het detecteren van de stopbit wordt de teller gestopt. Pas bij het detecteren van de volgende startbit (de volgende neergaande flank) wordt de teller opnieuw gestart (vanaf stand 0). Er vindt hier dus bij elk karakter synchronisatie plaats. De startbit zorgt zowel voor bit- als voor karaktersynchronisatie.

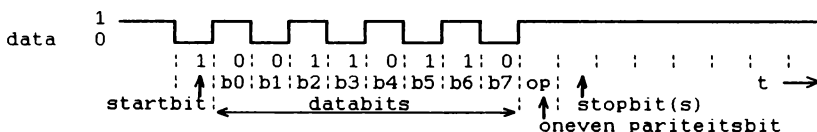


Fig. 10.14 Dataframe (met oneven pariteitsbit)

De minstwaardige bit van het karakter wordt het eerst gezonden. De stopbit is noodzakelijk voor de synchronisatie. Zou immers na een data-, respectievelijk pariteitsbit, die 0 is, direct een startbit volgen, dan zou deze geen neergaande flank veroorzaken. Hierdoor bestaat het gevaar dat de ontvanger uit synchronisatie raakt. De stopbit zorgt ervoor dat de lijn hoog is bij elke startbit, zodat op elke startbit opnieuw kan worden gesynchroniseerd.

Zo'n patroon van stopbit, databits, pariteitsbit en stopbit wordt een frame genoemd.

In paragraaf 10.10 zult u kennismaken met de ACIA, een programmeerbare component, die zorg draagt voor de afhandeling van asynchroon serietransport met startbit en stopbit(s).

10.5.4 Handshaking

Bij communicatie treedt er buiten de synchronisatie nog een ander probleem op. Tot nu toe hebben we ons niet bekommerd om de verwerkingssnelheid van het randapparaat. Er is steeds

aangenomen, dat het randapparaat de gezonden data op tijd kan verwerken. In de praktijk is dit echter lang niet altijd het geval. Het randapparaat moet dan de mogelijkheid hebben om te signaleren dat het datatransport tijdelijk gestopt dient te worden. Dit probleem kan worden opgelost met behulp van handshaking.

Hierbij meldt het ene station dat het wil communiceren. Het andere station meldt terug dat het gereed is voor de communicatie. Hiervoor zijn twee zogenaamde handshake-lijnen nodig. Deze besturingslijnen kunnen softwarematig of hardwarematig worden bediend. In het laatste geval is extra hardware nodig. Het voordeel hierbij is dat de handshake-lijnen niet met instructies worden geset en gereset, zodat er minder tijd verloren gaat. In paragraaf 10.9 zal blijken dat de PIA in de zogenaamde handshake mode kan opereren.

Het is ook mogelijk om de (terug)melding geheel softwarematig te laten plaatsvinden via de datalijnen. De synchronisatie van de signalen moet dan evenwel op een andere wijze geschieden. Het datatransport begint nu bijvoorbeeld met een ENQ-sigitaal (ENquiry) om te vragen of de ontvanger gereed is. Er wordt dan gewacht, totdat het signaal ACK (positive ACKnowledgement) terugkomt. Dan begint het datatransport.

Men kan hier spreken van een software handshake.

Aan het einde van het transport kan de ontvanger terugmelden of de data al dan niet correct is overgekomen. Dit kan geschieden met de signalen ACK, respectievelijk NAK (Negative ACKnowledgement). Ook kunnen er voorzieningen getroffen zijn om het datatransport te onderbreken. Het blijft hier niet bij een eenvoudige melding en terugmelding, maar er is sprake van een communicatieprocedure met diverse mogelijkheden. Men spreekt hier over communicatieprotocollen.

ENQ, ACK, NAK en dergelijke zijn afzonderlijke karakters in ASCII (zie appendix A).

Handshaking kan geschieden op twee manieren:

1. pulsed handshake
2. interlocked handshake
 - return to zero
 - non return to zero

Pulsed handshake

Bij de pulsed handshake geschiedt de melding (bijvoorbeeld "hier is data") met behulp van een puls, net als bij strobing. Hierbij wordt het meldingssigitaal weggehaald, zonder zich te bekommeren om de terugmelding. De ontvanger moet maar zien dit signaal te detecteren en de data tijdig binnen te halen. Het verwerken van de data kan langzamer gebeuren, want de zender wacht met het sturen van de volgende melding "hier is data", totdat er een terugmelding is ontvangen. De terugmelding (bijvoorbeeld "data verwerkt") geschiedt eveneens door middel van een puls. Dit signaal kan tevens worden opgevat als "klaar voor ontvangst van nieuwe data". Ook hier bekommert men zich er

niet om of dit signaal aan de andere kant van het communicatiekanaal ontvangen wordt. De ontvanger wacht nu op een nieuwe puls "hier is data".

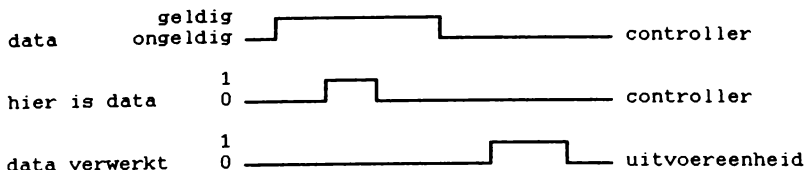


Fig. 10.15 Pulsed handshake bij uitvoer van data

Fig. 10.15 laat een pulsed handshake zien in geval van uitvoer. Stel dat een computer via een output controller gegevens wil uitvoeren naar een uitvoereenheid, bijvoorbeeld een printer. Als de data geldig is, geeft de controller een puls "hier is data". De uitvoereenheid moet deze data onmiddellijk inlezen. Daarna kan de data in alle rust worden verwerkt. De controller zal pas nieuwe data verzenden, nadat hij het signaal "data verwerkt" heeft ontvangen. Hij wacht dus op de terugmelding van de uitvoereenheid.

Fig. 10.16 toont een pulsed handshake in geval van invoer. Veronderstel dat een computer via een input controller data wil binnenhalen van een invoereenheid, bijvoorbeeld een optische schijfeenheid. Hier geeft de controller een puls af op de lijn "verzoek om data". Nadat dit signaal door de invoereenheid is ontvangen, zet deze de data op de datalijnen en wordt er een puls verzonden naar de controller via de lijn "data aanwezig". De controller moet deze data onmiddellijk inlezen en kan deze later op zijn gemak verwerken. Als de controller opnieuw gereed is voor het ontvangen van data, meldt hij dit met behulp van de puls "verzoek om data". Daarna zal er door de invoereenheid aan de andere kant opnieuw data worden verzonden, hetgeen wordt teruggemeld met de puls "data aanwezig".

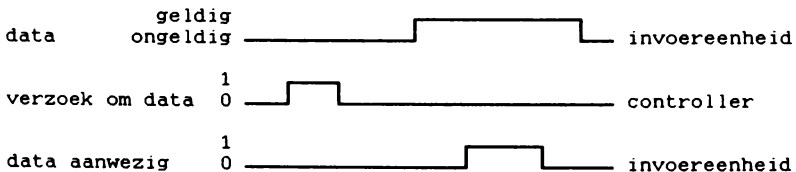


Fig. 10.16 Pulsed handshake bij invoer van data

Het nadeel van pulsed handshake is, dat de apparaten steeds op elkaar moeten letten, teneinde te voorkomen dat er een melding verloren gaat. De data moet onmiddellijk worden ingeklokt om deze later te kunnen verwerken. Gebeurt dit (net als bij strobing) door het handshake-sig-naal zelf, waarbij tevens een

flipflop wordt geset, dan gaat er bij de pulsed handshake geen informatie verloren.

Bij de pulsed handshake kan de toestand van de interface niet op elk moment worden bepaald. Het 0 zijn van beide handshake-lijnen kan duiden op de rusttoestand, maar ook op de situatie halverwege de handshake.

De methode van pulsed handshake is te vergelijken met het "overhandigen" van een vaas bloemen door te roepen "hier zijn bloemen", terwijl de vaas met bloemen in een grote boog naar de ontvanger wordt geworpen. Deze moet dan maar zien de vaas tijdig op te vangen. Vervolgens worden de bloemen ergens neergezet en wordt een lege vaas teruggeworpen onder het roepen van "bedankt". Nu is het de beurt aan de gever om de lege vaas op te vangen. Daarna kan deze eventueel weer gevuld worden met bloemen om ze daarna aan de ander te "overhandigen".

Interlocked handshake (R2)

Een elegantere wijze voor het overhandigen van bloemen is de vaas met bloemen in de richting van de ontvanger uit te strekken met de mededeling "hier zijn bloemen" en de vaas vast te houden, totdat de ontvanger de bloemen eruit heeft genomen onder het zeggen van "bedankt". Daarna wordt de lege vaas teruggetrokken en gaat de ontvanger de bloemen wegzetten. Vervolgens kan de gever eventueel een nieuwe vaas bloemen aanbieden.

Deze methode wordt toegepast bij de interlocked handshake (volledig overlappende handshake). Deze vorm van handshake vertoont veel overeenkomsten met het handenschudden in het dagelijkse leven. Hierbij neemt persoon A de hand van persoon B, die hierop zijn hand om die van A dichtknijpt. Nu kunnen de handen worden geschud. Vervolgens laat A los, waarop ook B los laat.

Hierna kan het handenschudden eventueel worden herhaald.

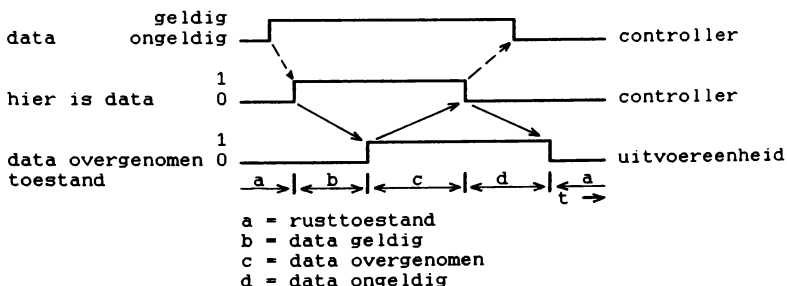


Fig. 10.17 Interlocked handshake bij uitvoer van data

Fig. 10.17 laat zien hoe interlocked handshaking plaatsvindt

bij uitvoer van data. De controller zet data gereed en meldt dit met het activeren (1 maken) van de lijn "hier is data". De uitvoereenheid zal op een zeker moment het actief zijn van deze lijn opmerken en de data overnemen. Hij meldt dit door de lijn "data overgenomen" actief te maken. Pas nadat de controller dit gedetecteerd heeft, zal deze de lijn "hier is data" inactief maken en de data weghalen. De controller wacht dus met het ongeldig verklaren van de data, totdat hij zeker weet dat de uitvoereenheid de data heeft overgenomen. De uitvoereenheid maakt de lijn "data overgenomen" pas inactief, na het constateren dat de controller "hier is data" inactief heeft gemaakt, dus na er zeker van te zijn dat de controller de terugmelding "data overgenomen" opgemerkt heeft. Als de controller ontdekt dat "data overgenomen" weer inactief is geworden, zal hij weer nieuwe data kunnen gaan verzenden. Er wordt hierbij keurig netjes op elkaars melding gewacht, zodat er geen informatie verloren kan gaan, ook niet als er even niet wordt opgelet.

Bovendien kan aan de toestand van de handshake-lijnen worden gezien in welke fase het datatransport zich bevindt:

- rusttoestand; er vindt geen datatransport plaats; het vorige transport is geheel beëindigd, waarbij de uitvoereenheid het ongeldig zijn van de data heeft bevestigd
- het datatransport is net gestart; de controller wacht op de terugmelding van de uitvoereenheid
- de uitvoereenheid heeft de data overgenomen
- de controller heeft de terugmelding ontvangen en beëindigt het datatransport

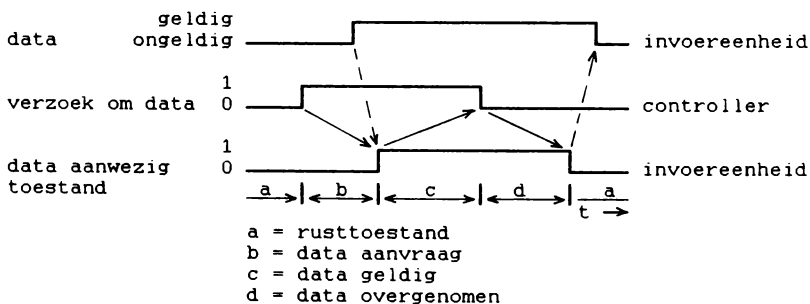


Fig. 10.18 Interlocked handshake bij invoer van data

Fig. 10.18 toont een interlocked handshake voor het invoeren van data. De controller vraagt nu om data door het activeren van de lijn "zend data a.u.b.". Deze lijn blijft actief, totdat de controller de melding "data aanwezig" ontvangt. De data wordt nu door de controller overgenomen, waarna deze de lijn "verzoek om data" inactief maakt. Hierop antwoordt de invoereenheid met het deactiveren van de lijn "data aanwezig". Als de controller de data heeft verwerkt, kan deze eventueel

- opnieuw om data verzoeken door de lijn "verzoek om data" te activeren, echter pas nadat de lijn "data aanwezig" inactief is. De vier toestanden van het datatransport zijn hier:
- rusttoestand; er vindt geen datatransport plaats; het vorige transport is geheel beëindigd, waarbij de data door de invoereenheid ongeldig is verklaard
 - de controller vraagt om data
 - de invoereenheid heeft dit verzoek ontvangen, data op de datalijnen gezet en deze data geldig verklaard
 - de controller heeft de terugmelding ontvangen en de data overgenomen

U hebt gezien dat er vier overgangen optreden bij een handshake met volledige overlapping. Aan het einde van elke handshakecyclus zijn de handshakelijnen beide weer in de 0-toestand. Vandaar dat men hier spreekt van een Return to Zero handshake.

Interlocked handshake (NRZ)

Elke overgang heeft een zekere tijd nodig. Voor datatransporten waarbij hoge snelheden vereist zijn, kunt u het aantal overgangen halveren door de zogenaamde overlappende NRZ-handshake (Non Return to Zero) toe te passen. Fig. 10.19 laat dit zien voor data-uitvoer.

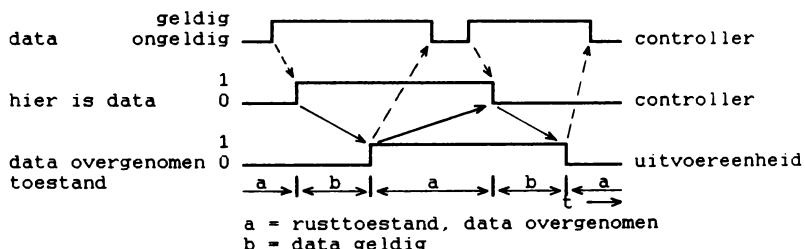


Fig. 10.19 Overlappende NRZ-handshake bij uitvoer van data

Door een verandering van het signaal "hier is data" geeft de controller aan, dat de data geldig is. Door het signaal "data overgenomen" te veranderen meldt de uitvoereenheid terug, dat de data is overgenomen. Na het ontvangen van dit signaal zal de controller de data weghalen en door nieuwe data vervangen. Hierbij is de data tijdelijk ongeldig. Heeft de data de nieuwe waarde bereikt, dan meldt de controller dit met het signaal "hier is data". Ook hier wordt dus telkens gewacht op het antwoordsignaal.

De twee toestanden zijn als volgt te beschrijven:

- rusttoestand; er is geen datatransport; de uitvoereenheid heeft gemeld, dat de vorige data is overgenomen
- de controller heeft de terugmelding van de uitvoereenheid ontvangen en heeft gemeld dat er nieuwe data gereed staat

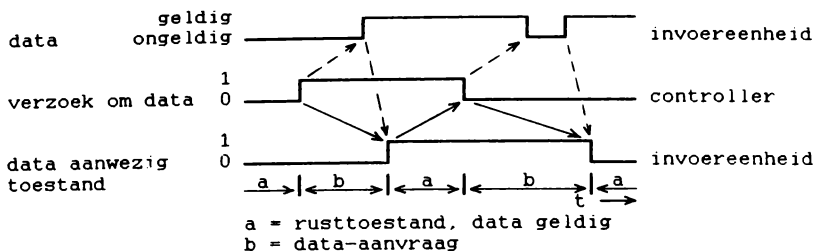


Fig. 10.20 Overlappende NRZ-handshake bij invoer van data

In fig. 10.20 wordt een overlappende NRZ-handshake getoond voor de invoer van gegevens. Nu vraagt de controller om data door het veranderen van het niveau van de lijn "verzoek om data". De invoereenheid reageert hierop door het gereed zetten van data en het geldig verklaren van de data door het signaal op de lijn "data aanwezig" te veranderen. Na het detecteren hiervan haalt de controller de data binnen en verzoekt eventueel om nieuwe data, hiermee tegelijkertijd mededelend, dat de vorige data is overgenomen. Ook hier kan er geen verlies van data optreden, omdat er telkens op een antwoordsignaal wordt gewacht.

De twee toestanden kunnen als volgt worden omschreven:

- rusttoestand; er is geen datatransport; de invoereenheid heeft voldaan aan het vorige verzoek om data te zenden
- de controller heeft te kennen gegeven data te willen ontvangen en heeft hiermee tegelijkertijd gemeld dat de vorige data overgenomen is

Bij een NRZ-handshake zijn de handshakelijnen in de rusttoestand aan elkaar gelijk (zie fig. 10.20).

Hoewel in principe twee lijnen voldoende zijn voor handshaking, bestaan er ook zogenaamde 3-lijns handshake-technieken. De derde handshake-lijn bewijst zijn diensten bij ingewikkeldere communicatieprotocollen. Deze vallen echter buiten het bestek van dit boek, zodat we hierop niet verder zullen ingaan.

Time-out

Bij handshaking moet men zich rekenschap geven van het feit dat het terugmeldingssignaal wel eens achterwege kan blijven. Dit is bijvoorbeeld het geval als het randapparaat niet is ingeschakeld. Om een communicatiekanaal hierdoor niet langdurig te blokkeren met het vergeefs wachten op antwoord, moet de communicatie op een gegeven ogenblik worden afgebroken. Men spreekt hierbij van time-out. De tijd gedurende welke gewacht wordt op het terugmeldingssignaal, moet worden bewaakt. Bij overschrijding van een bepaalde limiet wordt het wachten gestaakt en de communicatie verbroken. Deze tijd kan softwarematig worden afgeteld in een wachtlus. Als er echter werkzaamheden zijn, die tijdens het wachten op antwoord kunnen

worden uitgevoerd, is het beter om gebruik te maken van een zogenaamde watchdog timer. Deze timer wordt op een bepaald moment gestart om na een ingestelde tijd een interrupt af te geven. Wordt de watchdog timer gestart bij het geven van een melding, dan genereert deze een interrupt als de maximaal toegestane wachttijd op het antwoord verstreken is. Komt het antwoord binnen deze tijd, dan wordt de timer stopgezet, zodat de interrupt uitblijft.

10.6 Programmed I/O met de PIA

De PIA, Peripheral Interface Adapter, is een IC, behorend bij de MC68XX processoren van Motorola. Het typenummer van de PIA is MC6821. De MC68A21 en de MC68B21 zijn snellere versies van de MC6821. Ze werken met klokfrequenties van 1.5 MHz en 2.0 MHz in plaats van met 1.0 MHz.

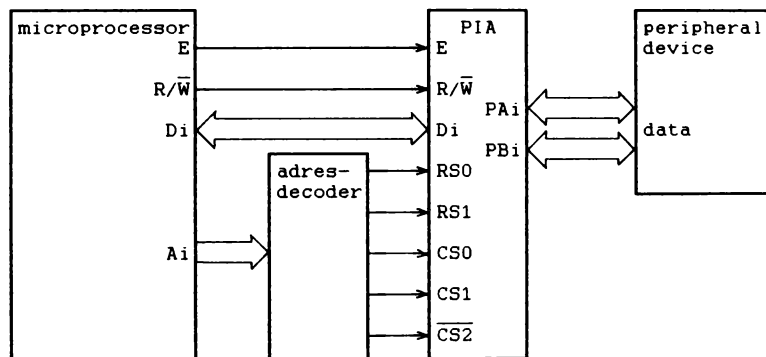


Fig. 21 PIA als communicatiechip tussen processor en periferie

Communicatie tussen PIA en processor

Voor de communicatie met de microprocessor heeft de PIA een 8-bits bidirectionele databus D0 t/m D7. De datasignalen kunnen derhalve zowel input als output zijn, zodat de microprocessor informatie naar de PIA kan sturen, alsook informatie vanuit de PIA kan binnenhalen. Om aan te geven of er gelezen of geschreven wordt in de PIA is er een $R/\bar{W}$ -ingang.

Voorts heeft de PIA een klokingang E, teneinde het datatransport tussen microprocessor en PIA te synchroniseren. Dit betekent dat de communicatie tussen PIA en microprocessor synchroon is. Om de PIA te kunnen selecteren zijn er drie chip select ingangen CS0, CS1 en $\overline{CS2}$. De PIA is alleen dan toegankelijk vanuit de microprocessor als geldt $CS0=CS1=1$ en $\overline{CS2}=0$. De PIA-ingangen $R/\bar{W}$ en E zijn verbonden met de processoruitgangen $R/\bar{W}$ en E.

De chip select ingangen zijn via een externe adresdecoder verbonden met de adresbus van de microprocessor (zie fig. 10.21). Zijn de chip select ingangen van de PIA niet alle drie geactiveerd, dan zijn de databusaansluitingen van de PIA geblokkeerd.

Communicatie tussen PIA en periferie

Voor de communicatie met de periferie heeft de PIA twee 8-bits poorten PA0 t/m PA7 en PB0 t/m PB7. Deze zestien aansluitingen kunnen elk afzonderlijk worden geprogrammeerd als input of als output. Hiertoe bevat de PIA twee 8-bits data direction registers, DDRA en DDRB. Als bit $DDRA_i$ gelijk is aan 1, is de overeenkomstige aansluiting PA_i ingesteld als output. Voor $DDRA_i=0$ functioneert PA_i als input. Als $DDRB_i=0$ is PB_i als input ingesteld, als $DDRB_i=1$ is PB_i output. De aansluitingen PA_i en PB_i die als output zijn geprogrammeerd, krijgen hun informatie vanuit de overeenkomstige bits van het Output Register A, respectievelijk B, ook wel genoemd Peripheral Register A of B (PRA, PRB). De Peripheral Registers worden vanuit de microprocessor geladen tijdens een schrijfofdracht ($R/\bar{W}=0$). De aansluitingen PA_i en PB_i die als input zijn ingesteld, ontvangen hun informatie van het peripheral device. Leest de microprocessor op PA_i of PB_i ($R/\bar{W}=1$), dan verschijnt de informatie van PA_i , respectievelijk PB_i rechtstreeks op de databus Di . Op deze wijze wordt zowel de inputinformatie als de outputinformatie op de PA-poort, respectievelijk de PB-poort door de microprocessor binnengehaald (zie fig. 10.22). Als $DDRA_i$ of $DDRB_i$ gelijk is aan 0, wordt de inputdata van de periferie binnengehaald in de processor. Dit geschieht via de datalijnen Di . Is $DDRA_i$ of $DDRB_i$ gelijk aan 1, dan wordt de outputdata van PRA_i, respectievelijk PRB_i (via Di) teruggelaten in de processor. De door de processor gelezen data bestaat dus zowel uit inputsignalen als uit outputsignalen, afhankelijk van de inhoud van DDRA en DDRB.

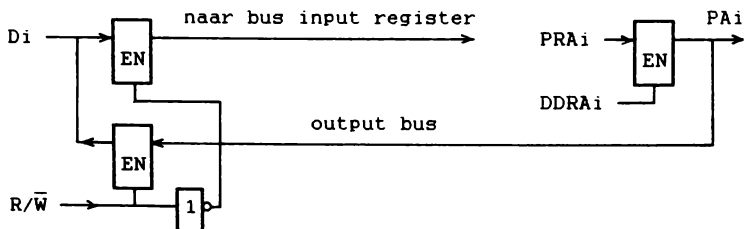


Fig. 10.22 Richting van datatransport bij PIA

Samenvattend:

$DDRA_i = 0$: PA_i is input
 $DDRA_i = 1$: PA_i is output

$DDRB_i = 0$: PB_i is input
 $DDRB_i = 1$: PB_i is output

De synchronisatie tussen PIA en periferie kan op diverse wijzen plaatsvinden, zoals in dit hoofdstuk nog ter sprake zal komen. De wijze waarop dit gebeurt, wordt vastgelegd in de Control Registers CRA en CRB. Bovendien bevatten deze registers ook informatie omtrent de status van de PIA. Fig. 10.23 toont het blokdiagram van de PIA.

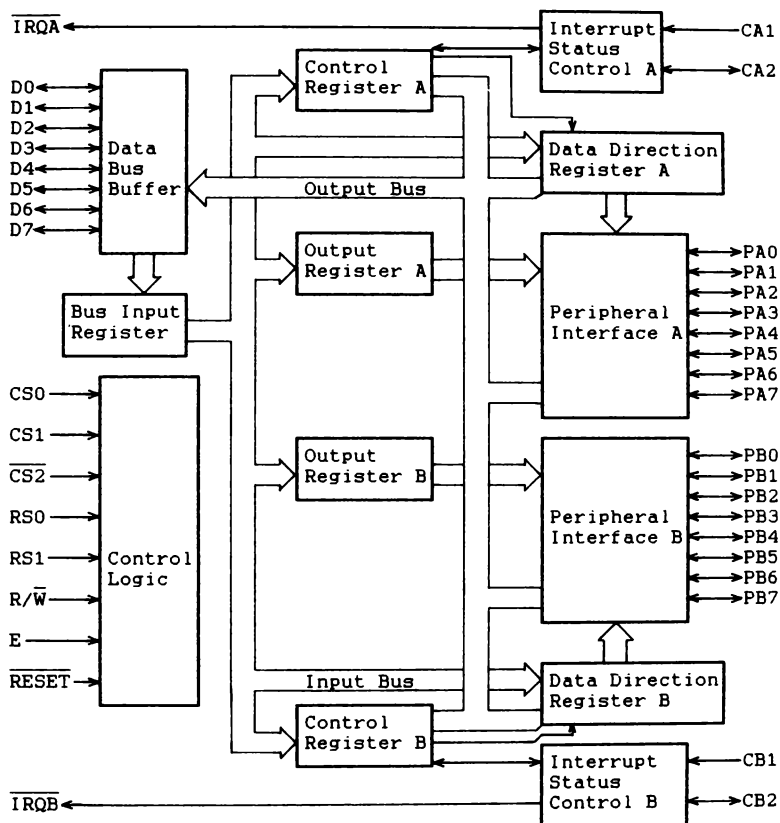


Fig. 10.23 Blokschema van de PIA

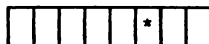
Een 0 op de RESET-ingang van de PIA reset alle PIA-registers, dus ook DDRA en DDRB. Dit betekent dat alle PA- en PB-aansluitingen als input worden ingesteld. Hiermee wordt voorkomen dat er na een reset grote stromen kunnen optreden. Er zouden ten onrechte PA- of PB-aansluitingen als output ingesteld kunnen worden, terwijl er op die lijnen periferiezenders aangesloten zijn. De RESET-ingang van de PIA is doorgaans aangesloten op hetzelfde signaal dat ook de RESET-

ingang van de microprocessor aanstuurt, zodat de PIA bij het inschakelen van de voedingsspanning in een veilige stand komt te staan (alle periferielijnen worden input).

2

Registerselectie

CRA, CRB



Samenvattend kunnen we zeggen dat de PIA twee programmeerbare Data Direction Registers en twee programmeerbare Control Registers bevat. Daarnaast moet de microprocessor kunnen communiceren met de twee outputpoorten PA en PB. Bij het schrijven gebeurt dit via de twee Peripheral Registers PRA en PRB, bij het lezen rechtstreeks. Daar de databus slechts 8 bits breed is, kan er met één register of met één outputpoort tegelijk worden gecommuniceerd. Voor de selectie hiervan is de PIA uitgerust met twee Register Select ingangen RS0 en RS1. Hiermee kan slechts een keuze worden gemaakt van 1 uit 4, in plaats van 1 uit 6. Om toch de 6 registers te kunnen selecteren, functioneren de bits 2 van CRA en CRB als extra selectiebits. Fig. 10.24 toont de registerselectie.

RS1	RS0	geselecteerd register
0	0	DDRA (als CRA2=0) en PRA (als CRA2=1)
0	1	CRA
1	0	DDRB (als CRB2=0) en PRB (als CRB2=1)
1	1	CRB

Fig. 10.24 Selectie van de PIA-registers

Deze oplossing heeft als voordeel dat een PIA slechts vier plaatsen in de memory map kost. Met drie register select ingangen zouden er dit 6 of 8 zijn, afhankelijk of de twee overbodige codes intern in de PIA worden gebruikt of niet. Bovendien zou er een extra pen nodig zijn op de chip of er zou een chip select ingang minder moeten zijn. Dit zou weer leiden tot een (kleine) vergroting van de adresdecoder.

Het nadeel van de gekozen oplossing met slechts twee register select ingangen is de omslachtige initialisatie van de PIA. Er kan immers niet zomaar in PRA en PRB of in DDRA en DDRB worden geschreven. Hiertoe moeten eerst CRA2 en CRB2 de juiste waarde krijgen.

Voorbeeld 10.5

Veronderstel dat de processor 13 signalen ontvangt van diverse sensoren (opnemers) en 3 besturingssignalen moet zenden naar actuatoren (kleppen, motoren en dergelijke). Dit kan geschieden via een PIA. Hiertoe moet de PIA eerst op de juiste wijze worden ingesteld. Indien de uitgangssignalen worden verbonden met PA5, PA6 en PA7 en de ingangssignalen op PB0 t/m PB7 en op PA0 t/m PA4, dan ziet de initialisatieroutine van de PIA er als volgt uit:

```

100 *****
110 * Subroutine INIPIA *
120 *****
130
140 *Progr.: J.G. Rouland
150 *Revisie 0    Datum: 30-08-1986
160 *
170 *INIPIA stelt PA5, PA6 en PA7 in als output en de overige
180 *PA- en PB-lijnen als input. Daarna worden PRA en PRB
190 *geadresseerd.
200 *
210 INIPIA  CLR    CRA            adresseer de registers
220                CLR    CRB            DDRA en DDRB
230                LDA    #%11100000    stel PA7 t/m PA5 in als output
240                STA    DDRA          en PA4 t/m PA0 als input
250                CLR    DDRB          stel PB7 t/m PB0 in als input
260                LDA    #%00000100    adresseer
270                STA    CRA            de registers
280                STA    CRB            PRA en PRB
290                RTS

```

In de regels 210 en 220 wordt bit 2 van de CRA en CRB gelijk aan 0 gemaakt. Hierdoor kan DDRA, respectievelijk DDRB worden geselecteerd. Vervolgens worden de bits DDRA7 t/m DDRA5 geset en DDRA4 t/m DDRA0 gereset (regel 230 en 240). Hierdoor wordt PA7 t/m PA5 output en PA4 t/m PA0 input. Door het resetten van DDRB wordt poort PB helemaal input (regel 250). In de regels 260 t/m 280 wordt bit 2 van CRA en CRB geset, zodat de registers PRA en PRB geselecteerd kunnen worden. Natuurlijk moeten met equate directives geheugenadressen worden toegekend aan de symbolische adressen PRA, PRB, DDRA, DDRB, CRA en CRB. PRA en DDRA moeten hetzelfde adres krijgen, evenals PRB en DDRB.

Het is meestal handig de adresselectie zodanig te kiezen dat de registers PRA en PRB in de memory map naast elkaar liggen. In dat geval is het mogelijk om met een enkele instructie de inhoud van PRA en PRB te wijzigen, dan wel te lezen. Om dit te bereiken wordt de adreslijn A0 verbonden met de ingang RS1, terwijl A1 wordt verbonden met RS0. Zo ontstaat bij opeenvolgende adressen de situatie als gegeven in fig. 10.25.

A1	A0	RS1	RS0	geselecteerd register
0	0	0	0	DDRA of PRA
0	1	1	0	DDRB of PRB
1	0	0	1	CRA
1	1	1	1	CRB

Fig. 10.25 Mogelijke adressering van de PIA-registers

Wordt nu bijvoorbeeld DDRA/PRA geadresseerd via adres \$FF04, dan is DDRB/PRB bereikbaar via adres \$FF05. CRA en CRB worden nu geadresseerd via \$FF06 en \$FF07. Met behulp van de instructie LDD PRA kan nu de informatie van de 16 periferiesluitingen worden binnengehaald. Bij voorbeeld 10.5 kunnen de outputsignalen eventueel worden gemaskeerd met de instructie ANDA #00011111, nadat deze samen met de inputsignalen in PRA gelezen zijn.

Adresselectie

Voor het selecteren van een PIA zou één chip select ingang voldoende zijn. Doordat de PIA is uitgerust met meerdere chip select ingangen, is de externe adresdecoder eenvoudiger. Moet er via 64 lijnen worden gecommuniceerd met de periferie, dan kan dit geschieden via 4 PIA's. Fig. 10.26 laat een mogelijke selectie van deze PIA's zien. Hierbij zijn alle PIA-registers te bereiken via de adressen \$FF00 t/m \$FF0F.

Van de Y-uitgangen van de decoder in fig. 10.26 heeft slechts één uitgang de waarde 1. De overige uitgangen zijn 0. Op deze wijze wordt slechts één PIA tegelijk geselecteerd.

Als de waarde van S1S0 gelijk is aan i, heeft uitgang Yi van de decoder de waarde 1. Dus voor S1=1 en S0=0 is S1S0 = %10 = 2, dus geldt Y2=1 en Y0=Y1=Y3=0, zodat PIA2 wordt geselecteerd door de adressen \$FF08 t/m \$FF0B. Bedenk dat S1S0 = A3A2 en dat A3 het gewicht 8 heeft.

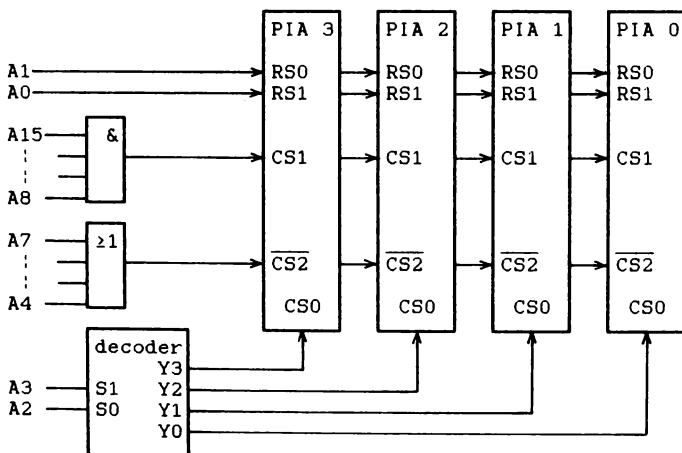


Fig. 10.26 Adresdecoder voor vier PIA's

De adresselectie is meestal instelbaar. Dit kan worden bereikt met behulp van schakelaars of straps (doorverbindslusjes, die men kan aanbrengen tussen twee pennen). Hiermee kan worden ingesteld of de adressignalen rechtstreeks, dan wel

geïnverteerd worden aangeboden aan de adresdecoder (zie fig. 10.27). Op deze wijze kan met behulp van een vaste adresdecoder toch elk willekeurig adres worden geselecteerd. Dit is erg nuttig bij het opbouwen van modulaire systemen. Hierbij is er een back panel (achterbord), ook wel bus board genoemd, dat een aantal connectors bevat, waarvan de overeenkomstige pennen via printsporen zijn doorverbonden. Deze printsporen vormen de microprocessorbussen. Het maakt nu niet uit in welke connector een bepaalde kaart wordt gestopt. Zo bestaan er bijvoorbeeld microprocessorkaarten, ROM-kaarten, RAM-kaarten en diverse soorten I/O-kaarten. Ook zijn er wel gecombineerde kaarten, bijvoorbeeld microprocessor/geheugenkaarten. Door nu de geheugenkaarten en de I/O-kaarten te voorzien van instelbare adresselecties kan de geheugenindeling vrij gekozen worden. Bovendien kan men meerdere identieke kaarten in het systeem toepassen door ervoor te zorgen dat ze via verschillende adressen geselecteerd worden. Zo ontstaat een modulaair systeem, dat eenvoudig is uit te breiden door modules (kaarten) toe te voegen.

In paragraaf 10.2 is reeds opgemerkt dat elk adres uniek moet zijn. Zorg ervoor dat de adresselectie zodanig wordt ingesteld, dat er met elk willekeurig adres slechts één locatie (van geheugen of I/O) wordt geselecteerd.

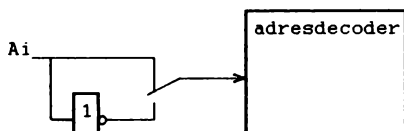


Fig. 10.27 Instelbare adresselectie

Om het aantal schakelaars of straps te reduceren, kan men een beperkt aantal adressignalen al dan niet inverteren. Het is dan niet meer mogelijk om de adresselectie in de gehele memory map in te stellen.

Voorbeeld 10.6 (Instelbare adresselectie)

Veronderstel we hebben een I/O-kaart met vier PIA's (zie fig. 10.26). Indien we het basisadres, het laagste adres van een module, instelbaar willen maken in het geheugengebied van adres \$FF00 t/m \$FFF0, dan kan dit met behulp van vier schakelaars. De adressignalen A4 t/m A7 moeten hiertoe geïnverteerd kunnen worden. Zij zorgen voor de keuze tussen \$XX0X en \$XXFX. Deze keuze moet variabel zijn.

De adressignalen A8 t/m A15 bepalen de keuze van de adresselectie tussen \$00XX en \$FFXX. In dit voorbeeld moet dit ingesteld worden op \$FFXX (zie fig. 10.28).

A0 t/m A3 verzorgen de selectie van de vier PIA's op de kaart alsmede de selectie van de bijbehorende registers. Dit gebeurt

via de adressen \$XXX0 en \$XXXF. In fig. 10.28 is het basisadres met behulp van straps ingesteld op \$FF90.

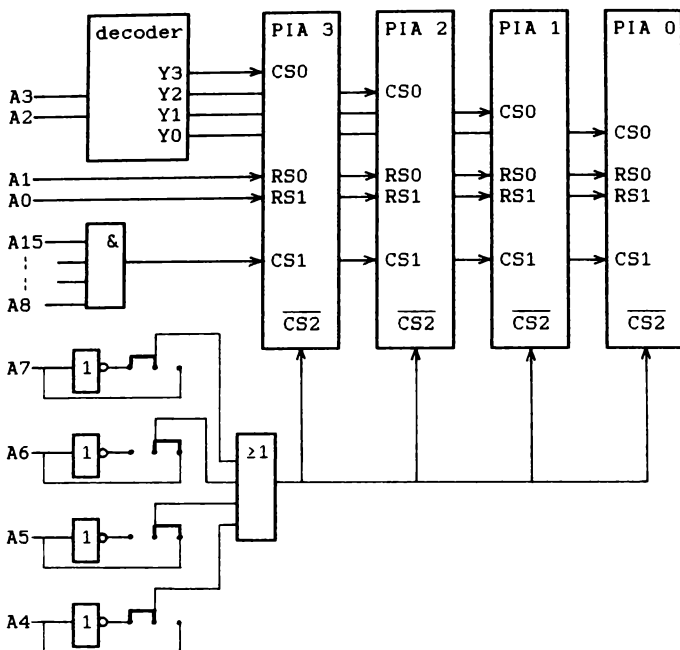


Fig. 10.28 Module met gedeeltelijk instelbare adresdecoder

Voorbeeld 10.7 (ADC-interface)

Via een Analooq-Digitaal-Converter moet een analoge meetwaarde worden opgeslagen in het computergeheugen. Veronderstel dat de ADC een 12-bits output heeft. Het omzettingsproces van de analoge waarde in de bijbehorende digitale code wordt gestart door een positieve puls op de ADC-ingang SOC (Start Of Conversion). Bij het 0 worden van SOC, wordt de uitgang EOC (End Of Conversion) gelijk aan 0 en begint de conversie. Aan het einde van de conversie wordt EOC gelijk aan 1 ten teken dat de uitgangscade gereed is.

De communicatie tussen de microprocessor en de ADC kan verlopen via een PIA. Bij de aansluitingen als gegeven in fig. 10.29 kan de subroutine INIPIA worden gebruikt om de PIA op de juiste wijze in te stellen.

Fig. 10.30 laat het structogram zien behorende bij de communicatie tussen microprocessor en PIA.

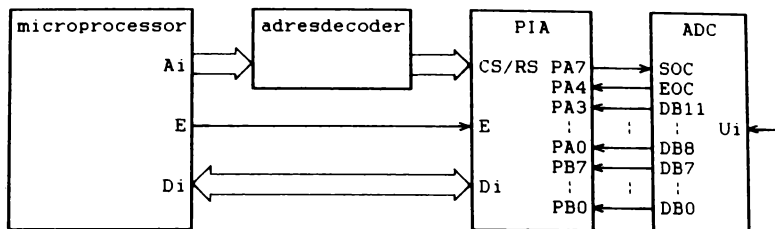


Fig. 10.29 Communicatie tussen microprocessor en ADC

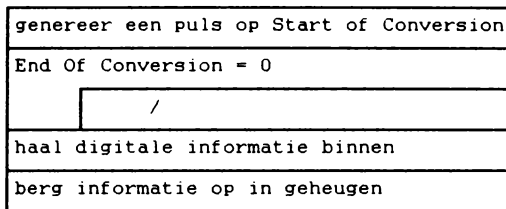


Fig. 10.30 Structogram voor de communicatie tussen microprocessor en ADC via een PIA

De subroutine die deze communicatie verzorgt en de meetwaarde opbergt in de geheugenlocaties \$8F00 en \$8F01, ziet er als volgt uit:

```

100 *****
110 * Subroutine ADC-input *
120 *****
130 *
140 *Progr.: J.G. Rouland
150 *Revisie 0   Datum: 03-09-1986
160 *
170 *ADCINP communiceert via een PIA met een ADC (zie fig.
180 *10.29). De PIA heeft als basisadres $FF04.
190 *Er wordt een SOC-puls gegenereerd om de conversie te
200 *starten.
210 *Na het 1 worden van EOC wordt de 12-bits uitgangscod van
220 *de ADC op de geheugenlocaties $8F00 en $8F01 geplaatst.
230 *
240 RESULT EQU    $8F00          adres voor digitale output
250 PRA     EQU    $FF04
260 PRB     EQU    $FF05
270 SOC     EQU    %10000000
280 EOC     EQU    %00010000
290 *
300 ADCINP LDA    #SOC
310          STA    PRA          set SOC
  
```


320	CLR	PRA	reset SOC
330	LEES	LDD PRA	lees EOC en uitgangscode
340	BITA	#EOC	test EOC
350	BEQ	LEES	spring als EOC=0
360	ANDA	#\$0F	berg de 12-bits
370	STD	RESULT	code op in het geheugen
380	RTS		

In de regels 300, 310 en 320 wordt een SOC-puls gegeven. Deze puls duurt 7 μ s, omdat de instructie CLR PRA 7 μ s kost. Moet de SOC-puls breder zijn, dan moet er een wachtlus worden opgenomen tussen regel 310 en regel 320.

De instructie LDD PRA (regel 330) heeft tot gevolg dat accumulator A wordt geladen met de inhoud van PRA en accumulator B met de inhoud van PRB. A en B samen vormen immers register D.

In regel 360 worden de vier meestwaardige bits van A en dus ook van D gereset, zodat D de waarde van de uitgangscode van de ADC bevat.

Indien EOC wordt aangesloten op PA7 en SOC bijvoorbeeld op PA4, kan regel 340 vervallen en moet in regel 350 BEQ worden vervangen door BPL. Er wordt dan eveneens teruggesprongen zolang (D) positief is, dus zolang EOC=0 is. Het programma is nu korter. U ziet hieruit dat een handige keuze van de aansluitingen van de signalen, het programma kan vereenvoudigen.

In plaats van te wachten op EOC=1, kan een wachtlus worden ingebouwd met een wachttijd die tenminste de maximale conversietijd bedraagt. Hierna kan de uitgangscode van de ADC worden gelezen en opgeborgen. In het geval de ADC geen EOC-sigitaal heeft, moet deze methode worden toegepast.

De ADC kan ook via adresseerbare registers op de processor worden aangesloten. Vaak zijn deze registers geïntegreerd in de ADC, zodat de ADC rechtstreeks op de processorbussen kan worden aangesloten.

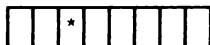
10.7 Interrupt-I/O met de PIA

In paragraaf 10.6 hebt u gezien hoe u met behulp van een Peripheral Interface Adapter programmed I/O kunt plegen. In deze paragraaf zal blijken dat de PIA erg geschikt is voor het bedrijven van interrupt-I/O.

5

CA2 en CB2 instellen

CRA, CRB



CA1 en CB1 zijn interruptingangen (zie ook fig. 10.23).

De PIA-aansluitingen CA2 en CB2 zijn te programmeren als

interruptingangen of als uitgangen. Dit geschiedt onder besturing van bit 5 van de Control Registers:

- CRA5 = 0: CA2 is interruptingang
- CRA5 = 1: CA2 is uitgang
- CRB5 = 0: CB2 is interruptingang
- CRB5 = 1: CB2 is uitgang

Actieve interruptflanken instellen CRA, CRB

			0	*				1
--	--	--	---	---	--	--	--	---

De interruptingangen zijn flankgevoelig. Dit wil zeggen dat de PIA een interrupt detecteert als er een overgang van 0 naar 1, dan wel van 1 naar 0, op deze ingang wordt aangeboden. Welke van deze twee flanken actief is, kan in de PIA worden ingesteld. Dit gebeurt door de control registers CRA en CRB met de juiste gegevens te laden:

- CRA1 = 0: neergaande CA1-flank is actief
- CRA1 = 1: opgaande CA1-flank is actief
- CRA4 = 0: neergaande CA2-flank is actief (mits CRA5 = 0)
- CRA4 = 1: opgaande CA2-flank is actief (mits CRA5 = 0)
- CRB1 = 0: neergaande CB1-flank is actief
- CRB1 = 1: opgaande CB1-flank is actief
- CRB4 = 0: neergaande CB2-flank is actief (mits CRB5 = 0)
- CRB4 = 1: opgaande CB2-flank is actief (mits CRB5 = 0)

In fig. 10.31 wordt dit symbolisch weergegeven voor het A-deel van de PIA. Een soortgelijk schema geldt voor het B-deel van de PIA.

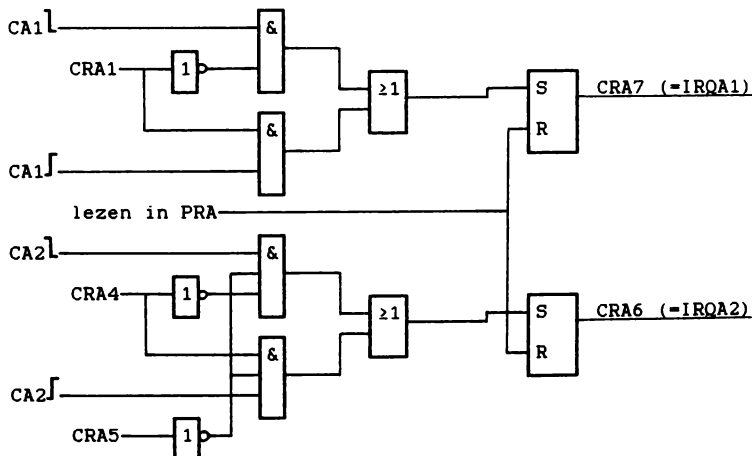


Fig. 10.31 Symbolische voorstelling van het setten en resetten van de interrupt request bits IRQA1 en IRQA2 van de PIA

Interrupts onthouden en resetten

CRA, CRB

*	*	0					
---	---	---	--	--	--	--	--

Indien de PIA een interrupt detecteert, wordt dit vastgelegd in de control registers CRA en CRB.

Bit 7 van CRA wordt geset, indien er een interrupt binnenkomt via de CA1-input van de PIA. Deze bit CRA7 wordt derhalve IRQA1 genoemd (Interrupt ReQuest van CA1). Evenzo wordt CRA6 geset bij een interrupt op de ingang CA2. CRA6 is dus de bit IRQA2. Zo is CRB7 de bit IRQB1 en CRB6 is IRQB2.

Is bit CRA1 gelijk aan 0, dan wordt IRQA1 (= CRA7) geset op de neergaande flank van CA1. Als CRA1=1 wordt IRQA1 geset op de opgaande flank van CA1. Zo wordt IRQA2 (= CRA6) geset op de neergaande flank van CA2, mits CRA4 = 0, en op de opgaande flank van CA2, mits CRA4 = 1 (zie fig. 10.31).
Op dezelfde wijze bepaalt CRB1 de actieve flank van CB1 en CRB4 de actieve flank van CB2.

De bits IRQA1, IRQA2, IRQB1 en IRQB2 kunnen alleen worden geset ten gevolge van de bijbehorende interrupts!

IRQA1 en IRQA2 worden alleen gereset bij het lezen van het peripheral register PRA! Zo worden de bits IRQB1 en IRQB2 alleen gereset bij het lezen van PRB! Dit hoeft niet noodzakelijk een load-instructie te zijn. Bedenk dat ook bij de instructies CMPB PRA, TST PRB, enzovoorts, in de bedoelde registers wordt gelezen. Ook al deze instructies hebben dus het resetten van de corresponderende interrupt request bits in de control registers tot gevolg. Dit gebeurt zelfs bij de clear-instructies CLR PRA en CLR PRB, omdat hier eerst de informatie in PRA en PRB wordt gelezen, daarna 0 gemaakt en tenslotte wordt deze 0 teruggeschreven in PRA, respectievelijk PRB. Alleen een store-opdracht op PRA en PRB heeft geen effect op de interrupt request bits!

Ook het rechtstreeks lezen of schrijven in de control registers zal de waarde van interrupt request bits CRA6, CRA7, CRB6 en CRB7 niet veranderen!

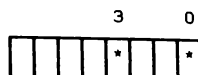
Samenvattend geldt:

- CRA7 (= IRQA1) := 1 bij actieve CA1-flank
- CRA7 (= IRQA1) := 0 bij lezen in PRA
- CRA6 (= IRQA2) := 1 bij actieve CA2-flank (mits CRA5 = 0)
- CRA6 (= IRQA2) := 1 bij lezen in PRA
- CRB7 (= IRQB1) := 1 bij actieve CB1-flank
- CRB7 (= IRQB1) := 0 bij lezen in PRB
- CRB6 (= IRQB2) := 1 bij actieve CB2-flank (mits CRB5 = 0)
- CRB6 (= IRQB2) := 1 bij lezen in PRB

Door de registers CRA en CRB te lezen en van beide de bits 6 en 7 te testen, kan de microprocessor achterhalen op welke interruptingangen van de PIA een interrupt is gedetecteerd.

Interrupts blokkeren

CRA, CRB



Onder besturing van bits van de control registers kan de informatie van de interrupt request bits al dan niet aan de $\overline{\text{IRQ}}$ -ingang van de microprocessor worden doorgegeven. Dit gebeurt via de uitgangen $\overline{\text{IRQA}}$ en $\overline{\text{IRQB}}$ van de PIA. Deze uitgangen mogen met elkaar worden doorverbonden, zodat er een wired-and ontstaat. De informatie van IRQA1 en IRQA2 wordt via de uitgang $\overline{\text{IRQA}}$ naar buiten uitgevoerd, terwijl IRQB1 en IRQB2 via $\overline{\text{IRQB}}$ worden verzonden.

Elke IRQ -bit kan afzonderlijk worden doorgegeven naar buiten of niet. Bit CRA0 is de interrupt enable bit van de interrupt request IRQA1 . Dit wil zeggen dat IRQA1 alleen wordt doorgegeven aan de microprocessor (via de uitgang $\overline{\text{IRQA}}$) als $\text{CRA0}=1$. Is $\text{CRA0}=0$, dan wordt de waarde van IRQA1 niet uitgevoerd. Zo zijn de bits CRA3 , CRB0 en CRB3 de interrupt enable bits van de interrupt requests IRQA2 , IRQB1 en IRQB2 . Fig. 10.32 geeft dit weer voor de interrupt requests van het A-deel van de PIA. Voor het B-deel geldt een soortgelijk schema, dus:

$\text{CRA0} = 0$: IRQA1 is disabled;	$\text{CRB0} = 0$: IRQB1 is disabled
$\text{CRA0} = 1$: IRQA1 is enabled;	$\text{CRB1} = 1$: IRQB1 is enabled
$\text{CRA3} = 0$: IRQA2 is disabled;	$\text{CRB3} = 0$: IRQB2 is disabled
$\text{CRA3} = 1$: IRQA2 is enabled;	$\text{CRB3} = 1$: IRQB2 is enabled

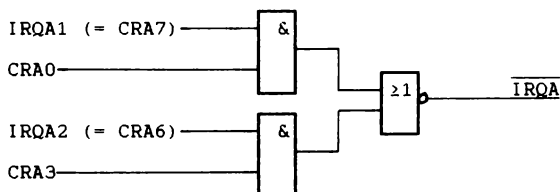


Fig. 10.32 De interrupt enable bits CRA0 en CRA3 van de PIA

Een binnenkomende interrupt op de interruptingangen van de PIA heeft altijd het zetten van de bijbehorende interrupt request bit tot gevolg, ook al is de interrupt enable bit gereset! De interrupt enable bit zorgt er alleen voor dat de betreffende interrupt al of niet aan de microprocessor wordt doorgegeven. Op deze wijze is het mogelijk sommige interrupts door te laten en andere te blokkeren.

Wordt de interrupt mask bit I in het condition code register van de microprocessor geset, dan worden alle IRQ -interrupts geblokkeerd!

Binnenkomende interrupts in de PIA worden altijd opgeslagen in de interrupt request bits van de control registers. Zodoende

gaan er geen interrupts verloren. Indien de processor binnenkomende interrupts blokkeert, kunnen ze later toch nog worden gedetecteerd, ook al is de aanvraag zelf reeds verdwenen. Door middel van polling van de interrupt request bits kunnen deze aanvragen te allen tijde op basis van programmed I/O worden afgehandeld. Door het opheffen van de blokkering kunnen ze alsnog op basis van interrupt-I/O worden afgehandeld. De interrupts worden opgeslagen in de bits 6 en 7 van CRA en CRB. Deze bits worden pas gereset bij het lezen in PRA, respectievelijk in PRB.

In beide gevallen wordt de interrupt request gereset door het lezen van het bijbehorende peripheral register. Op deze wijze meldt de processor aan de PIA dat de interrupt wordt afgehandeld. De interrupt request wordt hierbij automatisch gereset. De terugmelding aan het peripheral device moet (indien nodig) op een andere wijze geschieden. Hiertoe kan bijvoorbeeld een bepaalde code worden verzonden via de PA- of PB-lijnen.

Voorbeeld 10.8 (ADC-interface op basis van interrupt-I/O)

De communicatie tussen microprocessor en ADC geschiedde in voorbeeld 10.7 op basis van programmed I/O. Dit kan echter ook gebeuren op basis van interrupt-I/O. Hiertoe moeten we het signaal EOC verbinden met een van de interruptingangen van de PIA, bijvoorbeeld CA1. Verder dient de PIA op de juiste wijze te worden geïnitieëerd. De interruptvector moet worden geladen met het startadres van de interrupt service routine en de interrupt mask bit 1 van de microprocessor moet 0 zijn. In het hoofdprogramma wordt een SOC gegeven, waarna het programma wordt vervolgd.

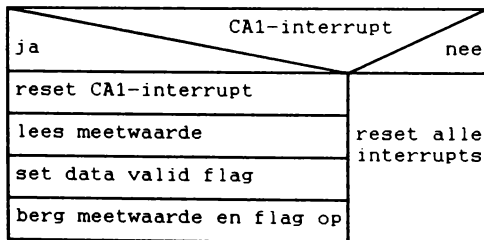


Fig. 10.33 Structogram van ISRADC voor communicatie met ADC

In de interrupt service routine wordt de meetwaarde binnengehaald en in het geheugen gezet (zie fig. 10.33). Op een gegeven moment zal deze meetwaarde in het hoofdprogramma worden verwerkt. Er zal dan evenwel bekend moeten zijn of er een interrupt is geweest of niet, met andere woorden, of er een geldige meetwaarde in het geheugen staat. Dit kan worden aangegeven met behulp van een data valid flag. Is deze flag geset, dan is de data in het geheugen geldig. Is de flag gereset, dan geldt de data niet. De data bestaat uit 12 bits.

Dit betekent dat de vier meestwaardige bits niet gebruikt worden, zodat we bijvoorbeeld bit 15 als flag kunnen gebruiken. Het programma ziet er als volgt uit:

```

100          ORG      $3A00
110  IRQVEC  EQU      $FFF8
120  RESULT  EQU      $8F00
130  PRA      EQU      $FF04
140  PRB      EQU      $FF05
150  DDRA     EQU      PRA
160  DDRB     EQU      PRB
170  CRA      EQU      $FF06
180  CRB      EQU      $FF07
190  *
200  PROG     ORCC     #$10          blokkeer IRQ
210          LDX      #ISRADC       laad interruptvector met
220          STX      IRQVEC        startadres van ISR
230          BSR      PIA          initialiseer de PIA
240          CLR      RESULT        reset de data valid flag
250          ANDCC     #$EF         enable IRQ
260          --
270          --
280          LDA      #$80          geef
290          STA      PRA           een
300          CLR      PRA           SOC-puls
310          --
320          --
330  FLAG     LDD      RESULT        lees de meetwaarde
340          BPL      FLAG          en kijk of deze geldig is
350          CLR      RESULT        reset data valid flag
360          ANDA      #$0F         reset 4 meestwaardige databits
370          --
380          --
390          SWI
400  *
410  *****
420  * PIA-INSTELLING *
430  *****
440  *
450  *PA0 t/m PA6 en PB0 t/m PB7 worden ingesteld als input, PA7
460  *als output.
470  *De CA1-interrupt wordt actief op de opgaande flank en de
480  *interrupt wordt ge-enabled.
490  *
500  PIA      CLR      CRA          adresseer DDRA
510          CLR      CRB          en DDRB
520          LDA      #$80         stel PA7 in als output en
530          STA      DDRA         PA0 t/m PA6 als input
540          CLR      DDRB         stel PB0 t/m PB7 in als input
550          LDA      #%00000111   adresseer PRA, enable IRQA1 en
560          STA      CRA          maak de opgaande flank van
570  *                               CA1 actief

```

```

580      LDA    #%00000100    adresseer PRB en
590      STA    CRB          disable IRQB
600      TST    PRA          reset de interrupt
610      TST    PRB          request bits
620      RTS
630 *
640 *****
650 * ISR voor de communicatie met ADC *
660 *****
670 *
680 *De informatie van PRA en PRB wordt opgeslagen in het
690 *geheugen op RESULT ($8F00 en $8F01), waarbij bit 15 (data
700 *valid flag) wordt geset.
710 *
720 ISRADC TST    CRA
730      BMI    CA1INT        spring als IRQA1=1 (dus bij
740 *                      neergaande flank op CA1)
750      TST    PRA          reset de interrupt
760      TST    PRB          request bits
770      BRA    ENDISR
780 CA1INT LDD    PRA          lees meetwaarde en reset
790 *                      interrupt request bits
800      ORA    #$80          set data valid flag
810      STD    RESULT        berg meetwaarde en flag op
820 ENDISR RTI
830      END    PROG

```

In regel 340 wordt getest of de meetwaarde al in het geheugen is gezet. Bit 15 is de data valid flag. Als deze 0 is, dus als (D) positief is, is de meetwaarde er nog niet en moet er gewacht worden.

Is bit 15 geset, dan kan het programma worden voortgezet.

De data wordt ongeldig verklaard (in regel 350), opdat hij bij een nieuwe EOC-interrupt weer geldig kan worden verklaard.

In regel 360 worden de vier niet ter zake doende bits gereset.

In dit voorbeeld is EOC de enige interrupt die kan voorkomen.

Toch wordt in regel 710 getest of de interrupt van CA1 (dus van EOC) afkomstig is. Dit is zeer aan te bevelen om te voorkomen

dat er ten onrechte een ongeldige meetwaarde wordt gelezen, bijvoorbeeld als gevolg van een storing op een van de andere interruptingangen. Bij een valse interrupt moet in ieder geval weer worden teruggekeerd naar het hoofdprogramma, nadat de interrupt request bits zijn gereset. Zou door een valse interrupt IRQB1 of IRQB2 worden geset, dan zou er telkens opnieuw naar de interrupt service routine worden gesprongen. Het programma zou hierdoor vastlopen.

Bedenk in regel 780 dat D wordt geladen met de informatie van PRA en PRB.

In regel 600 en 610 worden alle PIA-interrupts gereset. De kans bestaat dat een of meer interrupt request bits geset zijn door een storing op een van de interruptingangen van de PIA. De interrupt request bits moeten worden gereset om te voorkomen

dat de microprocessor telkens opnieuw naar de interrupt service routine (ISRADC) springt en zodoende niet meer verder kan gaan met het uitvoeren van het programma.

Het missen van PIA-interrupts

Om te voorkomen dat er interrupts gemist worden, moet worden vermeden dat een aanvrager zijn interruptaanvraag te vlug inactief maakt. Bij gebruik van een PIA zal dit niet gauw een probleem zijn. De interrupt is actief op een flank en wordt opgeslagen in een flipflop (van CRA of CRB). Er kan geen interrupt verloren gaan ten gevolge van het blokkeren van de interrupts in de processor. Vroeg of laat zal de interrupt toch worden gedetecteerd.

Er kunnen echter twee andere problemen optreden met betrekking tot interrupts:

1. Interrupts op dezelfde PIA-ingang komen te snel na elkaar. Indien de vorige interrupt nog niet is gedetecteerd en gereset door de processor, zal de volgende interrupt niet als een aparte interrupt gezien worden. De interrupt service routine wordt in dit geval slechts één keer uitgevoerd.

Remedie:

Meld de interruptafhandeling aan de aanvrager, zodat deze weet wanneer hij een volgende interruptaanvraag mag doen.

2. Een interrupt wordt gereset vlak voordat hij actief wordt.

a. Indien een interruptingang actief wordt tijdens het uitvoeren van een instructie waarbij het peripheral register PRA, respectievelijk PRB wordt gelezen, dan wordt de interrupt request bit meteen gereset, zodat de bijbehorende interrupt verloren gaat.

Remedie:

Lees een peripheral register alleen als het bijbehorende interrupt request bit is geset.

b. Een andere mogelijkheid om een interrupt te missen wordt veroorzaakt doordat twee verschillende interrupt request bits (namelijk IRQA1 en IRQA2, respectievelijk IRQB1 en IRQB2) tegelijkertijd worden gereset.

Stel dat CA1 en CA2 als interruptingangen worden gebruikt. Bij een interrupt van CA1 wordt er gesprongen naar de interrupt service routine:

```
100 MISINT LDA    CRA
110          LDB    PRA
120          --
130          --
140          RTI
```

In regel 100 wordt CRA gelezen teneinde de waarden van IRQA1 en IRQA2 binnen te halen. IRQA1=1 vanwege de interrupt van CA1. IRQA2=0, omdat er nog geen CA2-interrupt is geweest. In accumulator A is bit 7 dus geset en bit 6 gereset. Als er tijdens het ophalen van de instructie LDB PRA een CA2-interrupt optreedt, zal deze interrupt verloren gaan. Weliswaar zal IRQA2 geset worden, maar bij het uitvoeren van de instructie LDB PRA

worden IRQA1 en IRQA2 beiden gereset.

Wordt er nu gekeken welke interrupt er is geweest, dan wordt alleen IRQA1 gedetecteerd, immers in accumulator A is bit 7 geset en bit 6 gereset.

Naarmate er meer tijd verstrijkt tussen de instructies LDA CRA en LDB CRB (ten gevolge van tussengevoegde instructies), is de kans groter om een interrupt te missen.

Vandaar de aanbeveling om de interrupt request bits zo snel mogelijk te resetten.

Remedie:

Dit probleem is alleen te vermijden door slechts één CA-interrupt en één CB-interrupt toe te laten, tenzij men er zeker van is dat de beide CA-, respectievelijk CB-interrupts niet ongeveer gelijktijdig kunnen optreden. Komt CA2 nadat de instructie LDB PRA is uitgevoerd, dan wordt IRQA2 geset, nadat de IRQA-bits gereset zijn. Bit 6 van accumulator A is echter 0, zodat alleen de CA1-interrupt wordt afgehandeld.

Wordt er teruggekeerd naar het hoofdprogramma, dan veroorzaakt het geset zijn van IRQA2 echter een onmiddellijke terugkeer naar de interrupt service routine, zodat de CA2-interrupt in dit geval alsnog wordt afgehandeld.

10.8 Strobing met de PIA

Output modes

De signalen CA2 en CB2 van de PIA zijn behalve als interruptingangen ook als uitgangen te programmeren. In het laatste geval moet bit 5 van het control register CRA, respectievelijk CRB geset zijn. Vervolgens kan in de output mode gekozen worden uit drie functies voor de signalen CA2 en CB2:

1. strobe-signaal
2. handshake-signaal
3. following bit CRA3, respectievelijk bit CRB3

In elke output mode is CRA6 (IRQA2), respectievelijk CRB6 (IRQB2) gelijk aan 0.

5 4 3

Strobe mode

CRA, CRB

			1	0	1		
--	--	--	---	---	---	--	--

Om CA2 en CB2 als strobe-signalen te gebruiken, moet bit 4 van het overeenkomstige control register 0 zijn, terwijl bit 3 gelijk moet zijn aan 1.

Voor het eerst zullen nu verschillen optreden tussen het A-deel en het B-deel van de PIA. Het A-deel zal ingericht blijken te zijn op het binnenhalen van data en het B-deel op het verzenden van data.

In de rusttoestand zijn de uitgangen CA2 en CB2 hoog (1-niveau).

Ontvangen via poort PA

CA2 gaat in de strobe mode naar 0 op de eerste neergaande flank van de klokpuls E, volgend op het lezen van het Peripheral Register PRA door de microprocessor. Op de volgende neergaande flank van E keert CA2 terug op het 1-niveau. Hieruit blijkt dat het A-deel van de PIA is ingesteld voor het binnenhalen van data (zie fig. 10.34). Het lezen van de gegevens uit het Peripheral Register PRA heeft als gevolg dat er automatisch een puls wordt gegenereerd op de uitgang CA2 van de PIA. CA2 is derhalve op te vatten als het signaal: "verzoek om data".

Er is geen signaal dat aangeeft dat de data, die door het input device verstuurd wordt, geldig is. Is de response tijd (antwoordtijd) van het device bekend, dan kan deze tijd met een wachtlus of met een timer worden afgeteld, waarna de data door de processor kan worden binnengehaald door te lezen in PRA.

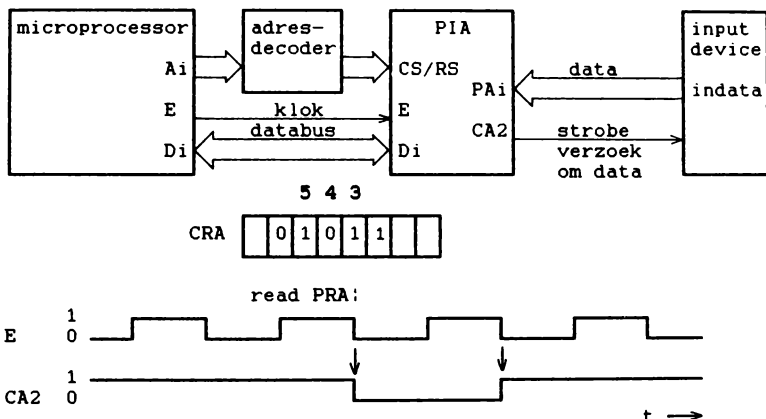


Fig. 10.34 Strobing met de PIA bij data-invoer

Zenden via poort PB

CB2 gaat in de strobe mode naar 0 op de eerstvolgende opgaande flank van E na het schrijven in het Peripheral Register PRB (door de microprocessor). Op de daaropvolgende opgaande flank van E wordt CB2 weer 1. Het B-deel van de PIA is ingesteld voor het verzenden van data. Na het schrijven in PRB wordt er op CB2 automatisch een puls gegenereerd (zie fig. 10.35). Deze puls is derhalve op te vatten als het signaal, dat aangeeft dat de data geldig is, dus als het signaal "hier is data".

Merk op dat er tussen het schrijven in PRB en de puls op CB2 een halve klokperiode verstrijkt. Dit is gedaan om te waarborgen dat deze puls niet bij het peripheral device aankomt, voordat de data daar aangekomen is. Bedenk hierbij dat

een instructiecyclus eindigt bij de neergaande flank van E. Looptijdverschillen tussen strobe en data zijn hierdoor niet van belang, mits ze niet groter zijn dan een halve klokpulsperiode van E.

Haalt het output device de data binnen op de opgaande flank van CB2, dan mogen de looptijdverschillen anderhalve klokpulsperiode bedragen.

Bij nog grotere looptijdverschillen moet de strobe (bijvoorbeeld met behulp van externe elektronica) worden vertraagd.

Ook hier ontbreekt de terugmelding van het peripheral device. Het output device geeft niet te kennen wanneer het de data overgenomen heeft. Is de verwerkingstijd van het device bekend, dan kan deze tijd afgeteld worden.

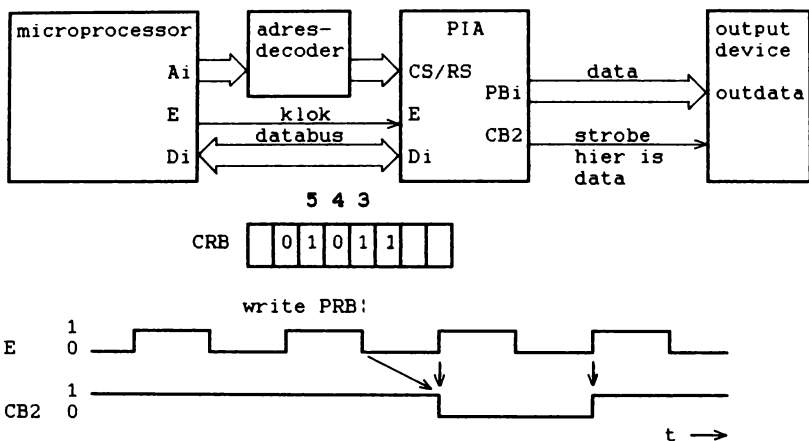


Fig. 10.35 Strobing met de PIA bij data-uitvoer

10.9 Handshaking met de PIA

Pulsed handshake in de strobe mode CRA, CRB

			1	0	1		
--	--	--	---	---	---	--	--

De zogenaamde pulsed handshake (zie paragraaf 10.5) kan met behulp van PIA's worden gerealiseerd door de PIA's aan weerszijden van het communicatiekanaal in te stellen in de strobe mode. Om de besturingssignalen niet te missen, worden deze ontvangen op de interruptingangen CA1, respectievelijk CB1. Hierdoor worden de interrupt request bits IRQA1 en IRQB1 geset bij een actieve flank op CA1, respectievelijk CB1.

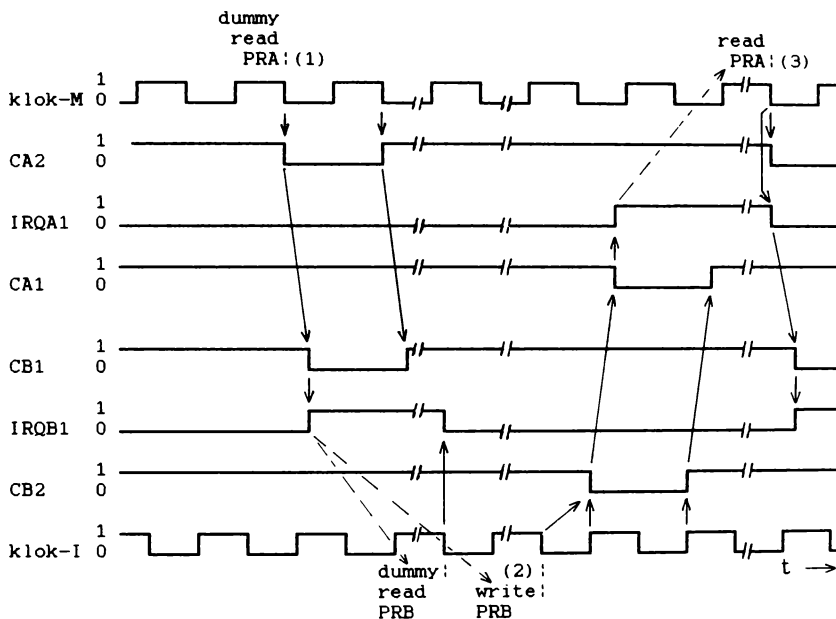
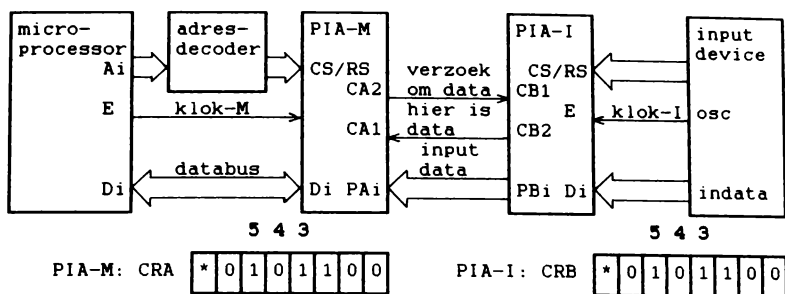


Fig. 10.36 Pulsed handshake voor data-invoer met PIA's in strobe mode

Fig. 10.36 laat zien hoe de invoer van data op deze wijze kan geschieden.

1. De processor start de communicatie door een verzoek om data te doen middels het signaal "verzoek om data". Dit gebeurt door een puls te genereren op $CA2$ van PIA-M, de PIA aan de microprocessorzijde. Hiertoe leest de processor het register PRA, zonder verder iets met deze informatie te doen. Men spreekt hier wel van een dummy read. Door te lezen in PRA worden tevens de bits $IRQA1$ en $IRQA2$ gereset.

2. Het signaal "verzoek om data" wordt ontvangen op de CB1-ingang van PIA-I, de PIA aan de zijde van de invoereenheid. Hierdoor wordt in PIA-I de bit IRQB1 van het control register geset. Dit zal op een gegeven moment ontdekt worden door de invoereenheid. Deze reageert hierop met een dummy read in PRB teneinde de bit IRQB1 weer te resetten. Vervolgens schrijft de invoereenheid data in het peripheral register PRB van PIA-I. Hierdoor ontstaat er een puls op de CB2-uitgang.
3. Deze puls wordt ontvangen op de CA1-ingang van PIA-M, zodat hier de bit IRQA1 wordt geset. De microprocessor zal dit detecteren en hierna de data binnenhalen door te lezen in PRA, waardoor tevens IRQA1 wordt gereset en een nieuwe puls "verzoek om data" wordt verzonden.
4. Zie punt 2.

Wil de microprocessor geen nieuwe data meer ontvangen, dan moet PIA-M voor het lezen van PRA zodanig worden ingesteld, dat er geen puls meer verschijnt op CA2, bijvoorbeeld door CA2 als input te programmeren. Om de verbinding CA2-CB1 hoog (= 1) te houden, moet deze lijn via een pull-up weerstand met de voedingsspanning zijn verbonden.

Omdat de invoereenheid de data verzendt via PRB van PIA-I en de processor de data binnenhaalt via PRA van PIA-M, moet poort PB van PIA-I verbonden zijn met poort PA van PIA-M.

Realiseer u dat klok-M van de microprocessor en klok-I van de invoereenheid niet synchroon lopen!

De verschuiving tussen de signalen CA2 en CB1, respectievelijk tussen CB2 en CA1, geeft de looptijd van deze signalen in de verbinding tussen de PIA's weer. Deze looptijd bedraagt ongeveer 5 ns/m, hetgeen voor een kabel van 20 meter neerkomt op ongeveer 100 ns.

De propagatietijden van de PIA's worden in de figuren niet weergegeven.

In het voorbeeld van fig. 10.36 zijn beide PIA's geprogrammeerd voor programmed I/O. De interrupt request bits IRQA1, respectievelijk IRQB1 worden niet doorgegeven aan de interrupt request uitgangen. De interrupt request enable bits CRA0 en CRB0 mogen hier geset zijn, zodat de handshake-signalen wel als interrupts worden doorgegeven.

Daar CRA1 en CRB1 zijn gereset, zijn de ingangen CA1 en CB1 actief op de neergaande flank. Dit mag ook de opgaande flank zijn, echter dan treedt er een extra vertraging op in de afhandeling van de handshake. Deze vertraging bedraagt twee keer een halve klokcyclus.

Natuurlijk kan de handshake ook op basis van interrupt-I/O worden afgehandeld. Hiertoe moet CRA0, respectievelijk CRB0 geset zijn.

Bij de communicatie met een AD-converter zijn SOC (= CA2) en EOC (= CB2) de handshake-signalen.

Handshake mode

CRA, CRB

				5	4	3				
				1	0	0				

De PIA kent ook een handshake mode. De signalen CA2 en CB2 functioneren hierbij als handshake-signalen en moeten als output geprogrammeerd worden. Dit gebeurt door de bits CRA5, respectievelijk CRB5, te setten. Verder moeten de bits 3 en 4 van de control registers CRA, respectievelijk CRB, gereset worden. Hiermee wordt de handshake mode ingesteld. Net als in de strobe mode is ook in de handshake mode het A-deel van de PIA ingesteld op ontvangen en het B-deel op zenden.

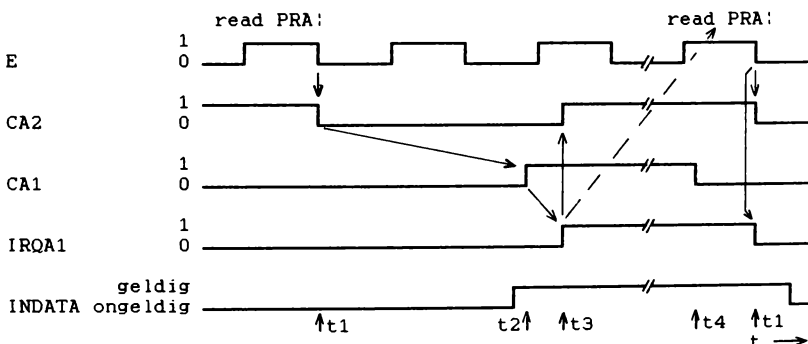
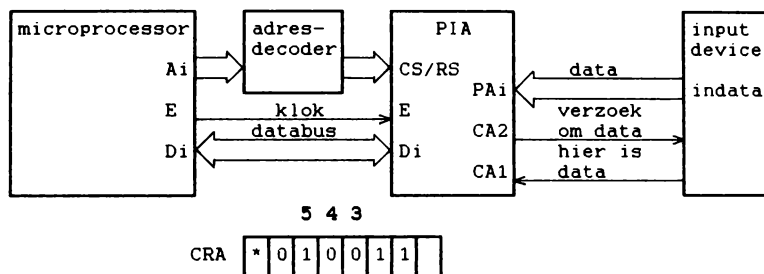


Fig. 10.37 Handshaking met de PIA bij data-invoer

Ontvangen via poort PA in de handshake mode

De werking van de handshake-signalen CA2 en CB2 is als volgt. CA2 en CB2 zijn hoog (= 1) in de rusttoestand. CA2 wordt laag (= 0) op de eerstvolgende neergaande flank van het kloksignaal E na het lezen in het peripheral register PRA. Hieruit blijkt dat het A-deel is ingesteld op ontvangen. CA2 wordt pas gelijk aan 1 bij de volgende actieve flank op de

interruptingang CA1 (zie fig. 10.37). Dit wijkt af van de strobe mode, waar CA2 slechts één klokcyclus laag blijft. De actieve flank van CA1 is bepaald door de waarde van bit CRA1 (zie paragraaf 10.7).

De betekenis van de overgangen van de handshake-signalen is als volgt:

t1: verzoek om data; de eventuele vorige data is overgenomen

t2: hier is data

t3: signaal "hier is data" is ontvangen; data is nog niet overgenomen!

t4: antwoordsignaal CA1 gaat terug naar de rusttoestand

Merk op dat het moment van optreden van de tweede flank van CA1 eigenlijk niet van belang is. De PIA gebruikt immers alleen de flank van CA1, waarop IRQA1 wordt geset. De andere flank moet natuurlijk wel optreden, omdat er anders geen actieve flank meer mogelijk is voor de volgende handshake. Er kunnen immers geen twee flanken met dezelfde richting direct achter elkaar voorkomen. De achterflank van CA1 mag eventueel pas komen na het lezen van de data in PRA.

Wel moet de data geldig blijven tot na het lezen van PRA.

Hiermee wordt tegelijkertijd medegedeeld dat de data is overgenomen en dat er opnieuw data kan worden ontvangen. Dit houdt in dat aan het einde van een handshake-cyclus onvermijdelijk een nieuwe handshake-cyclus wordt gestart. Dit is geen ideale situatie.

Om de data-invoer te stoppen, moet de processor een aparte boodschap naar de invoereenheid sturen om aan te geven dat de communicatie moet worden beëindigd. Hiervoor is een extra verbinding nodig tussen processor (of PIA) en invoereenheid. De invoer kan ook worden beëindigd door niet meer in PRA te lezen. Er wordt dan geen nieuwe handshake-cyclus meer gestart. De vorige cyclus is echter nog niet geheel afgelopen. De invoereenheid wacht nog steeds op het antwoordsignaal, dat de data is overgenomen. De invoereenheid moet de handshake nu beëindigen met behulp van time-out.

Zenden via poort PB in de handshake mode

CB2 gaat in de handshake mode automatisch naar 0 op de eerste opgaande flank van het kloksignaal E, die volgt op het schrijven in PRB. Dit impliceert dat het B-deel van de PIA is ingesteld op zenden. CB2 gaat terug naar 0 bij de volgende actieve flank op de interruptingang CB1. Bit CRB1 bepaalt welke flank de actieve flank is op CB1 (zie paragraaf 10.7). Fig. 10.38 laat de handshake via de PIA zien voor de uitvoer van data.

De overgangen van de handshake-signalen kunnen als volgt worden opgevat:

t1: hier is data

t2: data overgenomen

t3: data wordt ongeldig verklaard

t4: antwoordsignaal CB1 gaat terug naar de rusttoestand

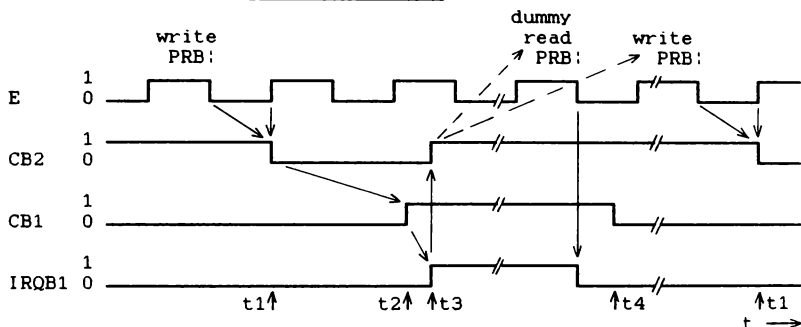
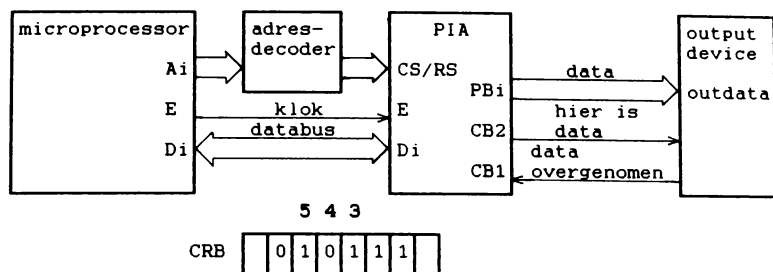


Fig. 10.38 Handshaking met de PIA bij data-uitvoer

De achterflank van CB1 wordt door de PIA niet gebruikt. Dit betekent dat deze flank eventueel ook na het schrijven in PRB mag optreden. De data blijft geldig tot na de volgende actieve flank van CB1. Deze kan alleen optreden als CB1 is teruggekeerd naar de rusttoestand.

Het zenden kan eenvoudig worden gestopt door niet meer in PRB te schrijven. De vorige handshake-cyclus is reeds helemaal uitgevoerd.

Let op dat eerst de bit IRQB1 wordt gereset alvorens een nieuwe handshake-cyclus te starten! Gebeurt dit niet, dan bestaat het gevaar dat het antwoordsignaal (in deze fase de actieve flank van CB1) al binnenkomt, terwijl IRQB1 nog geset is. In dit geval ontstaat er geen neergaande flank op CB2 bij het schrijven in PRB, zodat het signaal "hier is data" achterwege blijft.

Soort pulsed handshake in de handshake mode

Fig. 10.39 laat zien hoe een handshake plaatsvindt met behulp van twee PIA's. Hierbij haalt de microprocessor data binnen vanaf een invoereenheid.

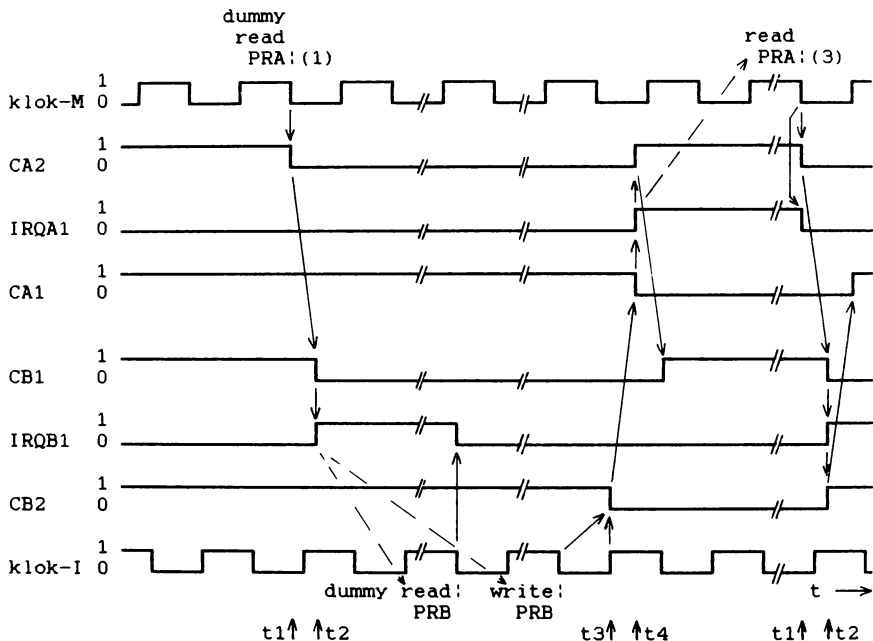
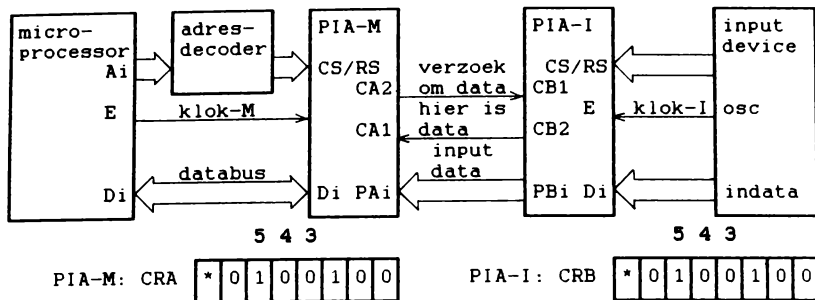


Fig. 10.39 Data-invoer met behulp van PIA's in de handshake mode

Bij het begin moeten de interrupt request bits IRQA1 en IRQB1 gereset zijn.

1. De handshake-procedure wordt door de microprocessor gestart met behulp van een dummy read in PRA. Hierdoor wordt CA2 gelijk aan 0, zodat er bij PIA-I een interrupt ontvangen wordt.
2. Hier wordt bit IRQB1 geset, evenals CB2, indien deze 0 was. Deze interrupt wordt op een gegeven moment ontdekt (doordat het input device CRB7 test). Het input device reset IRQB1 met behulp van een dummy read.

3. Daarna zet het input device data in PRB van PIA-I. Deze data wordt ontvangen in PRA van PIA-M. Door het schrijven in PRB wordt CB2 gelijk aan 0, zodat er een interrupt binnen komt op CA1 van PIA-M.
4. Hierdoor wordt IRQA1 geset en worden CA2 en dus ook CB1 weer hoog. De interrupt zal middels polling door de microprocessor worden ontdekt, waarna deze de data leest in PRA.
Hierdoor worden IRQA1 en CA2 gereset (zie punt 1). De neergaande flank op CA2 veroorzaakt wederom een interrupt op CB1 van PIA-I, enzovoorts.

De communicatie kan worden beëindigd door een speciaal karakter (bijvoorbeeld ETX, End of Text) te laten zenden door de invoereenheid. De microprocessor weet dan dat alle gegevens binnen zijn en de invoereenheid reageert op de laatste interrupt alleen nog met een dummy read in PRB, teneinde de interrupt request bit IRQB1 te resetten. Hiermee is de rusttoestand bereikt.

U ziet dat de synchronisatie van de signalen alleen niet voldoende is voor een goede communicatie. Er is ook een communicatieprotocol nodig om bijvoorbeeld het begin en het einde van een bericht aan te geven. De vele soorten communicatieprotocollen vallen echter buiten het bestek van dit boek.

De interrupt request bits mogen natuurlijk ge-enabled zijn om de handshake op basis van interrupt-I/O te kunnen afhandelen. De interruptingangen moeten in ieder geval actief zijn op de neergaande flank. Zou de interrupt actief zijn op de opgaande flank, dan kan het schrijven in PRB, respectievelijk het lezen in PRA, door de andere PIA niet ontdekt worden en kan er geen uitwisseling van data plaatsvinden. Bij write PRB en read PRA ontstaan er negatieve flanken op CB2 en CA2, zodat de actieve interruptflanken ook negatief moeten zijn.

Het uitvoeren van data van een microprocessor naar een uitvoerapparaat kan eveneens middels handshaking via PIA's geschieden (zie fig. 10.40).

IRQA1 en IRQB1 moeten gereset zijn.

1. De microprocessor kan nu beginnen met data te verzenden door te schrijven in PRB van PIA-M. Deze data wordt ontvangen in PRA van PIA-O (aan de kant van het output device). Hierdoor wordt CB2 gelijk aan 0, zodat er een interrupt binnenkomt op CA1 van PIA-O.
2. Deze interrupt set IRQA1 van PIA-O en eventueel ook het signaal CA2.
3. Als het output device deze interrupt waarneemt, kan de data gelezen worden in PRA. Hierbij wordt IRQA1 gereset en gaat tevens CA2 naar 0.

Veelal is het nodig dat het output device weet wanneer het datatransport eindigt. Dit moet aan het device worden meegedeeld met behulp van een aparte boodschap, bijvoorbeeld door het karakter EOT (End Of Transmission, zie appendix A) naar het output device te sturen.

Een input/output device kan via PIA's worden verbonden met een microprocessor. Er kan nu tegelijkertijd invoer en uitvoer van data plaatsvinden met behulp van handshake. Hiertoe worden aan weerszijden van de verbinding de beide PIA-helften gebruikt. We krijgen het volledige schema van deze zogenaamde full duplex verbinding door de figuren 10.39 en 10.40 te combineren (zie fig. 10.41).

Communicatie kan plaatsvinden in één of in twee richtingen:

- simplex: datatransport is slechts in één richting mogelijk
- half duplex: datatransport is in twee richtingen mogelijk, maar niet tegelijkertijd.
- full duplex: datatransport kan tegelijkertijd in twee richtingen plaatsvinden

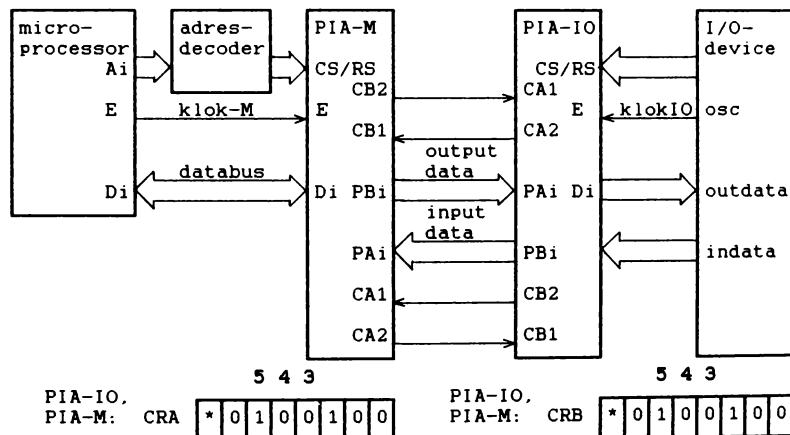


Fig. 10.41 Full duplex verbinding met PIA's in handshake mode

Bekijkt u de figuren 10.39 en 10.40 aandachtig, dan zult u zien dat het hier gaat om een soort pulsed handshake.

De (1-)pulsen op de handshake-lijnen overlappen elkaar niet. Op een neergaande flank van het ontvangen handshake-signaal wordt de interrupt request bit geset en tevens wordt het uitgaande handshake-signaal gelijk aan 1. Er gebeurt hierna niets meer totdat er wordt gelezen in PRA, respectievelijk geschreven in PRB (als reactie op de interrupt request). Hierdoor wordt het

uitgaande handshake-sigitaal gelijk aan 0.

Dit veroorzaakt aan de andere kant van de verbinding het zetten van de interrupt request bit en het 1 maken van het handshake-sigitaal. Wederom gebeurt er niets, totdat ook hier wordt geschreven in PRB, respectievelijk gelezen in PRA (als reactie op de interrupt request). Dan wordt het betreffende handshake-sigitaal weer 0.

Dit betekent dat de uitgaande handshake-lijn van een device achtereenvolgens wordt geset en gereset. Daarna gebeurt hetzelfde bij het andere device. Bij een overlappende handshake veranderen de uitgaande handshake-signalen van beide devices om beurten.

Bij een "gewone" pulsed handshake hebben de pulsen op de handshake-lijnen meestal een vaste duur (van bijvoorbeeld één klokcyclus). Deze pulsduur is in ieder geval onafhankelijk van het antwoordsigitaal. Bij de handshake van de PIA in de handshake mode is de pulsbreedte afhankelijk van het antwoordsigitaal. Een actieve (negatieve) flank op de binnenkomende handshake-lijn wordt onmiddellijk teruggemeld door de uitgaande handshake-lijn te deactiveren (1 te maken). Het binnenkomende actieve handshake-sigitaal wordt als het ware ge-echoed. Pas later, als de ontvanger op de actieve flank reageert door in PRA te lezen, respectievelijk door in PRB te schrijven, wordt de uitgaande handshake-lijn geactiveerd (0 gemaakt).

Overigens kan er met behulp van PIA's ook een "klassieke" handshake worden gerealiseerd met dezelfde schakeling als afgebeeld is in fig. 10.38. U moet de PIA's dan echter niet in de handshake mode instellen, maar in de following mode. In deze mode kunnen de "handshake-signalen" worden geprogrammeerd. Ze worden dan niet meer automatisch bediend door lees-en-schrijfoperaties in PRA en PRB en door actieve flanken op de interruptingangen. Ze moeten nu onder besturing van het programma worden geset en gereset.

10.10 De PIA in following mode

5 4 3

PIA instellen

CRA, CRB

		1	1	*			
--	--	---	---	---	--	--	--

De following mode van de PIA werkt als volgt.

De signalen CA2 en CB2 worden weer als uitgang geprogrammeerd. Hiertoe moeten CRA5 en CRB5 gelijk aan 1 zijn. Beide helften van de PIA zijn ingesteld in de following mode als verder geldt dat CRA4 en CRB4 geset zijn. De uitgang CA2 neemt nu telkens de waarde van CRA3 aan en CB2 krijgt steeds de waarde van CRB3. De uitgangen volgen dus de waarden van de bits 3 van de bijbehorende control registers.

Voorbeeld 10.9 (Interlocked handshake in de following mode)
 Fig. 42 laat zien hoe een interlocked handshake kan worden
 gerealiseerd in de following mode. In dit voorbeeld worden 10
 bytes binnengehaald in de microprocessor.

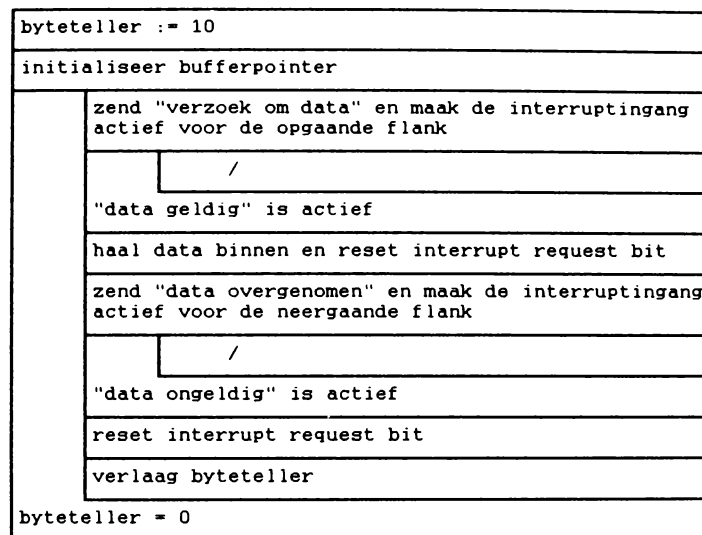


Fig. 10.42 Structogram van routine voor interlocked handshake met PIA-M in de following mode

De bijbehorende subroutine ziet er als volgt uit:

100	HANDIN	LDA	#10	initialiseer
110		STA	TELLER	de byteteller
120		LDX	#BUFFER	initialiseer de bufferpointer
130	SHAKE	LDA	CRA	zend "verzoek om data" en
140		ORA	##00001010	zorg dat CA1 actief is
150		STA	CRA	op de opgaande flank
160	GELDIG	LDA	CRA	wacht op "data geldig"
170		BPL	GELDIG	
180		LDB	PRA	reset IRQA1 en lees data
190		STB	,X+	berg data op en verhoog pointer
200		ANDA	##11110101	zend "data overgenomen"; activeer
210		STA	CRA	neergaande flank op CA1
220	RUST	LDA	CRA	wacht op "data ongeldig"
230		BPL	RUST	
240		TST	PRA	reset IRQA1
250		DEC	TELLER	
260		BNE	SHAKE	spring als geen 10 bytes gelezen
270		RTS		

Beide flanken van het antwoordsignaal CA1 moeten worden gedetecteerd. Dit betekent dat na het detecteren van de ene flank, de andere flank actief moet worden gemaakt, opdat hierop de bit IRQA1 wordt geset.
In regel 240 staat een dummy read in PRA om IRQA1 te resetten.

Fig. 10.43 toont het tijddiagram van de bijbehorende handshake.

Hierin is:

t1: verzoek om data

t2: hier is data

t3: data overgenomen

t4: rusttoestand; data wordt ongeldig

Daarna kan een nieuwe handshake-cyclus beginnen.

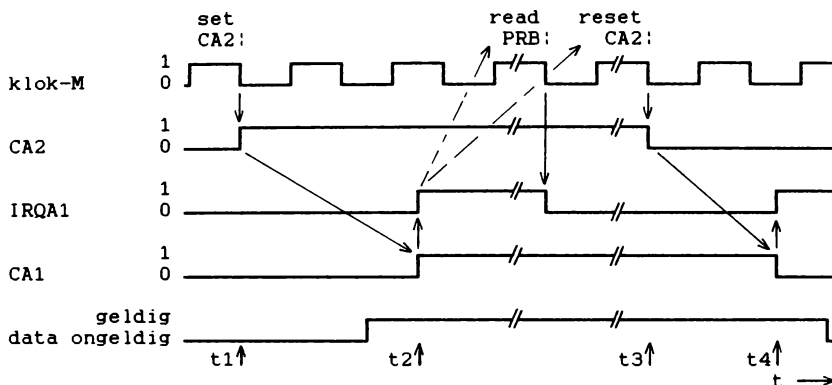


Fig. 10.43 Interlocked handshake voor data-invoer met behulp van PIA's in de following mode

PIA-M kan worden geïnitieerd met de subroutine IPIAM.

10	IPIAM	CLR	CRA	adresseer DDRA
20		CLR	CRB	en DDRB
30		CLR	DDRA	PRA wordt input
40		LDA	#\$FF	PRB wordt
50		STA	DDRB	output
60		LDA	00110110	PIA in following mode; CA2 en
70		STA	CRA	CB2 zijn 0; adresseer PRA en
80		STA	CRB	PRB; disable IRQA1 en IRQB1;
85				CA1 en CB1 opgaand actief
90		RTS		

Voor de PIA aan de zijde van het I/O-device kan dezelfde initialisatie routine worden gebruikt.

het omgekeerde van de duur van een elektrisch hoogniveau (of laagniveau). Het gaat bij de bit rate om "elektrische" bits. Bij de transmissie, die gebruik maakt van start- en stopbits, is de baud rate gelijk aan de bit rate. Bij de zogenaamde biphase modulatie van fig. 10.11 is de bit rate twee keer zo groot als de baud rate.

10.11.5 V24-signalen van de ACIA

De signalen $\overline{\text{RTS}}$, $\overline{\text{CTS}}$ en $\overline{\text{DCD}}$ worden doorgaans gebruikt bij de communicatie met modems (MODulator/DEModulators). Aangezien dit buiten het bestek van dit boek valt, zal slechts een summiere beschrijving van deze signalen volgen.

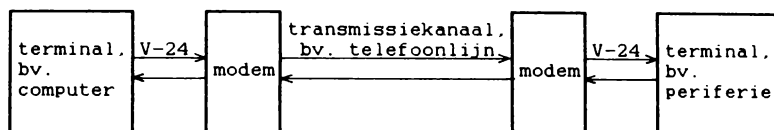


Fig. 10.48 Communicatie via modems

De uitgang $\overline{\text{RTS}}$, Request to Send, wordt 0 gemaakt om aan te geven dat de terminal (in dit geval de computer) wil gaan zenden over de communicatieverbinding. Deze ACIA-uitgang kan ook worden toegepast als $\overline{\text{DTR}}$ -signaal, Data Terminal Ready. $\overline{\text{DTR}}=0$ geeft aan dat de terminal gereed is om te communiceren met een andere terminal (zie fig. 10.48).

Het ingangssignaal $\overline{\text{CTS}}$, Clear to Send, van de ACIA wordt ook wel Ready for Sending (RFS) genoemd. Een 0 op deze ingang geeft aan dat de modem gereed is voor het verzenden van data. $\overline{\text{RFS}}$ is het antwoordsignaal van de modem op het signaal $\overline{\text{RTS}}$. De signalen $\overline{\text{RTS}}$ en $\overline{\text{CTS}}$ kunnen als handshake signalen worden gebruikt. In een V24-interface (zie fig. 10.48) zijn ze hiervoor eigenlijk niet bedoeld.

$\overline{\text{CTS}}=1$ blokkeert de Transmit Data Register Empty bit van de ACIA. Dit betekent dat $\overline{\text{TDRE}}=0$ is, zolang $\overline{\text{CTS}}=1$. Wordt het ingangssignaal $\overline{\text{CTS}}$ niet gebruikt, dan moet deze ingang 0 gemaakt worden. Dit is logisch omdat het geen zin heeft data te willen verzenden, als de modem hiervoor niet gereed is.

Het 0 zijn van de ingang $\overline{\text{DCD}}$, Data Carrier Detected, betekent dat de modem een draaggolf ontvangt (vanaf de andere kant van de lijn). Er kan nu data verwacht worden. Als de $\overline{\text{DCD}}$ -ingang 1 is, wordt de ontvanger in de ACIA geblokkeerd en geïnitieerd. Indien er geen draaggolf is, kan er geen zinvolle informatie worden ontvangen. Wordt het ingangssignaal $\overline{\text{DCD}}$ niet gebruikt, dan moet deze ingang 0 zijn.

De signalen Tx $\overline{\text{D}}$, Rx $\overline{\text{D}}$, $\overline{\text{RTS}}$, $\overline{\text{DTR}}$, $\overline{\text{RFS}}$ en $\overline{\text{DCD}}$ maken deel uit van

de interface tussen terminal en modem, zoals is vastgelegd in de V24-aanbeveling van de CCITT (Comité Consultatif International Telegraphique et Telephonique). Deze aanbeveling is vrijwel gelijk aan de Recommended Standard RS232-C van de EIA (Electronic Industries Association).

10.11.6 De registerselectie

U hebt gezien dat de communicatie via een ACIA verloopt middels de registers RDR en TDR. Het Status Register SR bevat informatie omtrent de toestand van de ACIA. Het Control Register CR bevat informatie waarmee de functie van de ACIA wordt ingesteld. Fig. 10.49 laat zien op welke wijze deze registers worden geselecteerd met behulp van de ingangen Register Select RS en Read/Write R/ $\bar{W}$.

RS	R/ $\bar{W}$	geselecteerde register
0	0	Control Register CR
0	1	Status Register SR
1	0	Transmit Data Register TDR
1	1	Receive Data Register RDR

Fig. 10.49 Selectie van de ACIA-registers

CR en TDR zijn write only registers. Zij zijn alleen te selecteren als R/ $\bar{W}$ =0, dus tijdens een schrijfoperatie. Dit betekent dat de informatie in deze registers niet teruggelezen kan worden. Is dit nodig, dan moet deze informatie op extra geheugenplaatsen worden bewaard.

SR en RDR zijn read only registers, want deze zijn alleen bereikbaar tijdens een leesoperatie.

CR en SR hebben hetzelfde adres, immers beiden worden geselecteerd als RS=0. Ook TDR en RDR hebben hetzelfde adres. Zij worden geadresseerd met RS=1.

10.11.7 Status register

De acht bits van het Status Register bevatten informatie omtrent de toestand van ACIA en datatransport.

Receive Data Register Full (RDRF), bit SR0, wordt geset als de ontvangen data vanuit het Receive Shift Register is overgebracht naar het Receive Data Register.

RDRF wordt gereset na het lezen in RDR en bij een master reset. De RDRF-bit blijft echter geset, indien er een overrun toestand is. De RDRF-bit wordt in dat geval pas gereset bij het opheffen van de overrun toestand (zie verderop in deze paragraaf).

RDRF is altijd 0 als het signaal DCD gelijk is aan 1, dus als de modem signaleert dat er geen draaggolf is.

Het verzenden van een bit via TxD gebeurt op de neergaande flank van de Transmit Clock TxCLK.

10.11.3 Ontvangen via de ACIA

Het ontvangen van data vanaf een peripheral device gaat als volgt:

1. Eerst wordt er gewacht totdat er een startbit wordt gedetecteerd. De data komt binnen op de serie-ingang RxD (Recieve Data). Om te kunnen synchroniseren moet de frequentie van de Receive Clock RxCLK groter zijn dan de frequentie van de binnenkomende data (zie paragraaf 10.5.3). Hiertoe kan RxCLK (intern) worden gedeeld door 16 of 64. Met de klok RxCLK wordt de binnenkomende data bemonsterd. De startbit wordt gedetecteerd bij het ontvangen van 8, respectievelijk 32 opeenvolgende nullen na de rusttoestand of na de stopbit(s).
2. Na het detecteren van een startbit worden de databits op elke 16de, respectievelijk 64ste opgaande flank van RxCLK in het Receive Shift Register geschoven, te beginnen met bit 0. De data wordt dus ingelezen op de (door 16 of 64) gedeelde klok RxCLK.
3. Nadat een (7- of 8-bits) karakter is ontvangen, wordt dit automatisch opgeslagen in het Receive Data Register RDR. Hierbij wordt de bit RDRF, Receive Data Register Full, geset. Dit is bit 0 van het Status Register SR. Eventueel wordt de pariteit gecontroleerd. Er wordt nagegaan of de een of twee stopbits in orde zijn.
4. Door te lezen in het register RDR wordt de bit RDRF gereset. Is het Receive Data Register nog niet gelezen als een volgend karakter is ontvangen in het Receive Shift Register, dan wordt dit karakter niet overgenomen door RDR. Dit karakter gaat verloren.

10.11.4 Communicatie via ACIA's

Asynchroon datatransport

Zoals in paragraaf 10.5.3 is uitgelegd, moet het kloksignaal aan de ontvangerzijde een hogere frequentie hebben dan het datasignaal, indien een binnenkomend signaal moet worden gesynchroniseerd. Vandaar dat de ontvangerklok RxCLK bij de ACIA intern kan worden gedeeld door 16 of door 64. In het laatste geval kan het midden van de bittijd nauwkeuriger worden opgespoord. Hierbij zijn er grotere verschillen in frequentie toegestaan tussen de zender- en de ontvangerklok.

Synchroon datatransport

De deelfactor kan echter ook 1 zijn. In dit geval kan er niet meer worden gesynchroniseerd. Dit betekent dat de klok synchroon moet lopen met de seriële data. Dit kan bijvoorbeeld

worden bereikt door de zenderklok via een aparte lijn met de data mee te sturen en deze aan de andere kant te gebruiken als ontvangerklok (zie fig. 10.46).

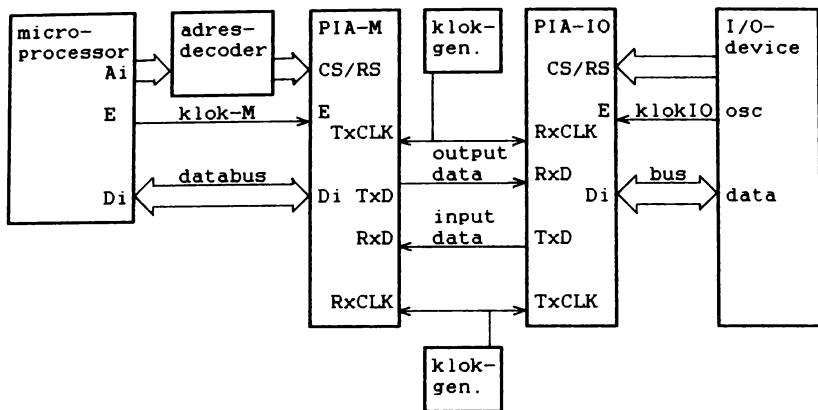


Fig. 10.46 Synchron serieel datatransport via ACIA's

Nu blijkt tevens het nut van het op de neergaande flank van de klok verzenden van data en het op de opgaande flank binnenschuiven van data. Het inlezen van data gebeurt dan ook bij synchrone transmissie midden in de bittijd. Hierdoor zijn looptijdverschillen tussen klok en data toegestaan, mits deze niet groter zijn dan de klokpulsbreedte (zie fig. 10.47).

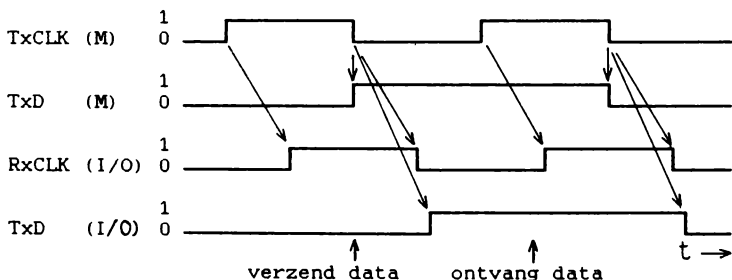


Fig. 10.47 Tijddiagram bij het synchron verzenden van data via ACIA's

Het synchron ontvangen van data gaat op soortgelijke wijze.

De snelheid waarmee wordt gecommuniceerd, wordt uitgedrukt in baud. De baud rate is het aantal gegevensbits dat per seconde wordt getransporteerd. Dit is $1/\text{bittijd}$. De baud rate is niet altijd gelijk aan de bit rate. Het aantal bits per seconde is

het omgekeerde van de duur van een elektrisch hoogniveau (of laagniveau). Het gaat bij de bit rate om "elektrische" bits. Bij de transmissie, die gebruik maakt van start- en stopbits, is de baud rate gelijk aan de bit rate. Bij de zogenaamde biphasemodulatie van fig. 10.11 is de bit rate twee keer zo groot als de baud rate.

10.11.5 V24-signalen van de ACIA

De signalen $\overline{\text{RTS}}$, $\overline{\text{CTS}}$ en $\overline{\text{DCD}}$ worden doorgaans gebruikt bij de communicatie met modems (MODulator/DEModulators). Aangezien dit buiten het bestek van dit boek valt, zal slechts een summier beschrijving van deze signalen volgen.

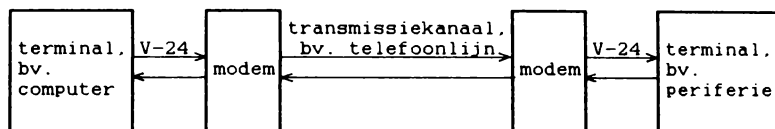


Fig. 10.48 Communicatie via modems

De uitgang $\overline{\text{RTS}}$, Request to Send, wordt 0 gemaakt om aan te geven dat de terminal (in dit geval de computer) wil gaan zenden over de communicatieverbinding. Deze ACIA-uitgang kan ook worden toegepast als $\overline{\text{DTR}}$ -signaal, Data Terminal Ready. $\overline{\text{DTR}}=0$ geeft aan dat de terminal gereed is om te communiceren met een andere terminal (zie fig. 10.48).

Het ingangssignaal $\overline{\text{CTS}}$, Clear to Send, van de ACIA wordt ook wel Ready for Sending (RFS) genoemd. Een 0 op deze ingang geeft aan dat de modem gereed is voor het verzenden van data. $\overline{\text{RFS}}$ is het antwoordsignaal van de modem op het signaal $\overline{\text{RTS}}$. De signalen $\overline{\text{RTS}}$ en $\overline{\text{CTS}}$ kunnen als handshake signalen worden gebruikt. In een V24-interface (zie fig. 10.48) zijn ze hiervoor eigenlijk niet bedoeld.

$\overline{\text{CTS}}=1$ blokkeert de Transmit Data Register Empty bit van de ACIA. Dit betekent dat $\overline{\text{TDRE}}=0$ is, zolang $\overline{\text{CTS}}=1$. Wordt het ingangssignaal $\overline{\text{CTS}}$ niet gebruikt, dan moet deze ingang 0 gemaakt worden. Dit is logisch omdat het geen zin heeft data te willen verzenden, als de modem hiervoor niet gereed is.

Het 0 zijn van de ingang $\overline{\text{DCD}}$, Data Carrier Detected, betekent dat de modem een draaggolf ontvangt (vanaf de andere kant van de lijn). Er kan nu data verwacht worden. Als de $\overline{\text{DCD}}$ -ingang 1 is, wordt de ontvanger in de ACIA geblokkeerd en geïnitieerd. Indien er geen draaggolf is, kan er geen zinvolle informatie worden ontvangen. Wordt het ingangssignaal $\overline{\text{DCD}}$ niet gebruikt, dan moet deze ingang 0 zijn.

De signalen TxD , RxT , $\overline{\text{RTS}}$, $\overline{\text{DTR}}$, $\overline{\text{RFS}}$ en $\overline{\text{DCD}}$ maken deel uit van

de interface tussen terminal en modem, zoals is vastgelegd in de V24-aanbeveling van de CCITT (Comité Consultatif International Telegraphique et Telephonique). Deze aanbeveling is vrijwel gelijk aan de Recommended Standard RS232-C van de EIA (Electronic Industries Association).

10.11.6 De registerselectie

U hebt gezien dat de communicatie via een ACIA verloopt middels de registers RDR en TDR. Het Status Register SR bevat informatie omtrent de toestand van de ACIA. Het Control Register CR bevat informatie waarmee de functie van de ACIA wordt ingesteld. Fig. 10.49 laat zien op welke wijze deze registers worden geselecteerd met behulp van de ingangen Register Select RS en Read/Write R/ $\bar{W}$.

RS	R/ $\bar{W}$	geselecteerde register
0	0	Control Register CR
0	1	Status Register SR
1	0	Transmit Data Register TDR
1	1	Receive Data Register RDR

Fig. 10.49 Selectie van de ACIA-registers

CR en TDR zijn write only registers. Zij zijn alleen te selecteren als R/ $\bar{W}$ =0, dus tijdens een schrijfoperatie. Dit betekent dat de informatie in deze registers niet teruggelezen kan worden. Is dit nodig, dan moet deze informatie op extra geheugenplaatsen worden bewaard.

SR en RDR zijn read only registers, want deze zijn alleen bereikbaar tijdens een leesoperatie.

CR en SR hebben hetzelfde adres, immers beiden worden geselecteerd als RS=0. Ook TDR en RDR hebben hetzelfde adres. Zij worden geadresseerd met RS=1.

10.11.7 Status register

De acht bits van het Status Register bevatten informatie omtrent de toestand van ACIA en datatransport.

Receive Data Register Full (RDRF), bit SR0, wordt geset als de ontvangen data vanuit het Receive Shift Register is overgebracht naar het Receive Data Register.

RDRF wordt gereset na het lezen in RDR en bij een master reset. De RDRF-bit blijft echter geset, indien er een overrun toestand is. De RDRF-bit wordt in dat geval pas gereset bij het opheffen van de overrun toestand (zie verderop in deze paragraaf).

RDRF is altijd 0 als het signaal DCD gelijk is aan 1, dus als de modem signaleert dat er geen draaggolf is.

Transmit Data Register Empty (TDRE), bit SR1, wordt geset als de inhoud van het Transmit Data Register is overgebracht naar het Transmit Shift Register.
TDRE=0 als $\overline{\text{CTS}}=1$, dus als de modem niet gereed is voor het verzenden van data.

Data Carrier Detect ($\overline{\text{DCD}}$), bit SR2 wordt geset bij een opgaande flank van het ingangssignaal $\overline{\text{DCD}}$. Dit treedt op als de draaggolf bij de modem is weggevallen. Het zetten van bit 2 veroorzaakt een interrupt request (de IRQ-bit, SR7, wordt geset), mits de Receiver Interrupt Enable bit (CR7) 1 is. Na het 0 worden van de $\overline{\text{DCD}}$ -ingang blijft de bit $\overline{\text{DCD}}$ gelijk aan 1 tot een master reset of tot het achtereenvolgens lezen van SR en RDR. Blijft de $\overline{\text{DCD}}$ -ingang hoog na een master reset of na het lezen van SR en RDR, dan wordt de interrupt gereset, terwijl de bit $\overline{\text{DCD}}$ de gelijknamige ingang volgt.

Clear-to-Send ($\overline{\text{CTS}}$), bit SR3, volgt het gelijknamige ingangssignaal.
Een master reset beïnvloedt de $\overline{\text{CTS}}$ -bit niet.

Framing Error (FE), bit SR4, geeft aan dat het dataframe niet correct is ontvangen. Dit wordt gedetecteerd door de afwezigheid van de eerste stopbit. De oorzaak hiervan is gelegen in een synchronisatiefout, een fout bij het overbrengen van data of een zogenaamde break conditie. Een break wordt gebruikt om de modem aan de andere kant van het transmissiekanaal te laten stoppen met het zenden van data. Hiertoe wordt de lijn TxD blijvend 0 gemaakt. De FE-flag wordt geset, dan wel gereset als de inhoud van het Receive Shift Register wordt overgebracht naar het Receive Data Register. Zodoende is deze foutindicatie aanwezig gedurende de gehele tijd dat het bijbehorende karakter beschikbaar is. FE wordt verder gereset bij een master reset.

Receiver Overrun (OVRN), bit SR5, geeft aan dat er een of meerdere karakters verloren zijn gegaan. Dit is het geval als er een of meerdere karakters zijn ontvangen, voordat het vorige karakter uit het Receive Data Register is gelezen. De overrun conditie begint in het midden van de laatste databit van het tweede karakter, dat na de laatste leesoperatie in RDR werd ontvangen. De OVRN-bit wordt pas geset nadat het geldige karakter uit RDR is gelezen. Tijdens de overrun conditie blijft de karaktersynchronisatie behouden.
De OVRN-bit wordt gereset na het lezen van data uit RDR of bij een master reset.

Parity Error (PE), bit SR6, geeft aan dat de pariteit van het ontvangen karakter niet correct is. De waarde van PE blijft gehandhaafd, zolang het bijbehorende karakter in RDR aanwezig is. Als er datatransport zonder pariteit plaatsvindt, wordt door de zender geen pariteitsbit gegenereerd en blijft PE=0.

Een master reset maakt PE gelijk aan 0.

Interrupt Request (IRQ), bit SR7, geeft de toestand weer van de $\overline{\text{IRQ}}$ -uitgang. Dit wil zeggen dat de IRQ-bit gelijk is aan 1 als de $\overline{\text{IRQ}}$ -uitgang 0 is. De IRQ-bit wordt gereset door een leesoperatie in RDR, respectievelijk een schrijfoperatie in TDR of door een master reset.

Let wel dat bij programmed I/O de bit IRQ niet moet worden getest, aangezien deze niet wordt geset ten gevolge een disabled interrupt. Dit is in tegenstelling tot bij de PIA, waar de interrupt request bits ook worden geset bij disabled interrupts. Bij de ACIA moeten de bits RDRF, respectievelijk TDRE rechtstreeks worden getest in het geval er met programmed I/O (dus bij geblokkeerde ACIA-interrupts) wordt gewerkt.

10.11.8 Control register

Het Control Register CR bestuurt de functie van de ACIA.

CR0 en CR1 zijn de counter divide select bits (zie fig. 10.50). Hiermee wordt de deelfactor ingesteld van de frequenties van zowel de zender als de ontvanger. Bovendien worden deze bits gebruikt voor het geven van een master reset van de ACIA.

CR1	CR0	functie
0	0	frequenties van RxCLK en TxCLK delen door 1
0	1	frequenties van RxCLK en TxCLK delen door 16
1	0	frequenties van RxCLK en TxCLK delen door 64
1	1	master reset

Fig. 10.50 Counter divide select bits

De ACIA heeft geen reset-ingang. Tijdens een master reset worden de bits van het Status Register gereset, uitgezonderd de bits $\overline{\text{CTS}}$ en $\overline{\text{DCD}}$ (zie hiervoor paragraaf 10.11.7). Verder zorgt de master reset voor het initialiseren van het zend- en het ontvang gedeelte van de ACIA. De overige bits van CR worden hierbij niet beïnvloed.

Na het inschakelen van de voedingsspanning moet er eerst een master reset worden gegeven om de ACIA te resetten. Daarna moet de ACIA worden geïnitieerd.

Gedurende de eerste master reset na het inschakelen van de voedingsspanning worden de outputs $\overline{\text{IRQ}}$ en $\overline{\text{RTS}}$ 1 gemaakt. Bij alle volgende master resets wordt de $\overline{\text{IRQ}}$ -uitgang 1 gemaakt, terwijl de $\overline{\text{RTS}}$ -uitgang 1 of 0 blijft, al naar gelang deze geprogrammeerd is (zie fig. 10.50).

De ACIA bevat interne elektronica om het inschakelen van de voedingsspanning te detecteren en om de chip in een reset

toestand te houden. Dit voorkomt verkeerde uitgangssignalen, voordat de ACIA op de juiste wijze is ingesteld. Deze reset toestand wordt verlaten bij een master reset. Pas daarna kan de ACIA worden geïnitieerd met behulp van het programmeerbare Control Register.

De word select bits CR2, CR3 en CR4 worden gebruikt om de woordlengte, de pariteit en het aantal stopbits te selecteren (zie fig. 10.51).

Bij 7-bits woorden is bit 7 van het Transmit Data Register don't care. Het doet er niet toe of deze bit 0 of 1 is. Hij wordt immers niet verzonden.

Bit 7 van het Receive Data Register is in het geval van de 7-bits mode altijd 0.

CR4	CR3	CR2	functie
0	0	0	7 bits + even pariteit + 2 stopbits
0	0	1	7 bits + oneven pariteit + 2 stopbits
0	1	0	7 bits + even pariteit + 1 stopbit
0	1	1	7 bits + oneven pariteit + 1 stopbit
1	0	0	8 bits + 2 stopbits
1	0	1	8 bits + 1 stopbit
1	1	0	8 bits + even pariteit + 1 stopbit
1	1	1	8 bits + oneven pariteit + 1 stopbit

Fig. 10.51 Word select bits

CR6	CR5	functie
0	0	RTS=0 + transmitter interrupt disabled
0	1	RTS=0 + transmitter interrupt enabled
1	0	RTS=1 + transmitter interrupt disabled
1	1	RTS=0 + transmitter interrupt disabled + break niveau op TxD-uitgang

Fig. 10.52 Transmitter control bits

De transmitter control bits CR5 en CR6 besturen de transmitter interrupt enable, het $\overline{\text{RTS}}$ -uitgangssignaal en de transmissie van

een break niveau (zie fig. 10.52).

Transmit Data Register Empty is de transmitter interrupt conditie.

Een transmitter interrupt doorlaten heeft geen zin als Request to Send niet is geactiveerd of als er een break niveau wordt gezonden. Een break niveau zenden houdt in dat er continu nullen worden verstuurd via TxD (zonder start- of stopbits). Het device aan de andere kant van de verbinding zal dit detecteren, doordat daar continu framing errors optreden,

waarbij de ontvangen data \$00 is. Hierop zal dit device stoppen met zenden. Een break niveau wordt gebruikt om het andere device in noodsituaties te laten stoppen met zenden.

De receiver interrupt enable bit CR7 laat toe dat bij een receiver interrupt conditie bit SR7 wordt geset en dat de $\overline{\text{IRQ}}$ -uitgang van de ACIA 0 wordt.

Als RIE=0 is de receiver interrupt geblokkeerd.

Receive Data Register Full, Overrun en een opgaande flank op de $\overline{\text{DCD}}$ -ingang zijn de receiver interrupt condities.

10.11.9 Interrupt request

De Interrupt Request output is actief (0), zolang de oorzaak van de interrupt aanwezig is en de betreffende interrupt ge-enabled is. De IRQ-bit (SR7) is dan 1.

Voor de receiver interrupt enable geldt: RIE = CR7

Voor de transmitter interrupt enable geldt: TIE = $\overline{\text{CR6}}$.CR5

IRQ = TIE.(transmitter interrupt) + RIE.(receiver interrupt)

Hierin is:

transmitter interrupt = TDRE.($\overline{\text{reset}}$)

receiver interrupt = (RDRF + OVRN + $\overline{\text{DCD}}$ opgaand).(reset)

In de reset toestand zijn de interrupts gereset en geldt dus IRQ=0.

10.11.10 Een voorbeeld van seriële communicatie

Voorbeeld 10.10 (Communicatie tussen twee computers)

De inhoud van een geheugenblok van een computer moet worden verzonden naar een andere computer. Gekozen wordt voor een seriële interface via een ACIA.

TxCLOCK en RxCLK worden aangestuurd vanuit een oscillator met een frequentie van 19,2 kHz. De baud rate moet 1200 Bd bedragen. Dit betekent dat de deelfactor 16 moet zijn. De te versturen gegevens bestaan uit 7-bits ASCII-karakters. Er wordt een even pariteitsbit meegezonden en één stopbit. Het aantal karakters per seconde bedraagt dus maximaal 120.

De signalen $\overline{\text{RTS}}$ en $\overline{\text{CTS}}$ worden gebruikt om de communicatie te besturen. De zender activeert $\overline{\text{RTS}}$ om aan te geven dat hij met het datatransport wil beginnen. Pas nadat $\overline{\text{CTS}}$ actief is, kan de bit TDRE geset worden en kan het zenden beginnen. Kan de ontvangende computer de data niet tijdig verwerken, dan maakt hij zijn ACIA-uitgang RTS inactief. Bedenk dat de $\overline{\text{CTS}}$ - en $\overline{\text{RTS}}$ -signalen van zender en ontvanger kruislings met elkaar zijn verbonden, evenals de signalen TxD en RxD (zie fig. 10.53).

De bit TDRE van de zender is dan wederom geblokkeerd, zodat er geen nieuw karakter verzonden wordt, totdat RTS bij de ontvanger en CTS bij de zender weer actief zijn.

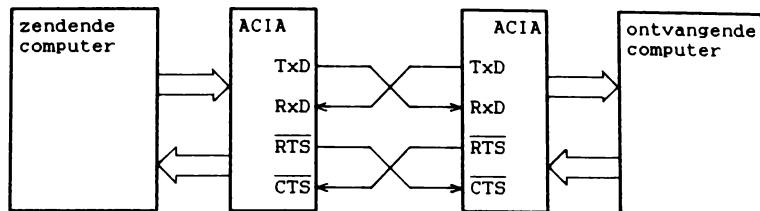


Fig. 10.53 Communicatie tussen computers via ACIA's

De zender begint met het zenden van het ASCII-karakter STX (Start of Text, \$02). Vervolgens worden de gegevens uit het geheugenblok verzonden en tenslotte het karakter ETX (End of Text, \$03).

Is de informatie bij de ontvanger goed aangekomen (geen pariteitsfouten, framing error of overrun), dan zendt deze het karakter ACK (ACKnowledge, \$06) terug, in het andere geval NAK (Negative Acknowledge, \$15).

Als de ontvangst hiervan bij de zender correct is, zet de zender deze terugmelding op de locatie, direct achter het geheugenblok en zendt hij een ACK naar de ontvanger. In het andere geval wordt NUL (NULL, \$00) weggeschreven en wordt er NAK teruggemeld aan de ontvanger. Hierna beëindigt de zender de communicatie door de lijn $\overline{\text{RTS}}$ inactief (1) te maken. De ontvanger zal na de ontvangst van ACK/NAK hetzelfde doen.

Het zenden vindt plaats op basis van programmed I/O, het ontvangen op basis van interrupt-I/O. De IRQ-interruptvector heeft als adres \$DFF8.

Fig. 10.54 en 10.55 geven de structogrammen van het hoofdprogramma en van de interrupt service routine van de zendende computer. Hierin is niet expliciet opgenomen, dat er telkens moet worden gewacht op TDRE=1, voordat een volgend karakter wordt verzonden.

Het niet correct over komen van de terugmelding ACK/NAK van de ontvanger blijkt uit de code NUL achter het geheugenblok.

Bij het begin van de communicatie wordt de BUSY-flag geset. Na het ontvangen van de terugmelding ACK, dan wel NAK, moet de communicatie worden beëindigd. De ISR INPUT stelt het hoofdprogramma hiervan in kennis door het resetten van de BUSY-flag.

Na de structogrammen volgt het bijbehorende programma.

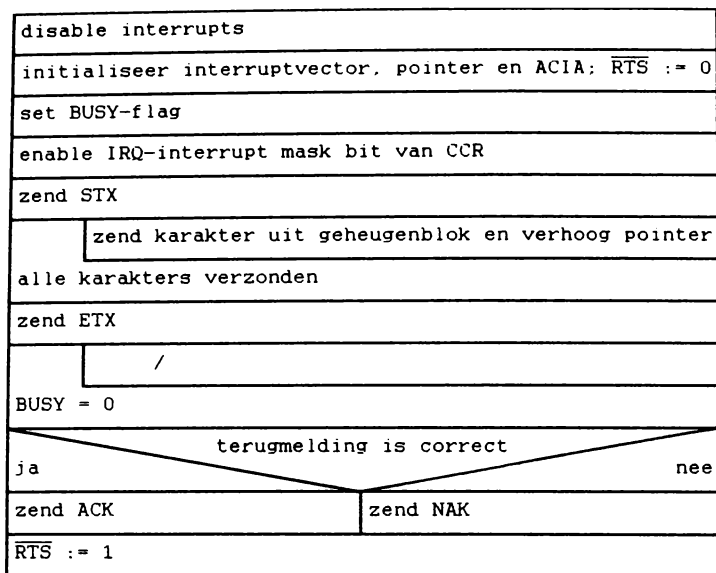


Fig. 10.54 Structogram voor het zenden van data via een ACIA

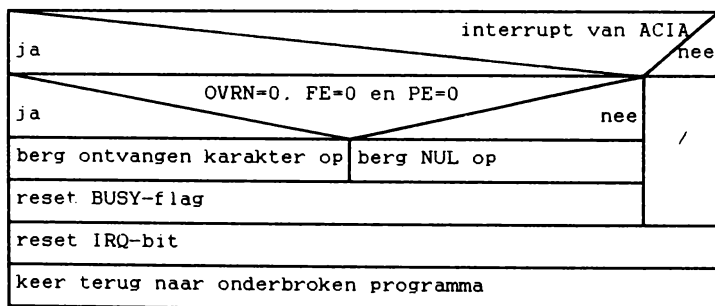


Fig. 10.55 Structogram van IRQ-ISR van zender

```

10          NAM    COMPUTERCOMMUNICATIE VIA ACIA'S
20 *Progr.: J.G. Rouland
30 *Revisie 0    Datum: 12-10-1986
40 *
50 *Het programma TRANS verzorgt de overdracht van gegevens
60 *vanaf geheugenlocatie FIRST t/m LAST, voorafgegaan door
70 *STX en afgesloten met ETX.
80 *De communicatie geschiedt via ACIA's, het zenden op basis
90 *van programmed I/O, het ontvangen met interrupt-I/O.
100 *Er wordt 7-bits data verzonden met even pariteit en 1

```

```

110 *stopbit. De ontvanger behoort ACK of NAK terug te melden.
120 *Deze code wordt opgeslagen op LAST+1, mits hij goed is
130 *ontvangen. Anders wordt NUL opgeslagen op LAST+1. In het
140 *eerste geval antwoordt de zender met ACK. in het tweede
150 *met NAK. Daarna wordt de communicatie beëindigd.
160 *
170 *
180         ORG         $5000
190 NUL      EQU        $00
200 STX      EQU        $02
210 ETX      EQU        $03
220 ACK      EQU        $06
230 NAK      EQU        $15
240 CR       EQU        $FF08
250 SR       EQU        CR
260 TDR      EQU        $FF09
270 RDR      EQU        TDR
280 RTSPAS   EQU        %11001001    ACIA-instelling met  $\overline{RTS}$ =1
290 RTSACT   EQU        %10001001    ACIA-instelling met  $\overline{RTS}$ =0
300 BUSY     RMB        1
310 *
320 TRANS    ORCC       #$50          disable interrupts
330          LDX         #INPUT        laad IRQ-interruptvector met
340          STX         $DFF8         startadres van de IRQ-ISR
350          LDX         #FIRST        initialiseer de pointer
360          BSR         INACIA        initialiseer de ACIA
370          LDA         #$80          set
380          STA         BUSY          BUSY
390          ANDCC       #$EF          enable IRQ-interrupt
400 CTSNA     LDA        SR
410          BITA        #$02          spring als TDRE=0
420          BEQ         CTSNA        ( $\overline{CTS}$  is niet actief)
430          LDA         #STX
440          STA         TDR          zend STX
450 NEXT      LDA        SR
460          BITA        #$02
470          BEQ         NEXT          wacht tot TDRE=1
480          LDA         ,X+          lees karakter en verhoog pointer
490          STA         TDR          verzend karakter
500          CMPX        #LAST+1      spring als nog niet alle
510          BNE         NEXT          karakters zijn verzonden
520 KLAAR     LDA        SR
530          BITA        #$02
540          BEQ         KLAAR        wacht tot TDRE=1
550          LDA         #ETX
560          STA         TDR          zend ETX
570 GEREED    TST        BUSY
580          BMI         GEREED        wacht op terugmelding ontvanger
590          TST         ,X          spring als terugmelding
600          BEQ         NOK          niet foutloos is ontvangen
610 OK        LDA        #ACK
620          BRA         ZEND

```

```

630 NOK      LDA      #NAK
640 ZEND     LDB      SR
650          BITB     #$02
660          BEQ      ZEND      wacht tot TDRE=1
670          STA      TDR       zend antwoord op terugmelding
680          LDA      #RTSPAS
690          STA      CR        RTS:=1, beëindig communicatie
700          SWI
710 *
720 *
730 *****
740 * SUBROUTINE INACIA *
750 *****
760 *
770 INACIA    LDA      #$03
780          STA      CR        reset de ACIA
790          LDA      #RTSACT   initialiseer de ACIA en
800          STA      CR        activeer RTS
810          RTS
820 *
830 *
840 *****
850 * IRQ-ISR INPUT *
860 *****
870 *
880 INPUT     LDA      SR        spring als er geen
890          BPL      RESINT     ACIA-interrupt is
900          BITA     #%01110000 spring als er een
910          BNE      ERROR     communicatiefout is
920 GOED      LDA      RDR       lees terugmelding van ontvanger
930          BRA      ZETWEG
940 ERROR     LDA      #NUL
950 ZETWEG    STA      ,X        schrijf melding op LAST+1
960          CLR      BUSY
970 RESINT    TST      RDR       reset interrupt met dummy read
980          RTI

```

Bij dit programma is er van uit gegaan dat de ontvanger slechts antwoordt met ACK of NAK en wel nadat de volledige tekst is overgestuurd. Derhalve wordt de terugmelding als zodanig niet gecontroleerd.

Het is bij communicatie doorgaans noodzakelijk om een time-out in te bouwen. Dit houdt in dat niet meer op antwoord wordt gewacht na het overschrijden van een bepaalde tijdlimiet. Gebeurt dit niet, dan loopt men het risico dat het programma blijft steken. Wordt de verbinding verbroken of wordt de voedingsspanning van het apparaat aan de andere kant van de verbinding uitgeschakeld, dan komt er nooit een terugmelding. Hiertegen moet men zijn programma beveiligen met behulp van een time-out. Dit kan worden gerealiseerd door de gestelde tijdlimiet in zijn programma af te tellen. Het nadeel hiervan

is, dat de computer geen andere activiteiten gedurende het aftellen kan ondernemen. Om dit te voorkomen kan men een externe schakeling aansturen, die na het verstrijken van de gewenste tijd een interrupt genereert. Zulke schakelingen zijn als IC verkrijgbaar. Sommige zijn geheel aangepast om aan te sluiten op een microprocessor. In de volgende paragraaf zal zo'n schakeling worden behandeld.

10.12 Programmable Timer Module

10.12.1 De opbouw van de PTM

De Programmable Timer Module, PTM (type MC6840), is een programmeerbare bouwsteen, die verkrijgbaar is in drie versies: MC6840, MC68A40 en MC68B40. Deze werken respectievelijk met een klokfrequentie van 1, 1.5 en 2 MHz.

Operatiemodes

De PTM heeft de volgende operatiemodes:

1. wave synthesis mode
 - continuous mode
 - * voor het genereren van een kloksignaal met programmeerbare frequentie en pulsbreedte
 - * voor het genereren van interrupts op programmeerbare tussenpozen
 - single-shot mode
 - * voor het genereren van een enkele puls met een programmeerbare duur en na een programmeerbare wachttijd
2. wave measurement mode
 - frequency comparison or period measurement mode
 - * voor het vergelijken van een periodetijd met een programmeerbare waarde
 - pulse width comparison mode
 - * voor het vergelijken van een pulsbreedte met een programmeerbare waarde

De PTM kan worden gebruikt

- als programmeerbare klokgenerator
- als one-shot met instelbare vertraging
- voor het meten van pulsbreedten
- voor het meten van frequenties
- voor het instellen van een bepaalde time-out.

Timers

De PTM bevat drie timers. Elke timer kan onafhankelijk van de anderen worden ingesteld in een van de operatiemodes. Elke timer is opgebouwd uit een 16-bits teller met bijbehorende latches, een control register en besturingslogica (zie fig. 10.52). De drie timers hebben een gemeenschappelijk status register.

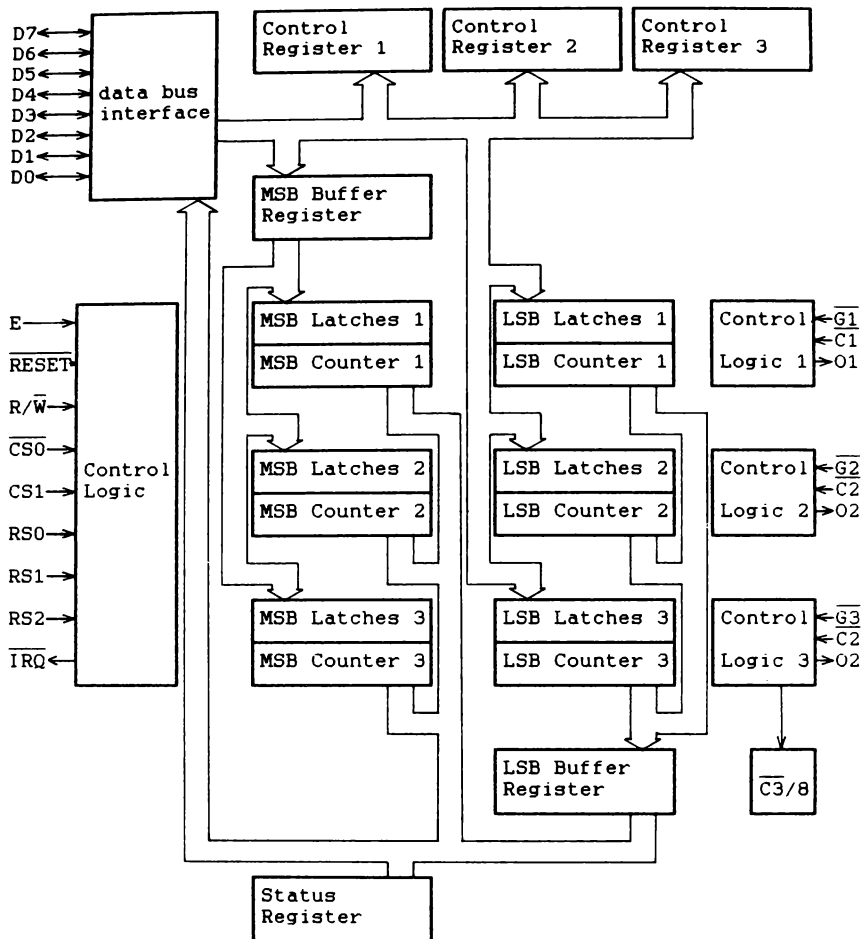


Fig. 10.56 Blokschema van de PTM

De latches (flipflops) vormen de registers waarin de tijden, die door de tellers moeten worden afgeteld, bewaard worden. Deze tijden worden in deze registers als het ware vergrendeld (latch = grendel). De waarden die in de latches opgeslagen zijn, veranderen niet bij het aftellen. Ze veranderen pas als er nieuwe informatie in de latches wordt geschreven.

Het aftellen gebeurt in de tellers zelf. Deze worden tijdens een zogenaamde tellerinitialisatiecyclus geladen met de informatie van de overeenkomstige latches. Verder wordt een

teller telkens opnieuw geladen vanuit de corresponderende latches bij de klokpuls die optreedt, nadat de teller 0 is geworden. Het aftellen is dus een cyclisch proces. Na het starten van de teller wordt deze bij elke klokpuls verlaagd. Als klokpuls kan de microprocessorklok E of de externe klok $\overline{C^*}$ worden gebruikt. De "*" staat voor 1, 2 of 3, behorende bij timer 1, timer 2 of timer 3.

Latches initialiseren

Omdat de PTM slechts 8 data-ingangen heeft, zouden de 2 bytes van de latches niet tegelijk kunnen worden gevuld. Hierdoor zou de teller foutief worden geïnitieerd, als dit gebeurt tussen het laden van de high-byte en de low-byte. Om dit te voorkomen is de PTM voorzien van een (gemeenschappelijk) MSB Buffer Register. Hierin wordt tijdelijk de high-byte opgeslagen.

Bij het laden van de 8 minstwaardige latches van een timer wordt de inhoud van het MSB Buffer Register in de bijbehorende 8 meestwaardige latches opgeslagen. Dit betekent dat de high-byte eerst moet worden opgeslagen in de PTM en daarna pas de low-byte.

Gebruiken we een store-instructie vanuit een 2-bytes register, dan gebeurt dit automatisch (zie paragraaf 6.3.2). Om hiervan gebruik te kunnen maken, is het MSB Buffer Register bereikbaar via drie verschillende adressen en zijn deze telkens 1 lager dan de adressen van de low-bytes van de bijbehorende latches. De latches en het MSB Buffer Register zijn write only registers.

Tellerstand lezen

Bij het lezen treedt een soortgelijk probleem op als bij het initialiseren van de latches. Ook hier kan slechts de helft van de teller tegelijk worden gelezen, zodat een onjuiste waarde zou worden gelezen, daar bij het lezen van de tweede helft de tellerstand inmiddels is verlaagd. Dit probleem wordt ondervangen door het LSB Buffer Register.

Bij het lezen van het meestwaardige deel van de teller wordt het minstwaardige deel automatisch overgenomen in het (gemeenschappelijke) LSB Buffer Register. Dit impliceert dat eerst het minstwaardige deel van de teller moet worden gelezen en daarna het meestwaardige deel.

Bij een load-instructie met een 2-bytes register gebeurt dit automatisch (zie paragraaf 6.3.2). Om dit voor de drie (read only) tellers te kunnen toepassen, is het (read only) LSB Buffer Register via drie adressen bereikbaar. Elk adres is 1 hoger dan het adres van de meestwaardige helft van de bijbehorende teller.

Status Register

De status van de PTM wordt vastgelegd in het Status Register. Hierin wordt met name vastgelegd welke interrupt is gegenereerd (zie paragraaf 10.12.5).

Communicatie tussen PTM en microprocessor

Datatransport tussen microprocessor en PTM geschiedt via de databus, onder besturing van de klokpuls E.

Het signaal E zorgt verder voor de synchronisatie van de RESET en van de externe klokpulsen en gate inputs van de PTM.

Het R/W-signaal bestuurt de richting van de data. Voor R/W=0 wordt er geschreven in de PTM, voor R/W=1 gelezen.

De microprocessor heeft alleen toegang tot de PTM als $\overline{CS0}=0$ en $CS1=1$. Deze Chip Select ingangen moeten derhalve via een adresdecoder verbonden zijn met de adresbus.

De (open drain) $\overline{IRQ}$ -uitgang is geactiveerd (0) als de composite interrupt flag (SR7) geset is.

10.12.2 Reset

Externe reset

Na het detecteren van een 0 op de RESET-ingang wordt de PTM gereset:

- Alle latches worden met hun maximale waarde (\$FFFF = 65.535) geladen.
- Alle Control Register bits worden gereset, uitgezonderd bit CR10 (de interne resetbit). CR10 wordt geset. CR10 is bit 0 van Control Register 1, het control register van timer 1.
- Alle tellers worden geladen met de waarden van de latches.
- De uitgangen O1, O2 en O3 worden gereset.
- Alle tellerklokken worden gedisabled.
- Alle bits van het Status Register worden gereset.

Interne reset

Tijdens een interne reset (CR10=1) gebeurt het volgende:

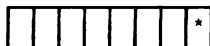
- Alle tellers worden geladen met de waarden van de bijbehorende latches.
- De uitgangen O1, O2 en O3 worden gereset.
- Alle tellerklokken worden gedisabled.
- Alle bits van het Status Register worden gereset.

De latches en de Control Registers worden niet beïnvloed door een interne reset. Zij kunnen tijdens een interne reset zelfs worden geladen met informatie.

10.12.3 Registerselectie

0

CR2



Via de Register Select inputs RS0, RS1 en RS2 (samen met het R/W-signaal) worden de interne registers van de PTM geselecteerd (zie fig. 10.57).

Register Select inputs			geselecteerde register	
RS2	RS1	RS0	R/W = 0	R/W = 1
0	0	0	Control Register 3 als CR20=0	geen enkel register
0	0	1	Control Register 1 als CR20=1	geen enkel register
0	1	0	Control Register 2	Status Register
0	1	1	MSB Buffer Register	High-byte Counter 1
1	0	0	Low-byte latches 1	LSB Buffer Register
1	0	1	MSB Buffer Register	High-byte Counter 2
1	1	0	Low-byte latches 2	LSB Buffer Register
1	1	1	MSB Buffer Register	High-byte Counter 3
1	1	1	Low-byte latches 3	LSB Buffer Register

Fig. 10.57 Adressering van de PTM-registers

Aangezien er negen write only registers zijn en slechts 3 Register Select ingangen, fungeert bit 0 van Control Register 2 als extra adresbit. CR1 en CR3 hebben hetzelfde externe adres (zie fig. 10.57) en zijn derhalve niet altijd direct toegankelijk. Eerst moet bit CR20 de juiste waarde krijgen. Een soortgelijke oplossing hebt u reeds gezien bij de PIA (zie paragraaf 10.6).

Net als bij de ACIA zijn de registers van de PTM hetzij read only, hetzij write only.

Indien nodig moet een kopie van de in de (write only) registers geschreven informatie worden bewaard.

Zogenaamde read-modify-write instructies mogen niet worden toegepast. Bij deze instructies, zoals INC en ROL, wordt er een byte opgehaald uit een bepaalde geheugenlocatie, deze wordt gewijzigd en tenslotte wordt de gewijzigde waarde op dezelfde locatie teruggedeschreven. Dit terugschrijven gebeurt bij de PTM (net als bij de ACIA) in een ander register, omdat bij het lezen en schrijven op hetzelfde adres verschillende registers worden geselecteerd (zie fig. 10.49 en 10.57).

10.12.4 Control Registers

De bits van de drie Control Registers (CR1, CR2 en CR3) worden gebruikt voor het instellen van de operatiemodus van de corresponderende timers. Hierbij zij opgemerkt dat de bits 1 t/m 7 gemeenschappelijke functies selecteren, die echter alleen betrekking hebben op de overeenkomstige timer. De bits 0 (CR10, CR20 en CR30) daarentegen selecteren elk een unieke functie.

Controlbits 0 selecteren unieke functies

CR10 is de interne resetbit:

CR10 = 0: de drie timers kunnen hun ingestelde functie uitvoeren

CR10 = 1: er treedt een interne reset op (zie paragraaf 10.11.2)

CR20 is de interne adresbit voor CR1 en CR3:

CR20 = 0: CR3 is adresseerbaar (alleen tijdens het schrijven)

CR20 = 1: CR1 is adresseerbaar (alleen tijdens het schrijven)

CR30 is klokbesturingsbit van timer 3:

CR30 = 0: de klokfrequentie voor timer 3 wordt niet gedeeld

CR30 = 1: de klokfrequentie voor timer 3 wordt gedeeld door 8

De frequentiedeling kan zowel worden toegepast op de interne klokpuls E als op de externe klokpuls $\overline{C3}$. De deling is alleen van invloed op timer 3!

Controlbits 1 t/m 7 selecteren gemeenschappelijke functies voor elke timer afzonderlijk

Voor deze controlbits wordt het nummer van de bijbehorende timer (1, 2 of 3) door "*" vervangen. Deze bits kunnen in elk Control Register onafhankelijk van elkaar worden ingesteld, zodat elke timer in een andere mode kan werken.

CR*1 is de klokselectiebit:

CR*1 = 0: timer * gebruikt de externe klokingang $\overline{C^*}$

CR*1 = 1: timer * gebruikt de interne klokpuls E

CR*2 is de Counting Mode Control bit

CR*2 = 0: normal 16-bits mode

De latches * bevatten een 16-bits getal.

Teller * wordt 0 na (N+1) klokperiodes vanaf $\overline{G^* L}$.

N is het 16-bits getal in latches *.

CR*2 = 1: dual 8-bits mode

De latches * bevatten twee 8-bits getallen

Teller * wordt 0 na (L+1)·(M+1) klokperiodes vanaf $\overline{G^* L}$.

L is de low-byte van latches *;

M is de high-byte van latches *.

CR5	CR4	CR3	operatiemode
0	0	0	continuous mode; tellerinitialisatie bij $\overline{G^* L}$ of schrijven in latches * of reset
0	0	1	frequency comparison mode; interrupt als (periodes $\overline{G^*}$) < (time-out *)
0	1	0	continuous mode; tellerinitialisatie bij $\overline{G^* L}$ of reset
0	1	1	pulse width comparison mode; interrupt als (0-puls op $\overline{G^*}$) < (time-out *)
1	0	0	single shot mode; tellerinitialisatie bij $\overline{G^* L}$ of schrijven in latches * of reset
1	0	1	frequency comparison mode; interrupt als (periodes $\overline{G^*}$) > (time-out *)
1	1	0	single shot mode; tellerinitialisatie bij $\overline{G^* L}$ of reset
1	1	1	pulse width comparison mode; interrupt als (0-puls op $\overline{G^*}$) > (time-out *)

Fig. 10.58 De selectiebits voor de operatiemodes

De bits CR*3, CR*4 en CR*5 selecteren de operatiemodus en de tellerinitialisatie van de timers (zie fig. 10.58). Met reset wordt een externe of een interne reset bedoeld.

Tellerinitialisatie

De tellerinitialisatie is gedefinieerd als de overdracht van data van de latches naar de bijbehorende teller. Hierbij wordt tevens de corresponderende individuele interrupt flag I* gereset.

Zoals uit fig. 10.58 blijkt, treedt tellerinitialisatie altijd op bij een externe reset (RESET=0) en bij een interne reset (CR10=1). Afhankelijk van de operatiemodus kan dit bovendien gebeuren bij het schrijven in de latches of bij het detecteren van een neergaande flank op de $\overline{G^*}$ -ingang.

Recycling of herinitialisatie van de teller

Het herstarten van een teller vindt plaats als er een negatieve flank op de bijbehorende klokkingang wordt gedetecteerd nadat de tellerstand 0 is geworden. De teller wordt dan geladen vanuit de latches.

CR*6 is de interrupt enable bit:

CR*6 = 0: interrupt I* van timer * is gedisable

CR*6 = 1: interrupt I* van timer * is ge-enabled

CR*7 is de timer output enable bit:

CR*7 = 0: output O* is gedisable

CR*7 = 1: output O* is ge-enabled

10.12.5 Status Register

Het read only Status Register bevat vier interruptbits. De overige vier bits worden niet gebruikt en zijn bij lezen altijd gelijk aan 0.

De bits 0, 1 en 2 zijn de individuele interruptbits I1, I2 en I3 behorende bij de timers 1, 2 en 3.

Bit 7 is de samengestelde interruptbit. Deze wordt geset indien een van de individuele interrupt flags is geset, mits die interrupt is ge-enabled (bit CR*6=1). Dus er geldt:

$$SR7 = I1 \cdot CR16 + I2 \cdot CR26 + I3 \cdot CR36$$

Ook hier geldt (net als bij de ACIA) dat de gemeenschappelijke interruptbit SR7 de toestand van de $\overline{IRQ}$ -uitgang weergeeft. Dit betekent dat disabled interrupts (bij programmed I/O) rechtstreeks moeten worden getest op de individuele interruptbits, omdat disabled interrupts SR7 niet setten.

Een individuele interrupt bit I* wordt in de wave synthesis mode geset bij een time-out en in de wave measurement mode bij bepaalde meetresultaten (zie de paragrafen 10.12.7 t/m 10.12.10).

Een individuele interrupt flag wordt gereset bij een interne of

externe reset, bij het schrijven in de corresponderende latches en bij de corresponderende tellerinitialisatie. Verder wordt I* gereset bij het lezen van de stand van teller *, mits tevoren het Status Register is gelezen, terwijl I* geset was. Deze voorwaarde is in de PTM geïmplementeerd om te voorkomen dat er een interrupt gemist wordt. Dit zou gebeuren als de interrupt optreedt na het lezen van het Status Register, maar voor het lezen van teller *.

10.12.6 Clock inputs, gate inputs en timer outputs

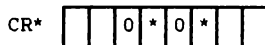
De clock inputs $\overline{C1}$, $\overline{C2}$ en $\overline{C3}$ en de gate inputs $\overline{G1}$, $\overline{G2}$ en $\overline{G3}$ mogen asynchroon zijn ten opzichte van de systeemklok E. Deze ingangen worden ingelezen met de klok E en op deze wijze gesynchroniseerd. Het synchroniseren en verwerken van de ingangssignalen kost drie periodetijden van E. Op de vierde E-puls wordt de bijbehorende teller verlaagd (met betrekking tot $\overline{C*}$) of geïnitieerd of vindt de tijdmeting plaats (met betrekking tot $\overline{G*}$). Er treedt dus een tijdverschuiving op tussen de externe ingangssignalen en de interne reactie daarop. Bij ingeschakelde 8-deler voor $\overline{C3}$ geldt het uitgangssignaal van deze deler als klokinput. Dit houdt in dat de maximaal toelaatbare frequentie van $\overline{C3}$ met ingeschakelde deler ongeveer 8 keer zo groot mag zijn. Op $\overline{G3}$ heeft de deler geen invloed.

De timer uitgangen O1, O2 en O3 leveren in de continuous mode en in de single shot mode een gedefinieerde golfvorm. Is bit CR*7 gelijk aan 0, dan blijft de uitgang O* gelijk aan 0, ongeacht de operatiemodus.

De golfvorm is niet voorspelbaar in de frequency en pulse width comparison mode, tenzij CR*7 = 0 is.

10.12.7 Continuous operating mode

5 4 3 2



De continuous operating mode (CR*3 = CR*5 = 0) kan worden gebruikt om

- op programmeerbare tijdintervallen interrupts te genereren, mits interrupt I* is ge-enabled (dus mits CR*6 = 1) en/of
- een blokgolf op te wekken op de timeruitgang O*, mits deze is ge-enabled (dus mits CR*7 = 1).

Werking

Het initialiseren van de teller geschiedt bij een interne of externe reset of bij het detecteren van een neergaande flank op de gate input $\overline{G*}$.

Als $CR^*4=0$ gebeurt dit tevens bij het schrijven in de latches.

Na het initialiseren van de teller vindt het verlagen van de teller telkens plaats bij het detecteren van een kloksignaal, zolang er geen reset is en zolang $\bar{G}^*$ gelijk aan 0 is.

Time-out

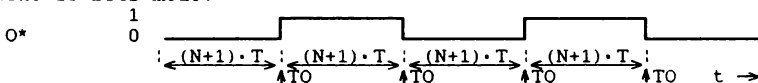
Een zogenaamde time-out treedt op bij de eerste tellerklok na het 0 worden van de teller. Hierbij wordt de individuele interrupt flag I^* geset en wordt de teller herstart, met andere woorden de teller wordt opnieuw geladen met de informatie in de bijbehorende latches.

Is de inhoud van de latches gelijk aan 0, dan treedt er bij elke klokpuls een time-out op. De stand van de teller blijft 0. De output O^* (mits ge-enabled) verandert bij elke klokpuls van waarde, zodat de frequentie van het uitgangssignaal de helft is van de klokpulsfrequentie.

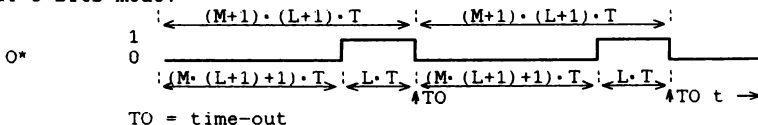
Golfvormen

Nu zijn er twee soorten blokgolven te selecteren (zie fig. 10.59). De ene treedt op in de gewone 16-bits mode, de andere in de dual 8-bits mode.

gewone 16-bits mode:



dual 8-bits mode:



TO = time-out

Fig. 10.59 Timer output O^* in the continuous operating mode

T = periodetijd van E als $CR^*1 = 1$

T = periodetijd van $\bar{C}^*$ als $CR^*1 = 0$

$T = 8 \cdot \text{periodetijd van } E \text{ of } \bar{C}^* \text{ voor timer 3 als } CR^*30 = 1$

M is de waarde van de high-byte van de latches, L is de waarde van de low-byte en N is de waarde van het gehele 16-bits getal in de latches.

De 16-bits mode ($CR^*2=0$)

De 16-bits teller wordt bij elke klokpuls met 1 verlaagd. Dit betekent dat de teller na N klokpulsen 0 is geworden. Bij de volgende klokpuls treedt een time-out op en wordt de teller herstart. Er treedt dus elke $(N+1) \cdot T$ seconde een time-out op. Is output O^* ge-enabled, dan verandert O^* elke $(N+1) \cdot T$ seconde van niveau, zodat de duur van het 0-niveau gelijk is aan de

duur van het 1-niveau. We zeggen dat de duty cycle 50% bedraagt. De duty cycle is gedefinieerd als de verhouding tussen de duur van de 1-puls en de periodetijd. De frequentie van het uitgangssignaal is $1/(2(N+1) \cdot T)$ Hz.

De dual 8-bits mode ($CR*2=1$)

Bij elke klokpuls wordt de low-byte van de teller met 1 verlaagd. Is de low-byte 0 geworden, dan wordt deze bij de volgende klokpuls opnieuw geladen met de waarde van de low-byte van de bijbehorende latches en de high-byte van de teller wordt met 1 verlaagd. De high-byte wordt dus telkens na $(L+1)$ klokpulsen verlaagd en wordt derhalve gelijk aan 0 na $M \cdot (L+1)$ klokpulsen. De output O^* wordt 1 op de klokpuls die volgt op het 0 worden van de high-byte (mits O^* is ge-enabled). De duur van de 0-puls van O^* bedraagt dus $(M \cdot (L+1) + 1)$ klokperiodes (zie fig. 10.59).

U hebt gezien dat de high-byte van de teller na $M \cdot (L+1)$ klokperiodes 0 is geworden. De low-byte is op dat moment opnieuw geladen met de inhoud L van de low-byte van de latches. L klokpulsen later zal ook de low-byte van de teller weer 0 zijn. Dit is dus na $(M \cdot (L+1) + L)$ klokpulsen, gerekend vanaf het initialiseren van de teller. Nu is de inhoud van de gehele teller 0 geworden. Bij de volgende klokpuls treedt er een time-out op. Dit is dus na totaal $(M \cdot (L+1) + L + 1) = (M+1) \cdot (L+1)$ klokperiodes. Hierbij wordt O^* gelijk aan 0, zodat de duur van het 1-niveau van O^* gelijk is aan L klokperiodes.

Ook wordt de teller herstart, zodat de gehele procedure wordt herhaald, zodat er telkens na $(M+1) \cdot (L+1) \cdot T$ seconden een time-out wordt gegenereerd.

U ziet dat niet alleen de frequentie, maar ook de duty cycle programmeerbaar is. De frequentie van het uitgangssignaal van de teller bedraagt $1/((M+1) \cdot (L+1))$ Hz en de duty cycle is $L/((M+1) \cdot (L+1))$.

Bijzonder geval: $L=0$

Indien $L=0$ in de dual 8-bits mode, ontstaat er een toestand die veel lijkt op de 16-bits mode. Telkens na $(M+1)$ klokpulsen treedt er een time-out op. Bij de 16-bits mode gebeurt dit telkens na $(N+1)$ klokpulsen! Bij elke time-out wordt er een interrupt gegenereerd (mits deze ge-enabled is) en verandert de output O^* van waarde (mits deze is ge-enabled). Als $L=0$ heeft het uitgangssignaal op O^* een frequentie van $1/(2(M+1) \cdot T)$ en een duty cycle van 50%.

16-bits mode versus dual 8-bits mode

	gewone 16-bits mode	dual 8-bits mode
duty cycle: 50 %		$L/((M+1) \cdot (L+1))$
frequentie: $1/(2(N+1) \cdot T)$ Hz		$1/((M+1) \cdot (L+1) \cdot T)$ Hz als $L > 0$ $1/(2(M+1) \cdot T)$ Hz als $L = 0$
time-out: na elke $(N+1) \cdot T$ s	na elke $(M+1) \cdot (L+1) \cdot T$ s	
	= min T s	= min T s
	= max $65.536 \cdot T$ s	= max $65.536 \cdot T$ s

Toepassingen

De continuous operating mode kan gebruikt worden voor het genereren van een programmeerbare blok golf en voor het genereren van interrupts met een programmeerbare herhalingsfrequentie. Deze interrupts kunnen bijvoorbeeld worden gebruikt om op gezette tijden gegevens binnen te halen van sensoren.

Ook is het mogelijk om op programmeerbare tijdintervallen het lopende programma te onderbreken en de besturing bijvoorbeeld aan een ander programma te geven. Zo kunnen diverse programma's om beurten gedurende een bepaalde tijd worden uitgevoerd. Op deze wijze worden verschillende programma's als het ware parallel, ogenschijnlijk gelijktijdig, uitgevoerd.

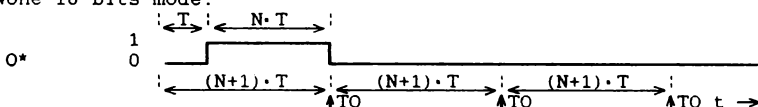
10.12.8 Single-shot operating mode

5 4 3 2

CR*

		1	*	0	*		
--	--	---	---	---	---	--	--

gewone 16-bits mode:



dual 8-bits mode:

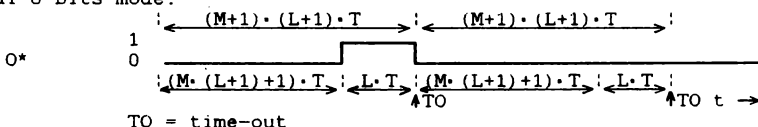


Fig. 10.60 Timer output O* in the single-shot operating mode

De dual 8-bits mode (CR*5=1 en CR*3=0) is identiek aan de continuous operating mode, afgezien van drie uitzonderingen:

1. De output O* is slechts voor één puls ge-enabled.
Bij de eerste time-out keert de output O* terug naar 0 en blijft 0 (totdat de teller opnieuw wordt geïnitieerd). Er treedt dus slechts één enkele uitgangspuls op, vandaar de naam single-shot.
Het inwendige telmechanisme blijft echter cyclisch. Bij elke time-out wordt de individuele interrupt flag I* geset en wordt de teller herstart (zie fig. 10.60).
2. Eenmaal ge-enabled is de tellerklok onafhankelijk van de gate input $\overline{G^*}$.
De input $\overline{G^*}$ hoeft niet 0 te blijven. Ook als $\overline{G^*}$ hoog zou worden in de single-shot mode, blijft de tellerklok ge-enabled.
3. L=M=0 of N=0 blokkeert het uitgangssignaal op O*.
Als L=M=0 (in de dual 8-bits mode) of als N=0 (in de 16-bits mode) gaat de uitgang naar 0 op de eerste klokpuls tijdens

of na de tellerinitialisatie. 0^* blijft 0 totdat de operatiemode wordt veranderd, of totdat er data in de latches wordt geschreven, die ongelijk aan 0 is. Time-outs treden aan het einde van elke klokperiode op als $L=M=0$, respectievelijk als $N=0$.

16-bits mode versus dual 8-bits mode

	gewone 16-bits mode	dual 8-bits mode
puls:	na T s	na $((M \cdot (L+1)+1) \cdot T$ s
pulsbreedte:	N · T s als $N > 0$ 0 s als $N = 0$	L · T s als $N > 0$ 0 s als $L = M = 0$
time-out:	na elke $(N+1) \cdot T$ s = min T s = max $65.536 \cdot T$ s	na elke $(M+1) \cdot (L+1) \cdot T$ s = min T s = max $65.536 \cdot T$ s

10.12.9 Frequency comparison mode

5 4 3 2

CR*

		*	0	1	*		
--	--	---	---	---	---	--	--

In deze mode ($CR*4=0$ en $CR*3=1$) kan de periodetijd van het ingangssignaal op $\overline{G^*}$ worden vergeleken met de geprogrammeerde time-out (zie paragraaf 10.12.7).

Uit de periodetijd kan de frequentie worden berekend.

De timer kan werken in de 16-bits mode of in de dual 8-bits mode ($CR*2$ is 0, respectievelijk 1).

Het initialiseren van de teller kan (behalve met een reset) alleen gebeuren op een neergaande flank van de gate input $\overline{G^*}$, waarbij de interrupt flag $I^*=0$ moet zijn.

De teller wordt geblokkeerd door een reset, door het schrijven in de latches en door het zetten van I^* .

De teller wordt ge-enabled op een neergaande flank van $\overline{G^*}$, mits er geen reset is, $I^*=0$ is en er niet wordt geschreven in de latches.

Frequentievergelijking voor $CR*5=1$

De interrupt flag I^* wordt in dit geval geset als de time-out optreedt vóór de eerste neergaande flank op $\overline{G^*}$ na de tellerinitialisatie, met andere woorden als de periodetijd van $\overline{G^*}$ groter is dan de time-out. De teller wordt nu geblokkeerd en kan pas worden geïnitieerd na het resetten van I^* en na het detecteren van een neergaande flank op $\overline{G^*}$.

Treedt de neergaande flank op $\overline{G^*}$ eerder op dan de time-out, dan wordt de teller opnieuw geïnitieerd. Hierbij treedt geen interrupt op. Het proces van frequentievergelijking wordt pas gestopt als de operatiemode wordt gewijzigd of als een periodetijd van $\overline{G^*}$ voldoet aan de voorgeschreven limiet.

Deze mode wordt toegepast om een tijdintervalvergelijking te doen.

Periodetijdmeting voor CR*5=0

Als CR*5=0, wordt I* geset als er een neergaande flank op $\overline{G^*}$ verschijnt vóór het optreden van de time-out, dus als de periodetijd van $\overline{G^*}$ kleiner is dan de time-out. De teller wordt geblokkeerd.

Komt de time-out eerder, dan blijft de teller ge-enabled. Er wordt intern een bit geset, die het zetten van I* verhindert, totdat de teller wordt geïnitieerd. Als deze interne bit is geset, veroorzaakt een neergaande flank op $\overline{G^*}$ een nieuwe tellerinitialisatie.

Deze mode kan worden toegepast om een tijdintervallmeting te doen. De teller stopt immers als de periodetijd van $\overline{G^*}$ kleiner is dan de time-out. Tevens wordt er een interrupt gegenereerd. Door de tellerstand af te trekken van de beginwaarde (de waarde in de latches) vinden we de periodetijd (uitgedrukt in T).

Het gedrag van de uitgangen O*

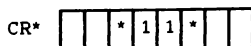
De output O* wordt gewoonlijk niet gebruikt in de wave measurement modes (ook wel time interval modes genoemd). Zowel in de frequency (of period) measurement mode als in de pulse width comparison mode werken de outputs O*, mits enabled, als volgt:

- O* zal 0 zijn tussen de herinitialisatie van de teller en de eerste time-out.
- Na de eerste time-out zal O* gelijk worden aan 1.
- Als er hierna meerdere time-outs optreden, zal de uitgang O* bij elke time-out van toestand veranderen.

Het uitgangssignaal O* is hier veelal onvoorspelbaar, omdat de time-out niet altijd optreedt. Dit hangt af van de periodetijd, respectievelijk van de pulsbreedte van $\overline{G^*}$. In de time interval modes zijn deze waarden meestal onbekend. Ze moeten juist worden gemeten.

10.12.10 Pulse width comparison mode

5 4 3 2



Deze mode (CR*3=CR*4=1) is te vergelijken met de frequency comparison mode. Het verschil is dat bij de pulse width comparison mode een opgaande flank in plaats van een neergaande flank van $\overline{G^*}$ het tellen stopt. Hierbij kan de pulsbreedte van het signaal op $\overline{G^*}$ worden vergeleken met de geprogrammeerde time-out. Het gaat hierbij om de duur van het 0-niveau op $\overline{G^*}$. De timer kan werken in de 16-bits mode of in de dual 8-bits mode.

Het initialiseren van de teller kan (behalve met een reset) alleen gebeuren op een neergaande flank van de gate input $\overline{G^*}$.

waarbij de interrupt flag $I^*=0$ moet zijn.

De teller wordt geblokkeerd door een reset, door het schrijven in de latches, door het zetten van I^* en door een opgaande flank op $\overline{G^*}$.

De teller wordt ge-enabled op een neergaande flank van $\overline{G^*}$, mits er geen reset is, $I^*=0$ is en er niet wordt geschreven in de latches.

Pulsbreedtevergelijking voor CR*5=1

Als $CR^*5=1$, wordt I^* geset als de time-out optreedt voor de opgaande flank van $\overline{G^*}$, met andere woorden als de breedte van de 0-puls op $\overline{G^*}$ groter is dan de time-out.

Deze mode wordt gebruikt om een pulsbreedtevergelijking te doen.

Pulsbreedtemeting voor CR*5=0

Als $CR^*5=0$, wordt I^* geset als de opgaande flank op $\overline{G^*}$ verschijnt, voordat de time-out optreedt. In dit geval is de duur van de 0-puls kleiner dan de geprogrammeerde time-out. Doordat de teller geblokkeerd wordt bij een opgaande flank op $\overline{G^*}$, kan op deze wijze de breedte worden bepaald van de 0-puls op $\overline{G^*}$, mits deze puls een interrupt veroorzaakt. In dit geval stopt de teller, voordat hij opnieuw wordt geladen met de inhoud van de latches. De pulsbreedte kan worden berekend uit het aantal malen dat de teller is verminderd met 1. Dit is het verschil van de oorspronkelijke tellerstand en de tellerstand die optreedt bij de interrupt. De breedte van de 0-puls op $\overline{G^*}$ is dan gelijk aan $((\text{inhoud latches}) - (\text{inhoud teller})) \cdot T$.

10.12.11 Voorbeelden

Voorbeeld 10.11 (Het genereren van een blok golf)

Veronderstel dat een hardwareschakeling een kloksignaal nodig heeft met een periodetijd van 8 ms en een pulsbreedte (van het 1-niveau) van 240 μs . Dit signaal zullen we genereren met behulp van teller 1 van de PTM. De instelling van de PTM zal zodanig moeten zijn dat voldaan wordt aan de volgende eisen:

- Het kloksignaal moet continu aanwezig zijn. Dit betekent dat de PTM moet werken in de continuous operating mode ($CR13=CR15=0$) en de uitgang moet zijn ge-enabled ($CR17=1$).
- De teller moet worden geïnitieerd door te schrijven in de latches ($CR14=0$).
- Er moet geen interrupt worden gegenereerd, zodat deze gedisableerd moet zijn ($CR16=0$).
- Ingang $\overline{G1}$ moet 0 zijn, opdat teller 1 ge-enabled is.
- De duty cycle is ongelijk aan 50%, zodat de dual 8-bits mode moet worden ingesteld ($CR12=1$).
- De pulsbreedte van het 1-niveau bedraagt 240 μs . Dit wordt bereikt door 240 perioden van de klokpuls E af te tellen, waarbij E een frequentie heeft van 1 MHz. Derhalve moet

- geldt L=240, terwijl E als tellerklok moet worden geselecteerd (CR11=1).
- De periodetijd moet 8 ms bedragen. Dit betekent dat $(M+1) \cdot (L+1)$ gelijk moet zijn aan 8000, dus $M = 8000/241 - 1 = 32$.
 - Van de niet-gebruikte tellers worden de uitgangen en de interrupts gedisabled.

Veronderstel dat het basisadres van de PTM \$FF08 is.
De initialisatieroutine van de PTM ziet er als volgt uit:

```

100 INIPTM CLR PTMCR2      adresseer PTMCR3
110      CLR PTMCR3      disable output 03 en interrupt 13
120      LDA #1          adresseer PTMCR1 en disable
130      STA PTMCR2      output 02 en interrupt 12
140      LDA #%10000110  enable output 01, disable
150      STA PTMCR1      interrupt 11, dual 8-bits
160 *          continuous operating met write-
170 *          initialisatie, E-klok, reset=0
180      LDA #32
190      LDB #240
200      STD PTMHR1      M1:=32, L1:=240 (periode=7953 µs).
210 *          initialiseer en enable teller 1
220      RTS

```

De toewijzing van de fysische adressen van de registers van de PTM kan geschieden met behulp van de volgende equate directives:

```

230 PTMCR3 EQU $FF08      write only control register 3
240 PTMCR1 EQU PTMCR3     write only control register 1
250 PTMCR2 EQU PTMCR3+1   write only control register 2
260 PTMHR1 EQU PTMCR3+2   write only MSB buffer register
270 PTMLR1 EQU PTMCR3+3   write only LSB counter latches 1
280 PTMHR2 EQU PTMCR3+4   write only MSB buffer register
290 PTMLR2 EQU PTMCR3+5   write only LSB counter latches 2
300 PTMHR3 EQU PTMCR3+6   write only MSB buffer register
310 PTMLR3 EQU PTMCR3+7   write only LSB counter latches 3
320 PTMSR EQU PTMCR3+1    read only status register
330 PTMHC1 EQU PTMCR3+2   read only MSB counter 1
340 PTMLC1 EQU PTMCR3+3   read only LSB buffer register
350 PTMHC2 EQU PTMCR3+4   read only MSB counter 2
360 PTMLC2 EQU PTMCR3+5   read only LSB buffer register
370 PTMHC3 EQU PTMCR3+6   read only MSB counter 3
380 PTMLC3 EQU PTMCR3+7   read only LSB buffer register

```

Voorbeeld 10.12 (Time-out)

Om te voorkomen dat een handshake-procedure vastloopt ten gevolge van het uitblijven van het antwoordsignaal, maken we gebruik van een time-out. Na het verstrijken van de ingestelde tijd wordt de handshake-procedure afgebroken. Hiertoe wordt de flag HANDTO geset. De time-out wordt ingesteld op 5 ms. Dit zijn 5000 perioden van E. Door het getal 4999 in de latches van

teller 2 te schrijven, wordt teller 2 geïnitieerd en ge-enabled. Het aftellen begint dan vanaf de beginwaarde 4999 en er zal een time-out worden gegenereerd na (N+1) = 5000 perioden van E, tenzij teller 2 voor die tijd opnieuw wordt geïnitieerd. Evenwel moet I2 bij het starten van teller 2 zijn gereset. I2 kan immers zijn geset ten gevolge van een voorafgaande (disabled) time-out. Het resetten van I2 kan geschieden met behulp van het achtereenvolgens lezen van het statusregister en het tellerregister (regel 400 en 410). Teller 2 moet werken in de 16-bits continuous mode (single-shot mag ook) met I2 ge-enabled en O2 gedisable. Teller 2 wordt met de volgende subroutine gestart (waarbij is aangenomen dat I2 is gedisable):

```

400 START2  TST    PTMSR          reset
410          TST    PTMLC2        I2
420          LDD    #4999         N2:=4999, dus time-out is
430          STD    PTMHR2        5000 µs en start teller 2
440          LDA    #01000010     disable O2, enable I2, 16-bits
450          STA    PTMCR2        continuous operating met
460 *                               write-initialisatie, E-klok
470          RTS

```

Bij het wegsturen van een handshake-sigitaal moet teller 2 telkens opnieuw worden geïnitieerd en herstart met behulp van de instructie

```
BSR    START2
```

Hiermee wordt teller 2 opnieuw gestart vanaf 4999 en I2 wordt ge-enabled.

Bij het ontvangen van het antwoordsigitaal moet I2 worden gedisable, opdat de time-out wordt genegeerd. Dit kan geschieden met behulp van de instructies

```

LDA    #00000010
STA    PTMCR2

```

Is de $\overline{\text{IRQ}}$ -uitgang van de PTM verbonden met de $\overline{\text{IRQ}}$ -ingang van de microprocessor, dan kan de IRQ-interrupt service routine er als volgt uit zien:

```

500 IRQISR  LDA    PTMSR          spring als er
510          BPL    NOPTM         geen PTM-interrupt is
520          BITA   #$02          spring als er
530          BEQ    NOTO2         geen time-out 2 is
540          TST    PTMLC2        reset I2
550          LDA    #$80
560          STA    HANDTO        set de flag HANDTO
570          BRA    ENDIRQ
580 NOTO2   --
590         --
600 NOPTM   --
610         --
620 ENDIRQ  RTI

```

Let op dat interrupt flag I2 moet worden gereset. Dit gebeurt door het lezen van teller 2 (regel 540).

Door de flag HANDTO te testen kan worden geconstateerd of de ingestelde tijdlimiet voor het antwoordsignaal is overschreden en of de handshake-procedure moet worden afgebroken.

Voorbeeld 10.13 (Tijdmeting met een PTM en een 2 kHz oscillator)

We willen de tijd meten tussen de eerste opgaande flank van een signaal p en de daaropvolgende neergaande flank van een signaal q (zie fig. 10.61). Het tijdinterval (dat maximaal 250 s kan bedragen) moet worden gemeten in ms.

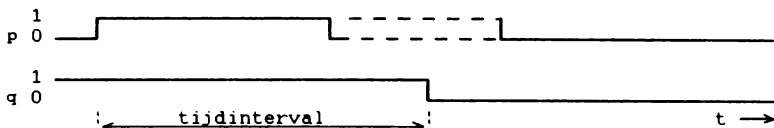


Fig. 10.61 Tijdmeting tussen twee flanken

Dit probleem kan worden opgelost door een teller te starten op de opgaande flank van p en te stoppen op de neergaande flank van q. Aanwezig is een oscillator met een frequentie van 2 kHz. Deze kan worden gebruikt om de tijd af te tellen. Hiermee worden echter intervallen van 0,5 ms afgeteld. Door te delen door 2 vinden we de tijd in ms.

De tijdmeting moet starten op een opgaande flank van het signaal p. Alleen bij de pulse width comparison mode bestaat de mogelijkheid om een interrupt te genereren op een opgaande flank van een signaal op $\overline{G^*}$. De meting start hierbij op een neergaande flank op $\overline{G^*}$. Indien signaal p op het 0-niveau begint, is er geen neergaande flank, voorafgaand aan de positieve flank. In dit geval wordt de meting niet gestart. We moeten derhalve het signaal p geïnverteerd aan de PTM aanbieden, bijvoorbeeld op ingang $\overline{G1}$. Nu kan de meting worden gestart in elke mode van de PTM.

De maximale tijdsduur bedraagt $250\text{ s} = 250.000\text{ ms} = 500.000$ klokperiodes van de 2 kHz oscillator. Dit past niet in de 16-bits tellerregisters van de PTM. Door de registerlatches te laden met \$FFFF en de PTM in te stellen in de continuous of single shot mode, wordt er elke $(N+1) = 2^{16}$ klokperiodes (van 0,5 ms) een interrupt gegenereerd. Hierop kan een (8-bits) teller worden verhoogd, zodat nu kan worden geteld tot maximaal $2^{24} - 1$. Dit is ongeveer 16.000.000, wat overeenkomt met 8.000 s.

In de continuous operating mode stopt het aftellen als $\overline{G^*}$ gelijk aan 1 wordt. In de single shot timer mode wordt dan gewoon doorgeteld. We stellen timer 1 derhalve in deze laatste mode in.

Het stoppen op de eerste neergaande flank van q kan niet onmiddellijk geschieden. Wel kan 1 μ s later een interrupt worden gegenereerd door q aan te sluiten op ingang $\overline{G2}$ van de PTM, latches 2 te laden met 0 en teller 2 af te tellen met de klok E. Er wordt dan 1 μ s na de neergaande flank van q een interrupt gegenereerd, tenzij timer 2 is ingesteld in de time interval mode.

Er moet nu achtereenvolgens het volgende gebeuren:

- Set de interrupt mask bit 1 van de microprocessor om te voorkomen dat eventuele vorige timerinterrupts effect hebben.
- Initialiseer timer 1: CR1 = %0111 0000; latches 1 = \$FFFF. Bedenk hierbij dat teller 1 alleen mag starten ten gevolge van een neergaande flank op $\overline{G1}$ en moet aftellen op de externe 2 kHz klok $\overline{C1}$.
- Initialiseer timer 2: CR2 = %0101 0010; latches 2 = 0. Teller 2 mag alleen starten bij een neergaande flank op $\overline{G2}$ en moet aftellen op E.
- Reset de interrupt mask bit 1.

Bij een interrupt van timer 1 wordt een 8-bits teller verhoogd. Bij de interrupt van timer 2 wordt de gemeten intervalltijd opgeborgen in het geheugen en worden verdere interrupts van timer 1 en timer 2 geblokkeerd.

De initialisatieroutine voor het starten van de tijdmeting ziet er als volgt uit:

100	TIJD	RMB	3	ruimte voor de gemeten tijd.
110	MEET	ORCC	#\$10	blokkeer IRQ-interrupts
120		LDA	#\$53	initialiseer timer 2 en
130		STA	PTMCR2	adresseer CR1
140		LDD	#0	
150		STD	PTMHR2	
160		LDA	#\$70	initialiseer timer 1; CR1 is
170		STA	PTMCR1	adresseerbaar omdat CR20=1!
180		LDD	#\$FFFF	
190		STD	PTMHR1	
200		TST	PTMSR	reset
210		TST	PTMLC2	I2
220		TST	PTMLC1	en I1
230		CLR	TIJD	reset meestwaardige byte van TIJD
240		ANDCC	#\$EF	enable IRQ-interrupts
250		RTS		

De interrupt service routine voor IRQ ziet er als volgt uit:

300	IRQISR	LDA	PTMSR	spring als er een
310		BMI	PTMINT	timerinterrupt is
320		JSR	ANDER	er is een andere interrupt
330		BRA	ENDISR	
340	PTMINT	BITA	#\$01	

350	BEQ	NOINT1	
360	BSR	TIMER1	er is een interrupt van timer 1
370	NOINT1	BITA	#\$02
380	BEQ	NOINT2	
390	BSR	TIMER2	er is een interrupt van timer 2
400	NOINT2	BITA	#\$04
410	BEQ	ENDISR	
420	BSR	TIMER3	er is een interrupt van timer 3
430	ENDISR	RTI	
440	*		
450	TIMER1	TST	PTMLC1
460		INC	TIJD
470		RTS	
480	*		
490	TIMER2	TST	PTMLC2
500		CLR	PTMCR1
510		CLR	PTMCR2
520		LDD	PTMHC1
530		COMA	
540		COMB	
550	*		
560		STD	TIJD+1
570		LSR	TIJD
580		ROR	TIJD+1
590		ROR	TIJD+2
600		RTS	

De timer heeft de hoogste prioriteit van alle IRQ-interrupts. Dit komt doordat bij de polling het eerste gekeken wordt of de timer een interrupt heeft veroorzaakt (regel 300).

Heeft de timer geen interrupt veroorzaakt, dan wordt in de subroutine ANDER nagegaan waar de interrupt vandaan komt. Bovendien wordt deze interrupt in ANDER afgehandeld (regel 320).

Zouden er meerdere interrupts tegelijk zijn opgetreden, dan worden eerst de timerinterrupts afgehandeld. Hierna wordt teruggekeerd naar het hoofdprogramma. Vervolgens wordt onmiddellijk teruggesprongen naar de interrupt service routine, omdat de andere interruptaanvragen nog aanwezig zijn op de IRQ-ingang van de microprocessor.

We moeten na de interrupt van timer 2 in principe zo snel mogelijk stoppen met aftellen, in verband met de nauwkeurigheid van de meting. Toch wordt eerst de interrupt van timer 1 afgehandeld (zie regel 340) en pas daarna interrupt 2. Stel dat de timers 1 en 2 ongeveer tegelijkertijd een interrupt aanvragen. Zou eerst I2 worden afgehandeld, dan wordt de waarde van TIJD gedeeld door twee, teneinde ms te krijgen. Wordt hierna interrupt 1 afgehandeld, dan wordt TIJD met 1 verhoogd. Deze 1 heeft de waarde $2^{16} \cdot 0,5$ ms, maar wordt opgevat als 2^{16} ms, omdat de omrekening naar ms al heeft plaatsgevonden. Vandaar dat interrupt 1 vóór interrupt 2 moet worden afgehandeld.

De regels 400, 410 en 420 zijn toegevoegd voor de afhandeling van een eventuele interrupt van timer 3. Is deze interrupt gedisabled, dan kunnen deze regels vervallen.

De instructies van de regels 450 en 490 resetten I1, respectievelijk I2, omdat het status register al eerder is gelezen (regel 300).

In regel 500 en 510 worden de interrupts I1 en I2 gedisabled. Eigenlijk hoeven alleen de bits 6 van de control registers CR1 en CR2 te worden gereset, maar dit kost meer instructies. De timers 1 en 2 worden niet meer gebruikt, zodat de overige bits niet van belang zijn.

De tijdmeting kan op elk willekeurig moment worden gestart door in het hoofdprogramma de subroutine MEET aan te roepen.

Voorbeeld 10.14 (Het aansturen van meerdere 7-segment displays)

Een 7-segment display heeft 7 ingangen a t/m g en een enable-ingang EN (zie fig. 10.62). Voor EN=0 is het display gedoofd. Is EN=1, dan lichten die LED's op, waarvan de betreffende ingangen 1 zijn. Op deze wijze kunnen met behulp van 7-segment displays de hexadecimale cijfers zichtbaar worden gemaakt (zie fig. 10.63).

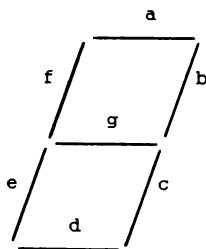


Fig. 10.62 7-segment display

hex. symbool	7-segment aansturing g f e d c b a	hex
0	0 1 1 1 1 1 1	\$3F
1	0 0 0 0 1 1 0	\$06
2	1 0 1 1 0 1 1	\$5B
3	1 0 0 1 1 1 1	\$4F
4	1 1 0 0 1 1 0	\$66
5	1 1 0 1 1 0 1	\$6D
6	1 1 1 1 1 0 1	\$7D
7	0 0 0 0 1 1 1	\$07
8	1 1 1 1 1 1 1	\$7F
9	1 1 0 1 1 1 1	\$6F
A	1 1 1 0 1 1 1	\$77
B	1 1 1 1 1 0 0	\$7C
C	0 1 1 1 0 0 1	\$39
D	1 0 1 1 1 1 0	\$5E
E	1 1 1 1 0 0 1	\$79
F	1 1 1 0 0 0 1	\$71
	0 0 0 0 0 0 0	\$00

Fig. 10.63 Aansturing van een hexadecimaal 7-segment display

De letters B en D worden als kleine letters b en d weergegeven, omdat ze anders niet zijn te onderscheiden van de cijfers 8 en 0. Zijn de ingangen a t/m g gelijk aan 0, dan zijn alle LED's gedoofd.

Willen we acht cijfers weergeven op acht 7-segment displays, dan kan dit gebeuren via vier PIA's. Het kan echter ook via één PIA als we de LED's beurtelings een korte tijd laten oplichten. Hierbij maken we gebruik van de traagheid van het oog, dat de snelle wisselingen van de displays niet kan volgen. Het lijkt dan alsof alle displays tegelijk aan zijn. Dit wordt bereikt door de uitgangen PA0 t/m PA6 te verbinden met de ingangen a t/m g van elk display. De enable-ingangen van de displays worden verbonden met de uitgangen PB0 t/m PB7 (zie fig. 10.64). Door nu beurtelings een van de uitgangen PB_i gelijk aan 1 te maken, worden de displays om beurten aangestuurd. Telkens zijn 7 displays gedoofd. De informatie in PRA van de PIA moet hierbij op de juiste wijze veranderen, zodat een bepaald display telkens van dezelfde informatie wordt voorzien. De lichtopbrengst van de displays is bij deze methode acht keer zo laag als bij het gebruik van vier PIA's.

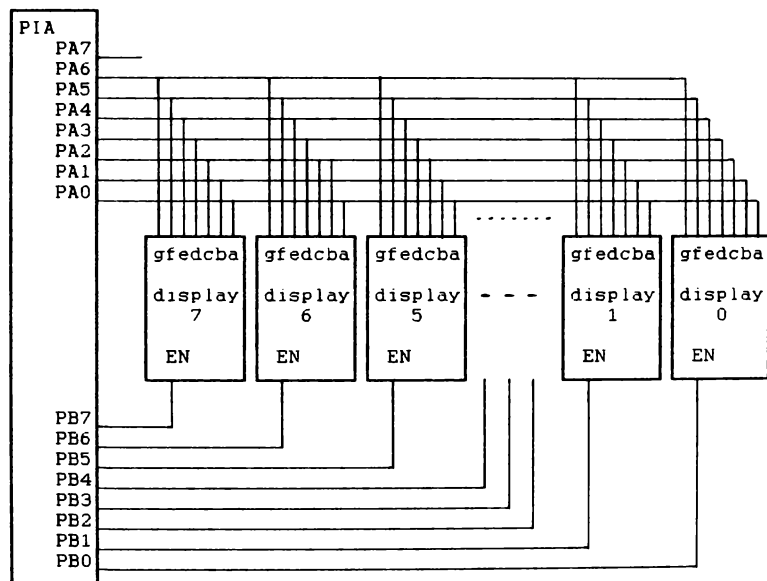


Fig. 10.64 Aansturing van 7-segment displays

Elke display wordt gedurende 1 ms aangestuurd. Dit kan worden bereikt door in het programma telkens 1 ms af te tellen. Echter in dat geval kan de processor niets anders meer doen. Laten we een PTM elke ms een interrupt geven, dan kan de processor in de tussentijd andere zaken afhandelen. Bij elke time-out van de PTM veranderen we de informatie in PRA en sturen we de volgende display aan. Dit laatste kan eenvoudig geschieden door een

enkele 1 in PRB te laten roteren.

De juiste aansturing van de displays zetten we in de tabel STUUR met behulp van de assembler directive FCB.

De acht karakters, die moeten worden gedisplayed, worden in het hoofdprogramma in een tabel KARTAB gezet.

Timer 2 van de PTM wordt ingesteld om elke ms een interrupt te genereren. De overige timers worden vooralsnog niet gebruikt.

Tabel STUUR wordt gemaakt met behulp van de assembler, terwijl er voor de tabel KARTAB en de pointer KARPOI ruimte wordt gecreëerd:

10	ORG	\$3000
20	STUUR	FCB \$3F,\$06,\$5B,\$4F,\$66,\$6D,\$7D,\$07
30	FCB	\$7F,\$6F,\$77,\$7C,\$39,\$5E,\$79,\$71,\$00
40	KARTAB	RMB 8
50	KARPOI	RMB 2

De subroutine PIATIM om PIA en PTM te initialiseren, alsmede de timer te starten, ziet er als volgt uit:

100	PIATIM	ORCC	#\$10	disable IRQ-interrupts
110		CLR	CRA	adresseer DDRA
120		LDA	#\$FF	PAi wordt
130		STA	DDRA	output
140		LDB	#\$04	adresseer PRA en disable CA1 en
150		STB	CRA	CA2 interrupts
160		CLR	CRB	adresseer DDRB
170		STA	DDRB	PBi wordt output
180		STB	CRB	adresseer PRB en disable CB1 en
190	*			CB2 int.
200		CLR	PRA	doof de display
210		LDA	#1	PB0:=1, wijs
220		STA	PRB	de rechter display aan
230		LDA	#1	adresseer
240		STA	PTMCR2	PTMCR1 en
250		STA	PTMCR1	reset de PTM
260		CLR	PTMCR1	hef de reset van de timer op
270		LDA	#\$01000010	disable O2, enable I2,
280		STA	PTMCR2	continuous 16-bits mode,
290	*			initialisatie door schrijven
300	*			in latches. klok E
310		LDD	#999	time-out 2 := 1000 μ s = 1 ms;
320		STD	PTMHR2	start timer 2
330		ANDCC	#\$EF	enable IRQ-interrupts
340		RTS		

De interrupt service routine ziet er als volgt uit:

350	IRQISR	LDA	PTMSR	spring als de PTM
360		BMI	PTMINT	een interrupt geeft
370		JSR	ANDER	

```

380          BRA      ENDINT
390 PTMINT   BITA     #2
400          BEQ      ENDINT          spring als I2=0
410          BSR      NEXDIS          stuur volgende display aan
420 ENDINT   RTI

```

Het structogram van subroutine NEXDIS is gegeven in fig. 10.65.

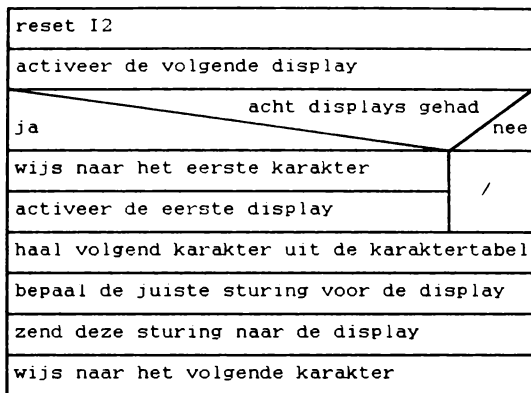


Fig. 10.65 Structogram voor het aansturen van de volgende display

De subroutine NEXDIS ziet er nu als volgt uit:

```

450 NEXDIS   TST      PTMLC2          reset I2
460          LSR      PRB             activeer volgende display
470          BCC      HAALOP          spring als niet 8 displays gehad
480          LDX      #KARTAB         wijs naar
490          STX      KARPOI          eerste karakter
500          LDB      #$80            activeer
510          STB      PRB             eerste display
520 HAALOP   LDX      KARPOI          haal volgend karakter
530          LDB      ,X              uit karakertabel
540          LDU      #STUUR          bepaal de overeenkomstige
550          LDB      B,U             besturing voor de display
560          STB      PRA             zend sturing naar de display
570          INC      KARPOI          wijs naar de volgende display
580          RTS

```

In de regels 540 en 550 mag U worden vervangen door X, omdat de oude waarden van X niet meer nodig zijn.

Het activeren van de volgende display geschiedt door de 1 in PRB één plaats naar rechts te schuiven (regel 460). Wordt hierbij de carry gelijk aan 1, dan hebben we de acht displays

gehad en moet er opnieuw worden begonnen met display 7 (de eerste display) door PRB te laden met \$80 (regel 500 en 510). Ook moet de karakterpointer KARPOI naar het eerste karakter wijzen. Dit wordt bereikt met de instructies in regel 480 en 490.

Door in de initialisatieroutine PIA11M PRB met \$01 te laden, bereiken we dat KARPOI wordt geïnitieerd bij de eerste timerinterrupt.

KARPOI moet een geheugenlocatie zijn, omdat een register aan het einde van de interrupt service routine met de oude waarde wordt overschreven. De parameteroverdracht kan derhalve bij interrupt service routines niet zonder meer via registers geschieden.

Doordat de displaysturingen in de besturingstabel STUUR zijn gerangschikt in binair oplopende volgorde van de karakters, wijst het karakter rechtstreeks de betreffende sturing aan. Zo vinden we de sturing van 0 op locatie STUUR+0, die van 1 op STUUR+1, enzovoorts. De sturing van C vinden we op STUUR+\$C = STUUR+12. Het ophalen van de juiste sturing geschiedt nu eenvoudigweg met de instructies op de regels 540 en 550. Hierbij wijst het indexregister naar de eerste sturing. Het adres van de juiste sturing wordt verkregen door de inhoud van het indexregister met de waarde van het te displayen karakter te verhogen.

Om alle LED's van een bepaalde display te doven, moet aan die display de code \$00 worden aangeboden. In dit geval wordt deze display als het ware overgeslagen. Er wordt geen karakter getoond op deze display. De code \$00 staat in de tabel STUUR op 16 locaties na de eerste code. Derhalve moeten we deze code aangeven met 16 = \$10. Zodoende kan de bijbehorende sturing op dezelfde wijze worden aangewezen als de overige sturingen.

In principe kunnen op de hier beschreven wijze ook ASCII-karakters zichtbaar worden gemaakt. De waarde van het ASCII-karakter kan worden gebruikt als offset om in een tabel de bijbehorende sturing aan te wijzen. Om alle ASCII-karakters zichtbaar te maken, is een 7-segment display te beperkt. Er is dan een display nodig dat uit meer LED's bestaat, bijvoorbeeld uit een 5x7 LED-matrix. De besturing hiervan is aanzienlijk complexer. Hiervoor wordt een zogenaamde character generator gebruikt, waarop we niet verder zullen ingaan.

Voorbeeld 10.15 (Lichtkrant)

Willen we meer karakters laten zien dan het beschikbare aantal displays, dan kan dit worden gerealiseerd door het principe van de lichtkrant toe te passen. Hierbij loopt de boodschap van rechts naar links, waarbij telkens links een karakter verdwijnt. Rechts wordt het volgende karakter van de boodschap toegevoegd. Zo wordt de boodschap als het ware door een venster gelezen, waarbij het venster naar rechts over de boodschap lijkt te schuiven (zie fig. 10.66).

```

      V
    Ve
  Vee

  :
  :
  :
  Veel suc
  eel succ
  el succe
  l succes
  succes!
  succes!
  ucces!

  :
  :
  :
  es!
  s!
  !

```

Fig. 10.66 Het principe van de lichtkrant

Staat de gehele boodschap in het geheugen vanaf het adres KARTAB, dan kan het principe van de lichtkrant worden gerealiseerd met een paar wijzigingen van het programma van voorbeeld 10.14. Hierbij moet de pointer KARPOI niet telkens worden geïnitialiseerd met de waarde van KARTAB. KARPOI moet bijvoorbeeld om de 0,3 s naar het volgende karakter in de karakertabel wijzen. Zo zien we elke 0,3 s het zichtbare gedeelte van de boodschap naar links schuiven. Hierbij verdwijnt het linker karakter en komt er rechts een bij.

Om de boodschap letter voor letter te laten beginnen, vullen we de zeven eerste locaties van de karakertabel met \$10. Daarna komt de boodschap, die bijvoorbeeld wordt afgesloten met acht keer \$10, gevolgd door een carriage return (\$0D). De acht "blanks" aan het einde zorgen ervoor dat de boodschap naar links geheel verdwijnt, voordat er opnieuw wordt begonnen. De boodschap wordt telkens weer vanaf het begin verzonden door de pointer opnieuw te initialiseren bij het detecteren van \$0D. Pointer BEGIN bevat het adres van het karakter dat in de linker display komt.

Timer 3 van de PTM stellen we in om elke 0,3 s een interrupt te genereren. Hiervoor kan de interne klok E (1 MHz) worden gebruikt, door E te delen door 8. De latches 3 moeten dan worden geladen met 37.500 om elke 0,3 s een interrupt te krijgen.

In het programma van voorbeeld 10.14 moeten de volgende regels worden tussengevoegd of gewijzigd:

60	BEGIN	RMB	2	adres 1ste te tonen karakter
322		LDA	##01000011	disable O3, enable I3, continuous
323		STA	PTMCR3	16-bits mode, initialisatie door
324 *				schrijven in latches, klok E/8
325		LDD	##37499	time-out 3 := 37.500·8 μs = 0,3 s;

326	STD	PTMHR3	start timer 3
327	LDX	#KARTAB	wijs naar de allereerste locatie
328	STX	BEGIN	van de karaktertabel

Verder moet een routine toegevoegd worden voor de time-out van teller 3. Hierbij moet het venster van de boodschap één positie verschuiven. Tevens moet bij detectie van \$0D (carriage return) opnieuw worden begonnen bij het begin van de karaktertabel.

400	BEQ	NO2	spring als I2=0
414	BITA	#4	
415	BEQ	ENDINT	spring als er geen I3 is
416	BSR	NEWKAR	verschuif venster van boodschap
480	LDX	BEGIN	
533	CMPB	#\$0D	spring als er
534	BNE	DISPL	geen carriage return is
535	LDX	#KARTAB	wijs naar eerste karakter van
536	STX	BEGIN	karaktertabel; begin opnieuw
537	LDB	#\$10	rechter display blijft gedooft
540	DISPL	LDU	#STUUR
600	NEWKAR	TST	PTMLC3
610	LDX	BEGIN	reset I3
620	LEAX	1,X	verschuif
630	STX	BEGIN	het venster
640	RTS		één positie

Indien het te displayen karakter de carriage return <CR> blijkt te zijn, wijst PRB naar de meest rechter display, immers daar wordt een karakter het eerst getoond. Er moet nu echter niet worden gedisplaysd vandaar dat B wordt geladen met een "blank", zodat de betreffende display gedooft blijft (regel 537).

Hierna moet het beginadres worden geïnitieerd (regel 535 en 536). Bij de volgende interrupt I2 zal in regel 460 de carry geset worden, omdat (PRB) gelijk was aan \$01. Hierdoor wordt de pointer geïnitieerd, zodat de boodschap vanaf het begin zal worden getoond.

10.13 Direct Memory Access

Programmed I/O, interrupt-I/O en Direct Memory Access

In tegenstelling tot bij de overige methoden van datatransport wordt er bij Direct Memory Access (DMA) niet gecommuniceerd via de processor. Het datatransport vindt rechtstreeks plaats tussen geheugen en input/output device (zij het meestal via een I/O-controller). Omdat de microprocessor hieraan niet te pas komt, geschiedt het datatransport zonder instructies. Zoals in

deze paragraaf zal blijken, biedt DMA de snelste methode van datatransport.

Bij programmed I/O kunnen grote wachttijden ontstaan ten gevolge van de traagheid van de periferie. In het programma moet worden nagegaan of het randapparaat de opdracht heeft uitgevoerd. Er kan worden gewacht totdat dit is gebeurd. Dan kan de processor echter geen andere acties meer ondernemen en gaat er veel tijd verloren. Wordt het programma voortgezet, dan zal er regelmatig moeten worden gecontroleerd of het randapparaat nog bezig is met het uitvoeren van de opdracht. Dit verstoort de normale flow van het programma.

Bij interrupt-I/O meldt het randapparaat zelf dat het klaar is, zodat de processor in de tussentijd het programma kan vervolgen. De interrupt van het I/O-device onderbreekt het programma, waarbij in de interrupt service routine de melding van het device wordt afgehandeld. Er zijn hierbij in principe geen wachttijden ten gevolge van de traagheid van de periferie. Ook wordt de flow van het programma niet verstoord ten gevolge van tussengevoegde instructies voor het bekijken van de toestand van het randapparaat.

Moeten er vele gegevens worden uitgewisseld, dan is het werken op interruptbasis niet erg efficiënt. Dit wordt veroorzaakt door de grote overhead, waarbij veel tijdverlies optreedt. Deze overhead bestaat uit het volgende:

- De processor moet eerst de lopende instructie afmaken, voordat de interruptaanvraag beantwoord kan worden.
- Een aantal registerinhouden moet worden gered.
- Er wordt gesprongen naar de bijbehorende interrupt service routine.
- Er moet worden nagegaan wie de interruptaanvrager is (met behulp van polling of vectoring).
- De interrupt wordt afgehandeld. Hierbij vindt het datatransport tussen periferie en geheugen via de processor plaats. Tevens moet er aan de aanvrager worden teruggemeld dat de interrupt wordt afgehandeld.
- De registerinhouden worden hersteld en er wordt teruggesprongen naar het onderbroken programma.

I/O op basis van DMA heeft slechts zeer weinig overhead. Derhalve is DMA zeer snel.

Het werkingsprincipe van DMA

Het datatransport tussen periferie en geheugen gaat via de databus. Hierbij is een adres nodig om aan te geven om welke geheugenlocatie het gaat. Het adres wordt via de adresbus aangeboden aan het geheugen. Tevens is het signaal $R/\bar{W}$ (in de controlbus) nodig om de richting van het transport aan te geven. Daar de processor zich niet bemoeit met het datatransport tijdens DMA, zullen geheugenadres en $R/\bar{W}$ -signaal van elders

moeten komen. Hiervoor is externe hardware nodig. Deze hardware is in de vorm van een IC beschikbaar en wordt DMA-Controller (DMAC) genoemd.

Er treedt nu een extra complicatie op. Er mag immers slechts één zender actief zijn op een lijn. Zou er door meerdere zenders tegelijk een 0 en een 1 op de lijn worden gezet, dan ontstaat een ongedefinieerd niveau, terwijl er bovendien zeer grote stromen kunnen gaan lopen (zie ook paragraaf 10.2). De DMAC mag derhalve pas een adres op de adresbus zetten als de microprocessor van deze bus is losgekoppeld. Geheugen of randapparaat mogen pas data verzenden, nadat de microprocessor zijn zenders van de databus heeft afgeschakeld. Samenvattend kunnen we zeggen dat de microprocessor eerst zijn $R/\bar{W}$ -, adres- en datazenders moet loskoppelen van de bus, alvorens DMA mag plaatsvinden. Gebruikt de microprocessor de buslijnen, dan moet de DMAC zijn zenders van $R/\bar{W}$ - en adreslijnen afschakelen, terwijl de I/O-Controller geen DMA-datatransfer mag doen.

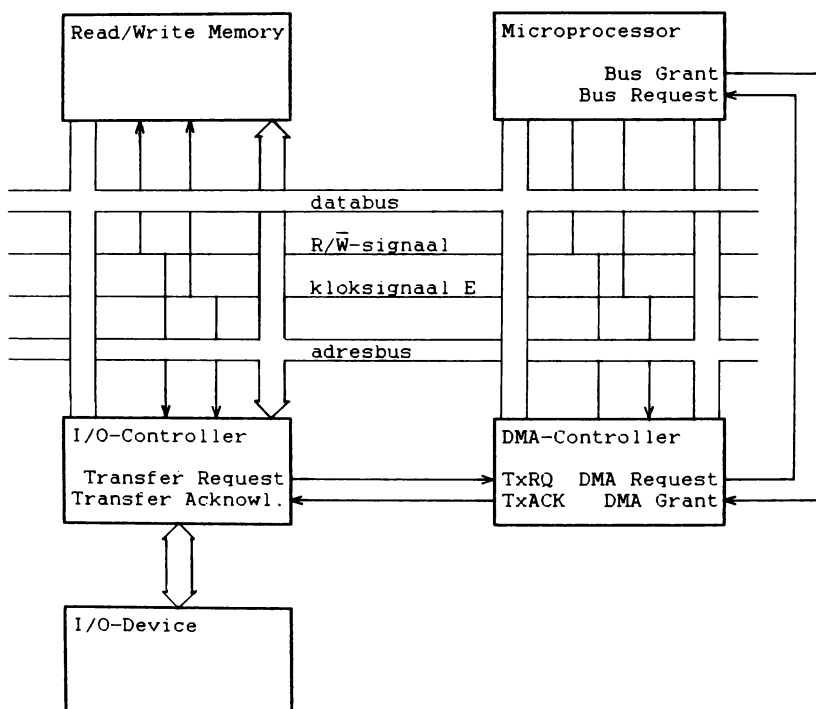


Fig. 10.67 I/O met behulp van DMA

Fig. 10.67 toont een configuratie waarin DMA kan plaatsvinden.

Normaal gesproken beheerst de microprocessor de bus en is derhalve busmaster.

1. Om DMA te plegen, moet de DMAC aan de microprocessor toestemming vragen om de bus te gebruiken. Deze aanvraag geschiedt door het activeren van het signaal DMA Request. De microprocessor ontvangt dit signaal op zijn ingang Bus Request en zal van de bus afschakelen. Hij meldt dit door het signaal Bus Grant te activeren (to grant = toestaan, schenken).
2. Dit wordt door de DMAC ontvangen op zijn DMA Grant input. De DMAC is nu busmaster geworden en zal de bussignalen zodanig besturen dat er data-overdracht kan plaatsvinden tussen de I/O-controller en het geheugen.
3. Na het datatransport zal de DMAC de bus weer vrijgeven door het signaal DMA Request 0 te maken.
4. De microprocessor reset Bus Grant en neemt de besturing van de bus over.

U ziet dat tussen processor en DMAC een handshake plaatsvindt om het beheer van de bus aan elkaar over te dragen (zie ook paragraaf 10.5.4).

Het initiatief bij DMA ligt altijd bij het I/O-device.

1. Zodra het I/O-device gereed is voor data-uitwisseling met het geheugen, vraagt het device het gebruik van de bus aan door het signaal Transfer Request te activeren. Meestal gebeurt dit via een I/O-Controller. Natuurlijk zal in het begin de opdracht voor de gegevensuitwisseling door de microprocessor worden gegeven. Het transport zelf wordt daarna steeds geïnitieerd door het I/O-device.
2. Na het ontvangen van een transfer request van het I/O-device vraagt de DMAC toestemming aan de microprocessor om de bus te mogen gebruiken. Na verkregen toestemming activeert de DMAC het signaal Transfer Acknowledge. Nu kan de data-uitwisseling tussen I/O-device en geheugen beginnen. De DMAC levert tijdens het transport van elke byte het juiste geheugenadres. Bovendien houdt hij bij hoeveel bytes er nog moeten worden overgedragen. Hiertoe moet de DMAC van tevoren door de processor zijn geïntialiseerd. Als alle gegevens zijn getransporteerd, kan de DMAC dit aan de processor melden door middel van een interrupt. Hierop voert de processor een zogenaamde post-DMA interrupt service routine uit.
3. Indien het I/O-device niet gereed is voor de uitwisseling van een volgende byte, reset hij het signaal Transfer Request en schakelt zijn buszenders af. De DMAC reset hierop DMA Request en Transfer Acknowledge en schakelt zijn buszenders uit.
4. De microprocessor reset Bus Grant en neemt de besturing van de bus weer over.

U ziet dat er ook een handshake optreedt tussen I/O-device of

I/O-Controller en DMA-Controller.

Het kloksignaal E zorgt voor de synchronisatie van de communicatie.

De hele gang van zaken wordt weergegeven in het structogram van fig. 10.68. De processor zal eerst de DMAC initialiseren om te voorkomen dat het I/O-device al data wil wegsturen of binnenhalen, voordat de DMAC weet om welke geheugenadressen het gaat.

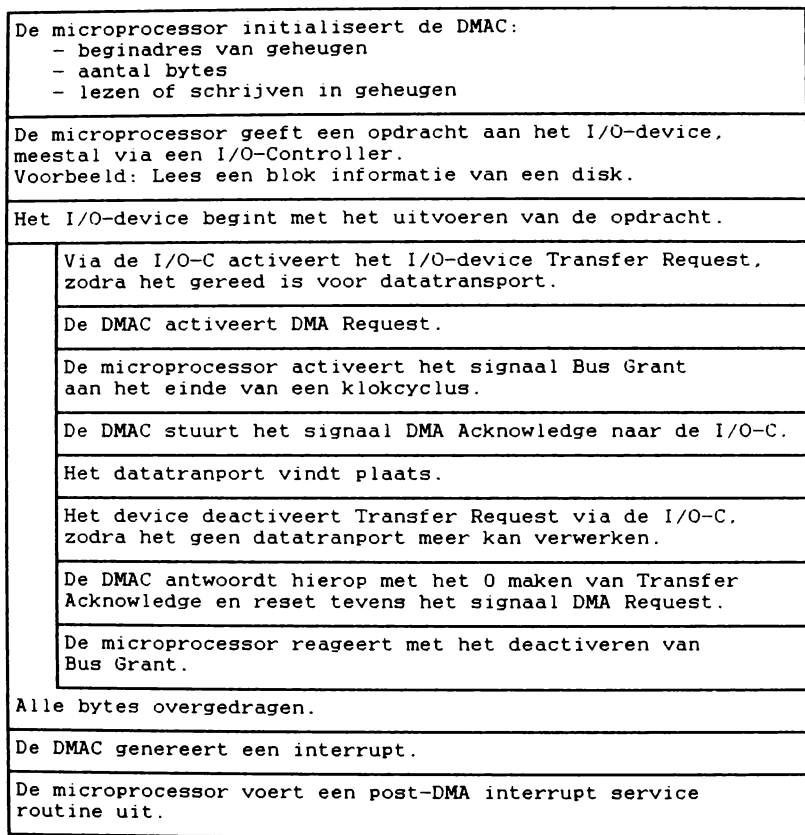


Fig. 10.68 Acties bij DMA

Overhead bij DMA

De indruk zou kunnen ontstaan dat de overhead bij DMA erg groot is. Derhalve worden hier enkele kanttekeningen geplaatst.

- Het geven van een opdracht aan het I/O-device is geen overhead, omdat dit altijd moet gebeuren, ongeacht of de

- communicatie verloopt op basis van programmed I/O, interrupt-I/O of DMA.
- De handshakes die optreden tussen I/O-device en DMAC (Transfer Request - Transfer Acknowledge) en tussen DMAC en processor (DMA Request - DMA Grant), kosten geen tijd. Deze handshakes vinden plaats via aparte signaallijnen, zodat de processor intussen door kan gaan met gebruiken van de bus voor het uitvoeren van instructies.
 - Datatransport van of naar het geheugen speelt zich in principe af in 1 klokcyclus. Bij DMA echter zijn er in totaal 3 klokcycli nodig, indien er telkens slechts één byte wordt overgedragen. Dit wordt veroorzaakt door het tussenvoegen van een zogenaamde dode cyclus voor en na het datatransport. Tijdens deze cyclus wordt de besturing van de bus overgedragen tussen processor en DMAC (zie ook fig. 10.69). Op de bus zelf vindt er geen activiteit plaats tijdens een dode cyclus. De huidige busmaster schakelt zijn zenders af, zodat de nieuwe busmaster zijn zenders tijdens de volgende klokcyclus kan inschakelen. Op deze wijze wordt voorkomen dat er twee zenders tegelijk actief zijn op de bus. Is de processor busmaster, dan spreekt men van MPU-mode. In de DMA-mode is de DMAC busmaster. Blijft de DMAC gedurende langere aaneengesloten tijd busmaster, zodat er meerdere bytes na elkaar middels DMA kunnen worden getransporteerd, dan kan er per klokcyclus één byte worden overgedragen. De dode cycli treden alleen op bij de overdracht van de busbesturing, dus bij het begin en bij het einde van de DMA-mode.
 - Bij programmed I/O en interrupt-I/O daarentegen zijn er veel meer klokcycli nodig voor het datatransport. Er zijn immers een load- en een store-instructie nodig. Zowel een LDA-extended en een STA-extended instructie kost 5 cycli: 3 voor het ophalen van de instructie zelf, 1 voor interne bewerkingen in de processor (het plaatsen van het I/O-adres in het adresregister) en 1 voor het transport van de data (tussen geheugen en processorregister). Dit betekent dus totaal 10 klokcycli per byte in plaats van 1 tot 3 bij DMA.
 - De overhead die optreedt bij DMA, bestaat uit het initialiseren van de DMAC en uit de post-DMA interrupt service routine. Deze overhead komt slechts één keer voor tijdens het transport van een aantal bytes en niet, zoals bij interrupt-I/O, bij elke byte. U kunt hieruit concluderen dat DMA alleen zinvol is als het gaat om grotere aantallen bytes. Anders gaat de winst van 7 tot 9 klokcycli per byte verloren ten gevolge van de extra overhead die optreedt in de vorm van het initialiseren van de DMA-Controller.

De microprocessor hoeft de lopende instructie niet eerst af te maken alvorens de besturing van de bussen over te dragen aan de DMAC. Wel moet de processor wachten tot het einde van een klokcyclus. Vervolgens schakelt de processor zichzelf van de

bussen af en gaat wachten totdat hij de besturing over de bussen terugkrijgt. In de tussentijd gebeurt er binnen in de processor niets, dat wil zeggen alle registers behouden hun gegevens, dus bijvoorbeeld ook het adresregister en het instructieregister. Omdat dit alles ongewijzigd blijft kan de processor later de draad weer oppakken en de instructie afmaken.

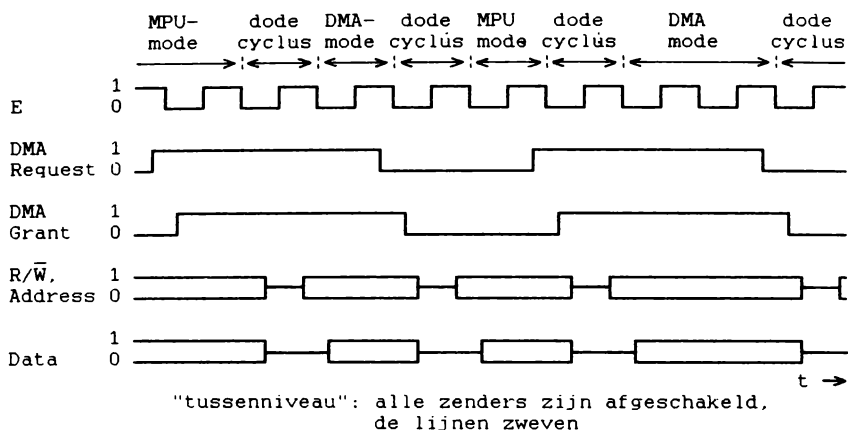


Fig. 10.69 Tijddiagram bij DMA

De DMA-mode kan ook worden afgebroken op initiatief van de processor. Hiertoe reset de processor het signaal Bus Grant, waarop de DMAC reageert met het resetten van Transfer Acknowledge en met het afschakelen van zijn buszenders.

Het actief zijn van de Bus Request ingang van de MC6809-microprocessor heeft de allerhoogste prioriteit. Zelfs tijdens een hardware reset van de MC6809 vindt DMA gewoon doorgang. DMA kan niet door een hardware reset van de MC6809 worden onderbroken.

In fig. 10.69 is een soort tussenniveau aangegeven. Dit is een ongedefinieerd niveau, dat ontstaat in het geval dat geen enkele zender actief is op de buslijn. Zowel de microprocessor als de DMAC en de I/O-Controller hebben hun zenders afgeschakeld. Hun uitgangen zijn dan hoogimpedant. Zenders die in deze hoogimpedante toestand kunnen staan, worden tri-state of three-state circuits genoemd. Ze kennen drie verschillende toestanden, de logische 0-toestand, de logische 1-toestand en de hoogimpedante toestand. In de laatste toestand zijn ze inactief. Ze zijn dan als het ware losgekoppeld van de lijn.

Er zijn diverse soorten DMA te onderscheiden:

1. cycle-stealing DMA
 - single byte mode
 - burst mode
2. simultaneous DMA

Cycle stealing DMA

Bij cycle-stealing DMA steelt de DMAC een klokcyclus van de microprocessor ten behoeve van de overdracht van een databyte op basis van DMA. Hierbij wordt de microprocessor even stilgezet. De executie van het lopende programma wordt hierdoor trager. Er wordt ten behoeve van de overdracht van elke byte één klokcyclus gestolen (afgezien van de dode cycli).

De DMAC kan op twee manieren te werk gaan bij cycle-stealing.

1. single byte mode

De DMAC kan na elke byte de besturing van de bus teruggeven aan de microprocessor.

Het nadeel van de single byte mode is dat de gegevensstroom bij zeer snelle randapparaten niet tijdig verwerkt kan worden. Er kunnen weliswaar meerdere klokcycli aaneengesloten gestolen worden. Kan het device echter niet tijdig een volgende byte uitwisselen, dan wordt de besturing van de bus aan de microprocessor teruggegeven. Hierbij gaan er tenminste twee klokcycli verloren, namelijk de twee dode cycli ten behoeve van het overdragen van de busbesturing van de DMAC aan de processor en omgekeerd. Eventueel komen hier nog een of meerdere MPU-cycli bij.

2. burst mode

De DMAC geeft de besturing pas terug nadat alle over te dragen bytes getransporteerd zijn. Vandaar dat men hier ook wel spreekt van block transfer mode.

In de burst mode wordt de microprocessor gedurende lange tijd aaneengesloten stilgezet. Dit kan onacceptabel zijn. Bedenk bijvoorbeeld dat zelfs een interrupt met een hoge prioriteit (zoals de reset en non-maskable interrupt) gedurende DMA is geblokkeerd. Ook een DMA-aanvraag van een device met hogere prioriteit moet wachten totdat alle bytes overgedragen zijn, zodat er problemen kunnen optreden met de doorvoer van data (throughput error).

Het voordeel van de burst mode is dat de data transfer zeer snel is, namelijk 1 byte per klokcyclus. Alleen aan het begin en aan het einde van de block transfer komt een dode cyclus voor.

Simultaneous DMA

Simultaneous DMA werkt volgens een heel ander principe. Hierbij vindt de DMA plaats gedurende de tijd dat de microprocessor de bus niet gebruikt.

Er kunnen twee werkwijzen worden onderscheiden.

a. Simultaneous DMA tijdens de klokcyclus waarin de processor de bus niet gebruikt

Deze methode gebruikt een gehele klokcyclus voor DMA en wel op een moment dat de microprocessor de bus niet gebruikt. Bij het uitvoeren van een willekeurige instructie wordt tenminste een klokcyclus de bus niet gebruikt. De processor is dan intern bezig met datatransport of met rekenwerk.

Bij een load-extended instructie moet het adres van de data (na het ophalen van de instructie) in het adresregister van de microprocessor worden geplaatst. Hiervoor is een klokcyclus nodig. Tijdens deze cyclus gebruikt de microprocessor de bus niet. Een MUL-instructie duurt totaal elf klokcycli (zie appendix B). De eerste is nodig om de (1-byte) instructie op te halen uit het geheugen. De overige tien worden gebruikt om de berekening uit te voeren en het resultaat in accumulator D op te bergen. Tijdens deze tien klokcycli is de bus in principe vrij voor gebruik door derden.

Burst mode is hierbij niet mogelijk, omdat DMA alleen plaats vindt gedurende de tijd dat de processor de bus niet nodig heeft. DMA kan derhalve slechts gedurende een of enkele aaneengesloten klokcycli plaatsvinden, zodat niet alle bytes achter elkaar overgezonden kunnen worden.

De microprocessor geeft met behulp van signalen aan dat hij de bus niet gebruikt. Hierop wordt teruggekomen in hoofdstuk 11. De DMAC kan deze klokcycli gebruiken voor het laten uitvoeren van DMA. De microprocessor wordt hierbij niet stilgezet, maar gaat gewoon door met het uitvoeren van de opgehaalde instructie. De executie van het programma wordt derhalve niet vertraagd. Het lijkt alsof de DMA en het uitvoeren van het programma tegelijk plaatsvinden. Derhalve wordt hier van simultaneous DMA gesproken. De microprocessor merkt van de DMA helemaal niets. Vandaar ook dat men deze methode van DMA ook wel hidden DMA noemt. De DMA blijft voor de processor immers verborgen.

Op deze wijze kan zeer efficiënt data worden overgedragen. Er gaat immers geen microprocestortijd verloren met het wachten tijdens DMA.

Een nadeel is evenwel de extra hardware die hierbij nodig is. De microprocessor schakelt zijn zenders niet van de bus af. Er wordt immers geen bus request gedaan, omdat de DMA zich buiten het gezichtsveld van de microprocessor afspeelt. Dit betekent dat de $R/\bar{W}$ -, adres- en datalijnen van de microprocessor via buffers van de bus moeten worden gescheiden. Deze buffers (tri-state poortschakelingen) moeten door externe besturingslogica op de juiste momenten in de hoogimpedante toestand worden gezet. Tevens moet deze extra hardware op de juiste momenten de DMAC voorzien van een DMA Grant signaal.

Een ander nadeel is dat het vrij lang kan duren voordat er een

DMA transfer kan worden uitgevoerd. Er moet steeds worden gewacht totdat de microprocessor de bus niet nodig heeft. Worden bijvoorbeeld alle registerinhouden op de stack gezet met behulp van een push-instructie, dan gebruikt de microprocessor de bus gedurende 12 aaneengesloten klokcycli. Snelle devices krijgen in dit geval throughput errors.

Het is ook mogelijk een combinatie van cycle-stealing DMA en simultaneous DMA te kiezen.

b. Simultaneous DMA tijdens een deel van de klokcyclus

Deze methode maakt gebruik van het feit dat de microprocessor slechts gedurende een deel van de klokcyclus de externe bus nodig heeft. Tijdens het eerste gedeelte van een klokcyclus is de microprocessor bezig met het gereed maken van het adres en eventueel de data voor het transport over de bus. Daarna vindt de eigenlijke data-overdracht plaats. Nu kan in principe het eerste gedeelte van elke klokcyclus DMA gepleegd worden. Dit betekent dat hier wel burst mode geoorloofd is.

Ook hier is vanzelfsprekend externe besturingslogica nodig om precies de momenten aan te geven waarop de externe microprocessorbuffers gedisabled moeten zijn en wanneer de DMA mag plaatsvinden. Bovendien is hiervoor een zeer snel geheugen en een zeer snelle I/O-C nodig. Zowel geheugen als I/O-C moeten nu binnen een klokcyclus twee keer een data-uitwisseling kunnen uitvoeren, één keer met de microprocessor en één keer ten behoeve van DMA. Dit betekent dat de vertraging van de signalen op de bus erg klein moet zijn en dat de externe besturingslogica zeer snel moet zijn.

Bij simultaneous DMA worden weliswaar klokcycli (of gedeelten daarvan) gebruikt voor het datatransport, maar dit gaat niet ten koste van de microprocessor. Derhalve is hier van cycle-stealing geen sprake.

Van de klokcyclus kan hier geen sprake zijn, omdat in elke cyclus een busoverdracht plaats moet kunnen vinden. De externe hardware moet derhalve zorgen dat er per klokcyclus twee dode tijden worden gerealiseerd voor het overdragen van de busbesturing (van processor naar DMAC en omgekeerd). Het is immers van tevoren niet bekend of de processor meer dan één klokcyclus de bus niet gebruikt.

Bij simultaneous DMA aan het begin van een klokcyclus moet de busbesturing altijd worden overgedragen aan de processor, omdat deze de rest van de klokcyclus gebruik maakt van de bus.

Samenvattend kunnen we zeggen:

Single byte mode cycle stealing DMA:

- Voor zeer snelle devices mogelijk te langzaam.
- Devices met hogere prioriteit kunnen niet worden geblokkeerd gedurende langere tijd.

Burst mode cycle stealing DMA:

- Zeer snel, geen throughput problemen voor devices.
- De bus wordt voor langere tijd voor andere devices geblokkeerd.

Simultaneous DMA:

- Vertraagt het lopende programma niet.
- Kost externe hardware.
- Snelle componenten nodig.
- Zeer snelle reponsie, indien DMA in deel van klokcyclus plaatsvindt.

10.14 Direct Memory Access Controller

10.14.1 Algemeen

Een DMA-Controller moet tenminste de volgende taken uitvoeren:

1. Hij moet de bus aanvragen aan de processor.
2. Hij moet aan de I/O-Controller meedelen wanneer de DMA transfer mag plaatsvinden.
3. Tijdens DMA-mode moet hij zorgen voor het geheugenadres.
4. Bovendien moet hij een R/ $\bar{W}$ -signaal genereren om aan te geven of er tijdens DMA in het geheugen moet worden gelezen dan wel geschreven.
5. Hij moet bijhouden hoeveel bytes er nog overgedragen moeten worden.
6. Als alle bytes overgedragen zijn, moet hij dit melden aan de processor.

De DMAC moet zijn uitgerust met een adresregister. Nadat een byte is overgedragen, wordt de inhoud van dit register met 1 verhoogd (of verlaagd). Zodoende is bij het volgende transport het nieuwe geheugenadres beschikbaar.

Verder is er een controlregister nodig van waaruit het R/ $\bar{W}$ -signaal geleverd wordt.

Het bijhouden hoeveel bytes er nog overgedragen moeten worden, gebeurt in een zogenaamd byte count register. De inhoud hiervan wordt telkens met 1 verlaagd bij het transporteren van een byte. Is de inhoud 0 geworden, dan zijn alle bytes overgedragen. Nu wordt in het statusregister de zogenaamde DMA-END bit geset. Door deze bit te testen, kan de microprocessor nagaan of alle bytes overgedragen zijn. Is dit het geval dan kan de microprocessor een post-DMA service routine uitvoeren.

De microprocessor moet het adresregister, het byte count register en het control register van de DMAC van beginwaarden voorzien.

Een DMAC kan naast de genoemde noodzakelijke functies een aantal andere taken uitvoeren. Bedient de DMAC meerdere I/O-Controllers, dan is er een mechanisme nodig om de prioriteit van deze I/O-C's vast te stellen. Eventueel kunnen DMA-aanvragen geblokkeerd worden door de DMAC.

De DMA-END bit kan eventueel als interrupt worden doorgegeven aan de microprocessor. Zodoende kan de post-DMA service routine worden afgehandeld op basis van programmed I/O of op basis van interrupt-I/O.

Indien de DMAC in meerdere DMA-modes kan werken, moet dit in het controlregister kunnen worden vastgelegd.

10.14.2 De opbouw van de DMAC

De MC6844-DMAC kan vier I/O-Controllers bedienen. Voor elke I/O-C is er een Address Register, een Byte Count Register en een controlregister nodig. Elke I/O-C moet immers een verschillend aantal bytes kunnen uitwisselen met een bepaald geheugengebied. In het zogenaamde Channel Control Register ligt vast of de betreffende I/O-C leest dan wel schrijft in het geheugen en welke DMA-mode hierbij gebruikt wordt. Ook wordt in dit register aangegeven of alle bytes van of naar de corresponderende I/O-C zijn overgedragen (DMA-END bit).

In het gemeenschappelijke Priority Control Register wordt de prioriteit van de vier I/O-C's bepaald. Tevens wordt hierin vastgelegd of de DMA-aanvraag van een I/O-C wordt afgehandeld. In het gemeenschappelijke Interrupt Control Register wordt vastgelegd welke I/O-C een interrupt mag veroorzaken. Tenslotte is er nog een gemeenschappelijk Data Chain Register. Op de functie hiervan wordt later teruggekomen. In elk register kan worden gelezen en geschreven.

Fig. 10.70 toont de opbouw van de MC6844.

De vier Address Registers en de vier Byte Count Registers bevatten elk 16 bits. De zeven overige registers zijn 8 bits breed. Voor de communicatie met de 15 registers van de DMAC via de 8-bits databus zijn 23 verschillende adressen nodig, immers de 15 registers bevatten samen 23 bytes. Deze bytes worden geselecteerd met behulp van de adresingangen A0 t/m A4. In de DMA-mode zijn alle DMAC-aansluitingen A0 t/m A15 uitgangen. De DMAC zet nu het geheugenadres op de adresbus. A0 t/m A4 functioneren tijdens MPU-mode als register select ingang en tijdens DMA-mode als adresuitgang.

$R/\bar{W}$ is tijdens MPU-mode ingang. De microprocessor deelt via deze ingang aan de DMAC mede of er een bepaald register moet worden gelezen of dat er in moet worden geschreven. Is $R/\bar{W}$ tijdens MPU-mode gelijk aan 1, dan wordt er in de DMAC gelezen.

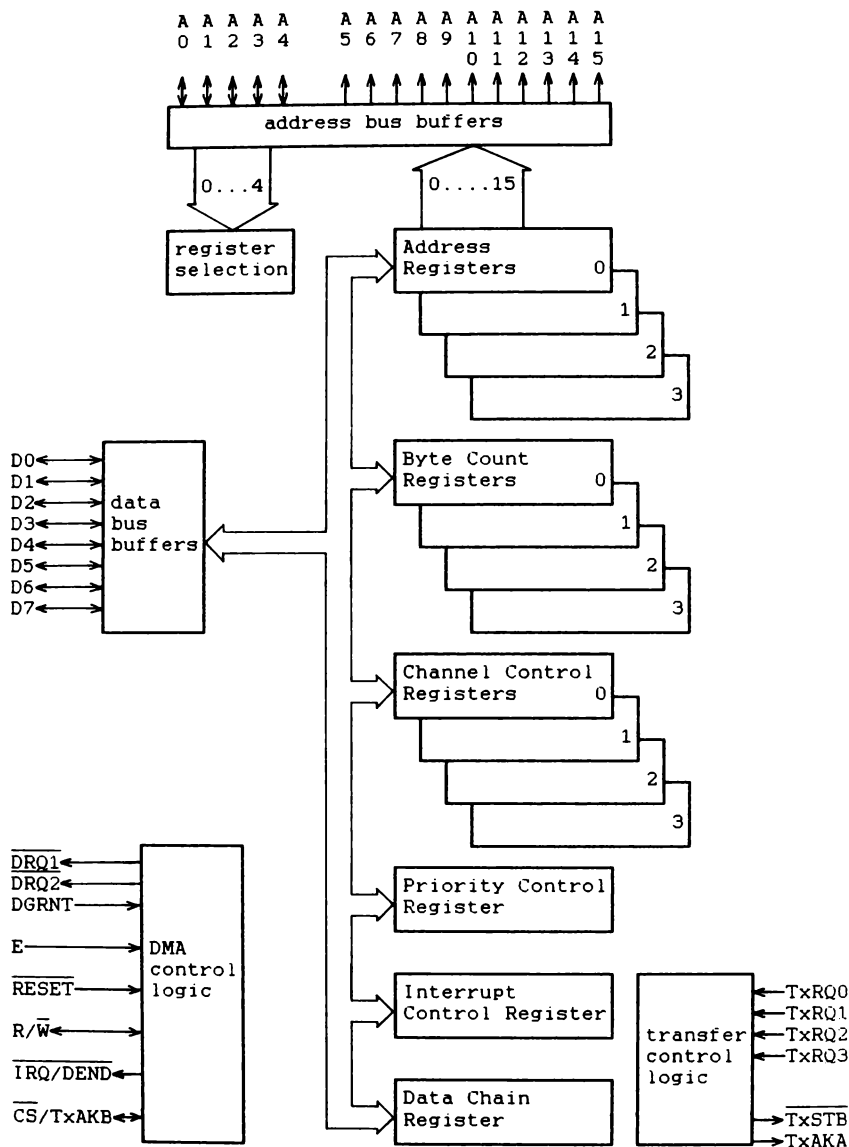


Fig. 10.70 Blokschema van de DMAC

Tijdens DMA-mode is $R/\overline{W}$ daarentegen uitgang, omdat de DMAC middels dit signaal aan het geheugen de richting van het datatransport moet meedelen. Een 1 geeft aan dat er in het geheugen wordt gelezen.

De ingangen TxRQ0 t/m TxRQ3 zijn de Transfer ReQuest signalen van de I/O-Controllers. Zodra een van deze ingangen van de DMAC 1 wordt, activeert de DMAC zijn uitgang $\overline{DRQ1}$ of $\overline{DRQ2}$ (DMA ReQuest), afhankelijk van de ingestelde DMA-mode. Wordt hierop door de microprocessor de DMAC-ingang DGRNT (DMA GRANT) 1 gemaakt, dan geeft de DMAC een Transfer AcKnowledge door aan de transferaanvrager met de hoogste prioriteit. Met behulp van de signalen TxAKA en TxAKB wordt aangegeven welke van de vier I/O-Controllers een datatransfer mag doen (zie fig. 10.71). Hierbij geeft TxSTB (Transfer STroBe) aan, wanneer de signalen TxAKA en TxAKB geldig zijn.

TxAKB TxAKA		transfer toegestaan van device
0	0	0
0	1	1
1	0	2
1	1	3

Fig. 10.71 Transfer Acknowledge

Zijn alle bytes van een peripheral overgedragen, dan stuurt de DMAC een DEND-signaal (DMA-END) naar de betreffende I/O-C en genereert daarna eventueel een interrupt ten behoeve van de microprocessor. Dit gebeurt via de uitgang $\overline{TRQ/DEND}$.

De aansluiting $\overline{CS}$ /TxAKB functioneert als $\overline{CS}$ -ingang tijdens de MPU-mode. Deze ingang is via een externe adresdecoder verbonden met de adresbus. Alleen tijdens (vier-kanaals) DMA-mode werkt deze aansluiting als TxAKB-uitgang.

De aansluitingen D0 t/m D7 werken alleen als uitgangen als de microprocessor in een van de registers van de DMAC leest. In alle andere gevallen zijn deze aansluitingen als input geschakeld. Ze hebben echter alleen effect als een van de registers van de DMAC is geselecteerd met behulp van de ingangen $\overline{CS}$ en A0 t/m A4.

Op de ingang E moet de systeemklok worden aangesloten. Deze zorgt voor de synchronisatie tussen DMAC, microprocessor, geheugen en I/O-C.

Door het activeren (0 maken) van de ingang $\overline{RESET}$ worden alle registers van de DMAC gereset, met uitzondering van de Address Registers en de Byte Count Registers.

10.14.3 Registerselectie

Fig. 10.72 laat zien bij welke adressen de diverse registers worden geselecteerd. Hierbij moet de ingang $\overline{CS}$ vanzelfsprekend gelijk zijn aan 0. Het basisadres van de DMAC wordt bepaald door de externe adresdecoder. Deze zorgt ervoor dat de ingang $\overline{CS}$ gelijk wordt aan 0 bij één bepaalde combinatie (het basisadres) van de adressignalen A5 t/m A15.

De Byte Count Registers bevatten het aantal bytes (maximaal 65.535), dat door de betreffende randapparaten moet worden overgedragen. Een transfer request wordt alleen gehonoreerd, indien de inhoud van het corresponderende Byte Count Register ongelijk is aan 0. Het Byte Count Register wordt telkens aan het begin van een DMA-cyclus met 1 verlaagd.

Een Address Register bevat het adres van de geheugenlocatie, waarin de volgende byte transfer met het overeenkomstige randapparaat plaatsvindt.

De overige registers worden in de volgende paragrafen besproken.

adres A4A3A2A1A0	geselecteerde register
0 0 0 0 0	Address Register 0 high
0 0 0 0 1	Address Register 0 low
0 0 0 1 0	Byte Count Register 0 high
0 0 0 1 1	Byte Count Register 0 low
0 0 1 0 0	Address Register 1 high
0 0 1 0 1	Address Register 1 low
0 0 1 1 0	Byte Count Register 1 high
0 0 1 1 1	Byte Count Register 1 low
0 1 0 0 0	Address Register 2 high
0 1 0 0 1	Address Register 2 low
0 1 0 1 0	Byte Count Register 2 high
0 1 0 1 1	Byte Count Register 2 low
0 1 1 0 0	Address Register 3 high
0 1 1 0 1	Address Register 3 low
0 1 1 1 0	Byte Count Register 3 high
0 1 1 1 1	Byte Count Register 3 low
1 0 0 0 0	Channel Control Register 0
1 0 0 0 1	Channel Control Register 1
1 0 0 1 0	Channel Control Register 2
1 0 0 1 1	Channel Control Register 3
1 0 1 0 0	Priority Control Register
1 0 1 0 1	Interrupt Control Register
1 0 1 1 0	Data Chain Register

Fig. 10.72 Tabel van de registerselectie

10.14.4 Channel Control Registers

De Channel Control Registers besturen de datatransfer behorende bij de betreffende I/O-Controller.

Bit 0 is de Read/Write-bit. De inhoud van deze bit wordt tijdens DMA-mode op de gelijknamige uitgang van de DMAC gezet. Een 1 betekent dat er in het geheugen wordt gelezen. Bij een 0 wordt in het geheugen geschreven.

Merk op dat de betekenis van het R/W-signaal op de ingang van de I/O-Controller tijdens DMA-mode verandert. In MPU-mode wordt er in de I/O-C gelezen als R/W=1. Tijdens DMA-mode wordt er geschreven in de I/O-C als R/W=1!

De mode control bits 1 en 2 (MCB, respectievelijk MCA), stellen de DMA-mode van de DMAC in (zie fig. 10.73).

MCA bit 2	MCB bit 1	DMA-mode	omschrijving
0	0	mode 2	single byte mode, aanvraag via <u>DRQ2</u>
0	1	mode 3	burst mode, aanvraag via <u>DRQ2</u>
1	0	mode 1	single byte mode, aanvraag via <u>DRQ1</u>
1	1	ongedefinieerd	

Fig. 10.73 De Mode Control bits van de Channel Control Registers

Bedenk dat het systeem in de burst mode is geblokkeerd, totdat de laatste byte is overgedragen! De uitgang DRQ2 blijft zolang actief, ook als de betreffende TxRQ wegvalt. Dit kan gebeuren doordat het device de volgende byte nog niet gereed heeft voor transport. Er wordt dan gewacht totdat TxRQ opnieuw actief wordt. Pas nadat de laatste byte van het blok gegevens is overgedragen, kan de microprocessor doorgaan met het uitvoeren van een programma of het afhandelen van een interrupt, of kan er een andere transferaanvraag worden afgehandeld.

De microprocessor kan de burst mode in principe onderbreken door DGRNT te resetten. De DMAC blijft een DMA request doen totdat alle bytes overgedragen zijn.

Voor de MC6809-microprocessor is het gebruik van uitgang DRQ1 niet van belang. Uitgang DRQ1 gedraagt zich hetzelfde als uitgang DRQ2 in de single byte mode.

DRQ1 wordt toegepast in systemen met MC6800-microprocessoren en wordt daar gebruikt om de processorklok kortstondig stil te zetten via een externe klokgenerator. De microprocessoruitgangen worden hierbij in de hoogimpedante toestand gezet en er kan DMA plaatsvinden. Men spreekt hierbij wel van three-state control stealing. Dit is mode 1. De uitgang DRQ2 wordt in MC6800-systemen gebruikt om de processor via de HALT-ingang in een soort pauzetoestand te

brengen. De interne klokken van de processor lopen hierbij door. Het is nu mogelijk single byte mode of burst mode DMA te plegen. Vandaar dat men hier wel spreekt van halt steal mode (mode 2), respectievelijk halt burst mode (mode 3).

Bit 3 is de Address $\overline{\text{Up/Down}}$ bit. Is bit 3 gelijk aan 0, dan wordt de inhoud van het Address Register verhoogd bij het begin van een DMA-cyclus. Is bit 3 gelijk aan 1, dan wordt het adres telkens verlaagd. Dit geeft de mogelijkheid om het geheugen in beide richtingen te doorlopen.

Bit 4 en 5 worden niet gebruikt.

De read only statusbit 6 is de busy/ready flag. Deze bit geeft aan of er een DMA bloktransfer aan de gang is voor het corresponderende device. Bit 6 wordt geset, indien de bijbehorende TxRQi wordt gedetecteerd, mits de inhoud van het Byte Count Register ongelijk 0 is. Bit 6 wordt gereset tijdens het transport van de laatste byte van het blok gegevens.

De read only statusbit 7 is de DEND flag van het overeenkomstige DMA-kanaal. Deze bit wordt geset tijdens het transport van de laatste byte van het blok gegevens. Tevens wordt dan de uitgang $\overline{\text{IRQ/DEND}}$ laag. Tijdens MPU-mode wordt deze uitgang opnieuw laag, mits de betreffende DMA-END interrupt is ge-enabled. Bit 7 wordt gereset na het lezen van het bijbehorende Channel Control Register.

10.14.5 Priority Control Register

De bits 0 t/m 3 zijn de overeenkomstige Request Enable bits RE0 t/m RE3 voor de kanalen 0 t/m 3. Een 0 in REi blokkeert een transfer-aanvraag van kanaal i.

De bits 4 t/m 6 zijn niet gebruikt.

Bit 7 is de rotate control bit.

Fixed priority

Is bit 7 gelijk aan 0, dan is de prioriteit vast. Dit is de zogenaamde fixed priority. Kanaal 0 heeft de hoogste prioriteit, daarna kanaal 1, vervolgens kanaal 2 en tenslotte kanaal 3. Wordt er via meerdere kanalen tegelijkertijd een transfer request gedaan, dan wordt het kanaal met het laagste nummer altijd het eerste bediend. Het nadeel hiervan is dat de devices met een lagere prioriteit de kans lopen nooit aan de beurt te komen. Dit is het geval als een of meerdere devices met een hogere prioriteit telkens een transfer request doen.

Rotating priority

Als bit 7 gelijk is aan 1, dan krijgen we de zogenaamde

rotating priority. Hierbij is de volgorde van prioriteit aanvankelijk dezelfde als bij de fixed priority. Nadat een device bediend is, krijgt dit echter de laagste prioriteit, waarbij de prioriteiten van de andere devices opschuiven (zie fig. 10.74). Rondgaand blijft de volgorde van de prioriteiten gelijk, te weten 0, 1, 2 en 3. Het verschil is echter dat device 0 niet altijd de hoogste prioriteit heeft. Het is alsof de prioriteit roteert. Hierbij komt elk device met zekerheid aan de beurt. Een bepaald device moet ten hoogste drie DMA-cycli wachten, voordat het weer aan de beurt is. Voor een zeer snel device kan dit echter betekenen dat er een throughput error optreedt. Dit risico kan worden verkleind door dit device in burst mode te bedienen.

prioriteit in afnemende volgorde	afgehandelde aanvraag	wachtende aanvragen
0 1 2 3	-	1,3
2 3 0 1	1	0,1,3
0 1 2 3	3	0,1
1 2 3 0	0	1
2 3 0 1	1	

Fig. 10.74 Prioriteitsvolgorde bij rotating priority (PCR7=1)

10.14.6 Interrupt Control Register

De bits 0 t/m 3 zijn de DEND-IRQ Enable bits DIE0 t/m DIE3 van de overeenkomstige devices. Is bit DIEi geset, dan wordt de DMA-END van device i als een interrupt doorgegeven aan de microprocessor (tijdens MPU mode). Is de betreffende bit gereset, dan is de interrupt geblokkeerd.

Let wel dat de uitgang $\overline{\text{IRQ/DEND}}$ altijd actief (0) wordt tijdens de overdracht van de laatste byte, dus tijdens DMA mode. Dit gebeurt ongeacht de waarde van de DEND-IRQ Enable bits. DEND is de signalering aan het device dat alle bytes zijn overgedragen. TxAKA en TxAKB bepalen voor welk device DEND geldt.

De bits 4 t/m 6 zijn niet gebruikt.

De read only statusbit 7 is de gezamenlijke DEND-IRQ flag. Deze flag wordt geset als DENDi=1 (bit 7 van Channel Control Register i), mits DIEi=1. Bit 7 wordt gereset na het lezen van het Channel Control Register van het device dat de DMA-END heeft veroorzaakt.

10.14.7 Data Chain Register

Met data chaining wordt hier bedoeld het in een ketting zetten van datablokken. Hierbij wordt de inhoud van Address Register 3 en van Byte Count Register 3 gekopieerd in de overeenkomstige registers van het kanaal dat voor data chaining is geselecteerd. Dit kopiëren vindt plaats gedurende de klokperiode, die volgt op het transport van de laatste byte van het blok data van het betreffende kanaal. Dit gebeurt steeds opnieuw. Zodoende wordt telkens hetzelfde geheugengebied van nieuwe informatie voorzien of er wordt telkens hetzelfde geheugengebied uitgelezen.

Data chaining geeft de mogelijkheid om telkenmale in een bepaald geheugengebied te lezen of te schrijven, zonder steeds opnieuw de DMAC te moeten initialiseren. Zo zouden regelmatig de meetgegevens van een bepaald proces op basis van DMA in een geheugenblok gezet kunnen worden. Het programma dat het proces regelt, hoeft zich hierom in het geheel niet te bekommeren. Immers de laatste meetgegevens staan steeds in het geheugengebied. Het regelprogramma hoeft ze daar slechts uit te halen voor verwerking. Het opslaan van de meetwaarden vindt geheel buiten het gezichtsveld van de microprocessor plaats.

De stuurinformatie voor het proces, die wordt berekend uit de meetgegevens, kan in een ander geheugenblok worden opgeborgen. Van hieruit kan deze vervolgens door het proces op basis van DMA worden opgehaald. Ook hier heeft de processor geen bemoeienis mee.

Bij deze toepassing van DMA gaat het niet om snelle gegevensuitwisseling. U ziet dat hier in feite twee samenlopende processen (concurrent processes of parallele processen) zijn gerealiseerd, namelijk een I/O-proces en een verwerkingsproces. Deze processen lopen los van elkaar, hoewel ze elkaars informatie gebruiken. Hierbij moet opgepast worden, indien er gegevens zijn, die meerdere bytes omvatten. Dan kan het gebeuren, dat het I/O-proces op DMA-basis een byte gewijzigd heeft, terwijl het verwerkingsproces het gehele gegeven binnenhaalt, voordat het I/O-proces alle bytes van dit gegeven heeft aangepast. In dit geval wordt dan een totaal verkeerde waarde verwerkt.

Dit is in ons voorbeeld te voorkomen door de DMA in burst mode te laten verlopen. Dan worden alle meetgegevens zonder onderbreking in het geheugen opgeslagen. De volgende bloktransfer kan een hele tijd later plaatsvinden. De bloktransfer stopt immers bij het overdragen van de laatste byte. Daarna moet het betreffende DMA-kanaal opnieuw worden geïnitialiseerd (vanuit kanaal 3), voordat de volgende bloktransfer (in burst mode) begint. Deze start nadat de betreffende TxRQ geactiveerd is door de I/O-Controller. Er moet voor gezorgd zijn, dat TxRQ inactief is tijdens het transport

van de laatste byte. Anders zou meteen een begin gemaakt worden met de volgende bloktransfer. Dan is het hele systeem opnieuw geblokkeerd tot het einde van die bloktransfer.

Bij concurrent processing wordt in het algemeen gebruik gemaakt van semaforen (seinpalen) om aan te geven of een bepaald geheugengebied in bewerking is en derhalve niet door een ander proces mag worden gebruikt. Deze semaforen worden gerealiseerd met behulp van flags.

U zult terecht opmerken dat er op kanaal 3 geen I/O-device aangesloten kan zijn, indien er gebruik wordt gemaakt van data chaining.

Bit 0 van het Data Chain Register is de Data Chain Enable bit DCE. Is deze bit 0, dan treedt er geen data chaining op. Is bit 0 gelijk aan 1, dan treedt er wel data chaining op.

De bits 1 en 2 zijn de Data Chain Select bits DCA, respectievelijk DCB. Hiermee wordt bepaald op welk kanaal data chaining wordt toegepast (zie fig. 10.75). Deze bits hebben vanzelfsprekend alleen betekenis als DCE=1 is.

DCB bit 2	DCA bit 1	geselecteerde kanaal voor data chaining
0	0	0
0	1	1
1	0	2
1	1	ongedefinieerd

Fig. 10.75 Data Chain Select bits

Bit 3 is de Two/Four Channel Select bit. Een 0 selecteert twee kanalen, een 1 selecteert er vier. Werkt de DMAC met slechts twee kanalen, dan functioneert $\overline{CS}/TxAKB$ altijd als $\overline{CS}$ -ingang, zowel in de MPU-mode als in de DMA-mode. In de vier-kanaals mode gedraagt $\overline{CS}/TxAKB$ zich gedurende de DMA-mode als $TxAKB$ -uitgang. In de two channel mode worden de transfer requests van de devices 2 en 3 niet gehonoreerd.

De bits 4 t/m 7 zijn niet gebruikt.

10.14.8 DMAC-handshakes

In paragraaf 10.13 is opgemerkt, dat er bij het aanvragen van een datatransport op basis van DMA sprake is van twee handshakes. De ene vindt plaats tussen I/O-Controller en DMAC (t1 t/m t4, fig. 10.77) en de andere tussen DMAC en microprocessor (T1 t/m T4, fig. 10.77). In deze paragraaf gaan

we dit nader bekijken. Als I/O-C nemen we de Floppy Disk Controller (FDC, type MC6843).

Op een disk (een schijf, voorzien van magnetisch materiaal) wordt de informatie in serie weggeschreven. Hiertoe is een disk verdeeld in sporen (tracks). Dit zijn concentrische cirkels, waarop de informatie (in serie) wordt opgeborgen. Om de gegevens in kleinere porties te kunnen opbergen, is de disk tevens verdeeld in sectoren, zodat elk spoor uit evenveel blokken is opgebouwd als er sectoren zijn. Elk blok kan nu worden geadresseerd middels een spoornummer en een sectornummer.

Wordt een floppy disk (flexibele schijf of diskette) bijvoorbeeld verdeeld in 20 sporen en 32 sectoren, dan bevat deze disk $20 \cdot 32 = 640$ blokken. Wanneer elk blok 256 databytes kan bevatten, heeft deze disk een capaciteit van 160 kbytes. In werkelijkheid kunnen meer bytes worden opgeborgen, immers elk blok bevat onder andere zijn blokadres (spoornummer en sectornummer). Dit wordt er tijdens het zogenaamde formatteren op geschreven. Pas na het formatteren kan er data op de disk worden opgeborgen. Vandaar dat de ongeformatteerde diskcapaciteit groter is dan de geformatteerde. Bevat de disk drive twee lees/schrijfkoppen, dan kan de disk aan beide zijden worden beschreven, zodat er twee maal zoveel informatie kan worden opgeslagen.

Door de lees/schrijfkop in radiale richting te verplaatsen, worden de diverse sporen bereikt. Door de disk zelf te roteren, worden de diverse sectoren bereikt.

Fig. 10.76 toont de signalen die nodig zijn voor het datatransport op basis van DMA. De hierbij relevante registers van de FDC zijn eveneens aangegeven.

We nemen aan dat de FDC is aangesloten op kanaal 0 van de DMAC en dat er verder geen andere I/O-C's aangesloten zijn op de DMAC.

De MC6809-microprocessor meldt DMA Grant door zijn uitgangen BA (Bus Available) en BS (Bus Status) beide 1 te maken.

We laten achtereenvolgens de verschillende fasen van de beide handshakes zien, zoals die optreden bij het aanvragen van een DMA transfer (zie fig. 10.77).

Tijdstip t1

Nadat de microprocessor de DMAC en de FDC heeft geïnitieerd, ligt het initiatief van de data-overdracht bij de FDC. Zodra de FDC data beschikbaar heeft (bij lezen), dan wel data kan ontvangen (bij schrijven), doet hij een aanvraag om DMA te mogen plegen. Hiertoe maakt hij zijn uitgang TxRQ (Transfer ReQuest) gelijk aan 1 (op tijdstip t1). Dit wordt in ons voorbeeld ontvangen op de ingang TxRQ0 van de DMAC.

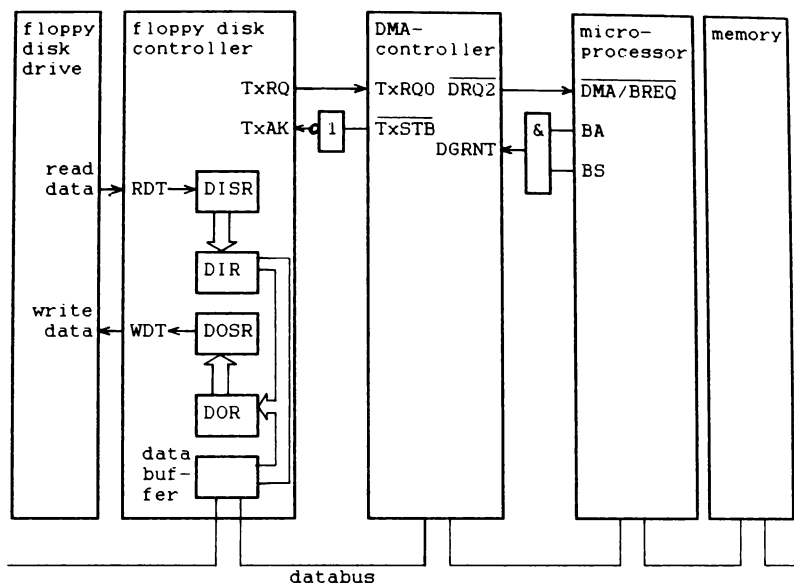


Fig. 10.76 Communicatie met floppy disk drive op basis van DMA

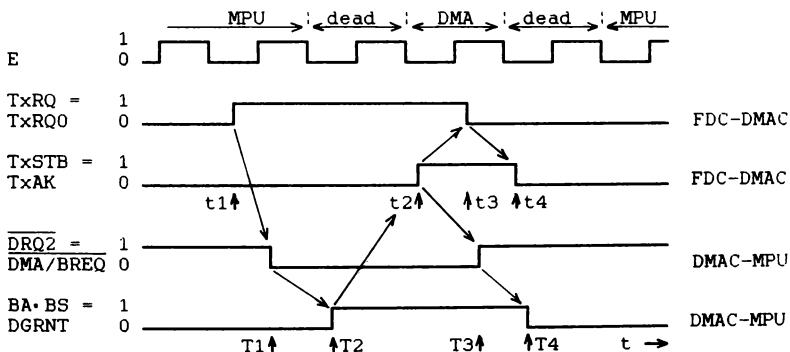


Fig. 10.77 Tijddiagram van de handshakesignalen in de single byte mode

Tijdstip T1

Na het ontvangen van een transfer request maakt de DMAC het signaal $\overline{DRQ2}$ gelijk aan 0, mits de DMAC is ingesteld in mode 2 of mode 3 (tijdstip T1).

Tijdstip T2

De microprocessor detecteert deze DMA-aanvraag op zijn ingang DMA/BREQ (DMA/Bus REQuest) en schakelt zijn R/W-, adres- en datazenders af van de bus. Daarna meldt hij Bus Grant door BA en BS 1 te maken (op tijdstip T2). Hierna kan de DMAC zijn adres- en R/W-zenders activeren.

Tijdstip t2

De DMAC ontvangt deze mededeling op zijn DGRNT-ingang (DMA GRaNT) en geeft een transfer acknowledge (TxAK) door aan de FDC (op tijdstip t2). Omdat er slechts één DMA-device is aangesloten, kan dit geschieden door de uitgang TxSTB (Transfer STroBe) via een inverter te verbinden met TxAK van de FDC.

Het geheugenadres en de juiste waarde van R/W zijn reeds door de DMAC op de bus gezet. Is R/W=0, dan zet de FDC de data op de databus en wordt deze in het geheugen ingelezen. Voor R/W=1 zet het geheugen de data op de databus en neemt de FDC deze over. Er vindt nu een overdracht van één databyte plaats. De klok E zorgt bij dit alles voor de synchronisatie.

Tijdstip t3

De FDC reset het signaal TxRQ na het ontvangen van TxAK=1. TxRQ wordt weer 1 gemaakt, zodra de FDC gereed is voor het transport van de volgende byte.

Tijdstip t4

Als de DMAC is ingesteld in de burst mode en als aan het einde van de klokcyclus TxRQ gelijk is aan 1, blijft TxSTB actief en wordt de volgende byte overgedragen.

Voor TxRQ=0 maakt de DMAC na de DMA-cyclus het signaal TxSTB gelijk aan 1, zodat TxAK=0 wordt (tijdstip t4). Hiermee is de FDC-DMAC-handshake afgelopen. Er wordt nu gewacht totdat TxRQ opnieuw 1 wordt. Een nieuwe handshake is dan begonnen en er kunnen wederom een of meerdere bytes worden overgedragen. Zo vinden er handshakes plaats tussen FDC en DMAC (t1, t2, t3, t4, t1, enzovoorts), totdat alle bytes zijn overgedragen. De handshake signalen tussen DMAC en MPU veranderen hierbij niet (DRQ2=0 en DGRNT=1).

Indien TxRQ=0 is, wordt TxSTB in de single byte mode gedeactiveerd na de DMA-cyclus (tijdstip t4). De handshake signalen tussen FDC en DMAC zijn nu in de rusttoestand teruggekeerd. De DMAC schakelt zijn zenders af van de bus en geeft de besturing terug aan de microprocessor. Nu kan er pas een nieuwe DMA-transfer plaatsvinden nadat de bus wordt aangevraagd aan de processor. Hiervoor zijn zowel een FDC-DMAC als een DMAC-MPU handshake nodig (via de tijdstippen t1, T1, T2 en t2).

Tijdstip T3

In de burst mode blijft DRQ2 actief, totdat alle bytes zijn overgedragen. De DMAC blijft hierbij bus master, ook al moet er

worden gewacht tot de FDC opnieuw TxRQ 1 maakt. Er kunnen dan klokcycli ontstaan, waarin geen data-overdracht plaats vindt. Men spreekt hier wel van dummy cycli. Tijdens de klokcyclus waarin de laatste byte wordt overgedragen, wordt DRQ2 gedeactiveerd door de DMAC (tijdstip T3). In de single byte mode wordt DRQ2 gedeactiveerd aan het einde van elke DMA-cyclus (tijdstip t3). De DMAC ontkoppelt zijn zenders van de bus na de DMA-cyclus en de MPU wordt bus master.

Tijdstip T4

Na het inactief (1) worden van DRQ2, haalt de microprocessor het signaal DGRNT weg (op tijdstip T4). Hiermee zijn de handshakesignalen tussen DMAC en microprocessor in de rusttoestand teruggekeerd. De microprocessor neemt de besturing van de bus over.

Indien alle bytes overgedragen zijn, blijft DRQ2 inactief, totdat de DMAC opnieuw wordt geactiveerd. Hiertoe moet het Byte Count Register worden geladen met een waarde ongelijk aan 0. Zijn niet alle bytes overgedragen, dan kan er een nieuwe handshake worden gestart doordat de FDC het signaal TxRQ activeert.

transfer requests door de FDC

De FDC maakt zijn uitgang TxRQ gelijk aan 1 als de Data Transfer ReQuest bit (DTRQ) geset wordt. DTRQ is een van de statusbits van de FDC.

Tijdens schrijfoperaties op de disk wordt de Data Transfer Request bit geset als het Data Output Register (DOR) gereed is om data te accepteren (zie ook fig. 10.76). Hierdoor wordt via de FDC-uitgang TxRQ de bus aangevraagd. Na toewijzing van de bus aan de FDC (TxAK=1) wordt er in DOR geschreven. Hierdoor wordt de DTRQ-bit gereset. De uitgang TxRQ is reeds gereset bij het ontvangen van TxAK=1.

Is de vorige byte in het Data Output Shift Register (DOSR) uit dit register geschoven en zodoende bit voor bit op de disk geschreven, dan wordt de inhoud van DOR overgenomen in DOSR om vervolgens serieel op de disk te worden geschreven. Nu is DOR opnieuw gereed voor het ontvangen van data. Derhalve zal DTRQ en dus ook TxRQ geset worden, zodat een nieuwe busaanvraag gedaan wordt.

Tijdens het leegschuiven van DOSR moet de volgende byte in DOR worden gezet. Gebeurt dit later, dan treedt er een transferfout op. De diskette kan niet even stilgezet worden om te wachten op data. Er kan slechts per blok worden gestopt.

Tijdens leesoperaties op de disk worden DTRQ en TxRQ geset, als het Data Input Register (DIR) wordt geladen vanuit het Data Input Shift Register (DISR, zie ook fig. 10.76). Het lezen vanaf de diskette gaat intussen gewoon door. Dit betekent dat er bit voor bit informatie van de diskette komt, die in DISR wordt geschoven. Voordat DISR vol is, moet de data

uit DIR zijn gelezen. anders treedt er wederom een transferfout op.

Wordt de busaanvraag via de uitgang TxRQ gehonoreerd (TxAK wordt 1), dan wordt de inhoud van DIR op de databus gezet en opgeborgen in het geheugen. Er wordt hierbij gelezen in DIR. Hierdoor wordt de DTRQ-bit gereset. TxRQ is al gereset bij het ontvangen van TxAK=1.

Zodra DIR opnieuw gevuld wordt vanuit DISR. worden DTRQ en TxRQ geset, waardoor een nieuwe busaanvraag wordt gedaan.

U hebt nu gezien dat het initiatief voor een busaanvraag bij de I/O-Controller ligt. Telkens als de I/O-C gereed is voor de overdracht van een byte, wordt de bus aangevraagd. Is de bus toegewezen, dan wordt de aanvraag door de I/O-C weggehaald en vindt de overdracht van data plaats tussen geheugen en I/O-C.

Voorbeeld 10.16 (DMA met FDC)

We willen in het voorbeeld van deze paragraaf een blok data van 256 bytes in single byte mode van de diskette lezen en de inhoud hiervan opbergen in het geheugen vanaf locatie \$48A0. De DMA-END moet op basis van interrupt worden afgehandeld. Er vindt geen data chaining plaats.

De registers van de DMAC moeten na de initialisatie de volgende inhoud hebben (zie ook appendix F), waarbij voor X willekeurig 0 of 1 mag worden ingevuld:

(Address Register 0) = \$48A0

(Byte Count Register 0) = \$0100 (= 256)

(Channel Control Register 0) = %00XX 0000

Het Address Register moet worden verhoogd, de bus wordt aangevraagd via DRQ2 en wel in single byte mode en er moet worden geschreven in het geheugen.

De overige Address, Byte Count en Channel Control Registers zijn niet van belang.

(Priority Control Register) = %XXXX 0001

Omdat er slechts één device is aangesloten, is er geen verschil tussen fixed en rotating priority. Alleen device 0 (de FDC) kan een transfer aanvragen. Het is aan te raden de request enables van de niet-aangesloten transfer request ingangen te resetten, opdat storingen op deze ingangen geen effect hebben.

(Interrupt Control Register) = %0XXX 0001

De DMA-END van de FDC moet worden afgehandeld op basis van interrupt-I/O. Bit 1, bit 2 en bit 3 zijn eigenlijk don't cares, want de overeenkomstige transfer request ingangen zijn niet met devices verbonden.

(Data Chain Register) = %XXXX XXX0

Omdat er slechts één device een transfer request kan doen, maakt het niet uit of de 2- of 4-channel mode wordt ingesteld. Bit 1 en bit 2 zijn don't cares, omdat data chaining is gedisabled.

10.15 Samenvatting

Er zijn diverse onderverdelingen te maken met betrekking tot I/O:

a. adressering

- * memory-mapped I/O
I/O-adressen worden behandeld als gewone geheugenadressen.
- * I/O-mapped I/O
Er zijn aparte instructies voor communicatie met I/O-devices.

b. afhandeling

- * programmed I/O
I/O vindt op initiatief van het programma.
- * Interrupt-I/O
I/O vindt plaats op initiatief van het I/O-device (via een interrupt).
- * DMA
I/O vindt plaats buiten de microprocessor om. Het initiatief ligt bij het I/O-device.

c. communicatiekanaal

- * serieel transport
De informatie wordt bit voor bit via een enkele lijn overgedragen.
- * parallel transport
Er worden een aantal bits tegelijk via een aantal lijnen overgedragen.

d. timing

- * synchroon transport
Er wordt een klokpuls meegezonden ten behoeve van de synchronisatie.
- * asynchroon transport
Er wordt geen klokpuls meegezonden.

Synchronisatie van het datatransport kan geschieden met behulp van:

1. strobing

Een puls geeft aan wanneer de data geldig is.

2. synchronisatie-informatie binnen de data zelf

- Zorgen voor voldoende flanken in de data, zodat de ontvanger hierop zijn klok kan synchroniseren of
- Werken met een startbit om aan te geven wanneer een volgende byte wordt verzonden.

3. handshaking

Een vraag-en-antwoord-spel tussen zender en ontvanger ten behoeve van het verzenden van data.

Er is onderscheid in

- pulsed handshake
De vraag- en antwoordsignalen zijn pulsen.
- interlocked handshake
(Return to Zero of Non Return to Zero)
De handshakesignalen overlappen elkaar.

Enkele interface IC's:

Peripheral Interface Adapter (MC6821)

- parallele communicatie (2 maal 8 lijnen voor input of output)
- 4 interruptingangen
- strobing
- handshaking

Zie verder appendix C.

Asynchronous Communications Interface Adapter (MC6850)

- seriële communicatie (één lijn voor input en één lijn voor output)
- start/stopbit protocol
- 7 of 8 databits
- even, oneven of geen pariteit
- 1 of 2 stopbits
- interruptmogelijkheden

Zie verder appendix D.

Programmable Timer Module (MC6840)

- 3 onafhankelijke timers
- externe klok of systeemklok E te gebruiken bij aftellen
- via hardware en software te starten
- interruptmogelijkheden (voor time-outs)
- 4 operatiemodes:
 1. continuous mode
voor kloksignaal met instelbare frequentie en pulsbreedte en voor interrupts op programmeerbare tussenpozen
 2. single-shot mode
voor puls met instelbare breedte en na instelbare wachttijd
 3. frequency comparison of period measurement mode
voor het meten van een periodetijd
 4. pulse width measurement
voor het meten van een pulsbreedte

Zie verder appendix E.

Direct Memory Access Controller (MC6844)

- 4 DMA-kanalen
- single byte of burst mode
- interruptmogelijkheden
- fixed of rotating priority
- data chain functie

Zie verder appendix F.

10.16 Opgaven

- 1 Schrijf een subroutine die ervoor zorgt dat de waarden op de even PA-lijnen van de PIA geïnverteerd worden weggestuurd via de oneven PA-lijnen: $(PA(i+1)) := (PA(i))$ voor $i = 0, 2, 4$ en 6 . Tevens moet het sign-and-magnitude getal op PB0 t/m PB3 als two's complement getal worden weergegeven op PB4 t/m PB7.

Schrijf ook de subroutine voor het initialiseren van de PIA.

- 2 In een woonwijk wordt een voetgangeroversteekplaats beveiligd met verkeerslichten. In de rusttoestand zijn de voetgangerslichten rood en de andere lichten zijn groen. Drukt een voetganger op een van de drukknoppen aan weerszijden van de straat, dan moeten de gele lichten gaan branden (gedurende 3 s) en daarna de rode lichten. 1 s later moet het voetgangerslicht groen worden en gedurende 15 s groen blijven. Daarna wordt het voetgangerslicht rood en 2 s later wordt het verkeerslicht groen. Dit moet vervolgens tenminste 20 s groen blijven.

De drukknoppen zijn aangesloten op de $\overline{NMI}$ -ingang van de microprocessor.

De NMI-interruptvector moet worden opgeslagen op de locaties \$DFFC en \$DFFD.

De lichten worden gestuurd vanuit een register met als adres LICHT = \$FF01.

Hierbij is: bit 0 = groen voor voetgangers

bit 1 = rood voor voetgangers

bit 5 = groen voor overige verkeer

bit 6 = geel voor overige verkeer

bit 7 = rood voor overige verkeer

Gegeven is een subroutine WACHT, waarin (B) seconden worden afgeteld.

Schrijf een programma voor deze verkeersregeling.

- 3 De stand van de schakelaars, verbonden met PB0 t/m PB3 van een PIA, moet zichtbaar worden gemaakt op de LED's, aangesloten op PB4 t/m PB7. Verandert een schakelaar van stand, dan moet dit zo vlug mogelijk op de LED's worden getoond. Tegelijkertijd loopt er over de LED's PA0 t/m PA7 een looplicht vanaf bit 0 naar bit 7 en vandaar weer opnieuw vanaf bit 0.

De snelheid van het looplicht moet worden verdubbeld bij een opgaande flank op CA1. De loopsnelheid moet worden gehalveerd bij een neergaande flank op CB1. Het aanpassen van de snelheid moet geschieden op basis van interrupt-I/O. De tijd, gedurende welke een LED op PAi oplicht, moet liggen tussen ongeveer 50 ms en ongeveer 5 s.

Een klokcyclus duurt 1 μ s.

De IRQ-interruptvector moet staan op \$DFF8 en \$DFF9.

Ontwerp het bijbehorende programma.

- 4 Teken het tijddiagram van een pulsed handshake voor data-uitvoer met PIA's, uitgaande van de beginsituatie, waarbij nog geen data is overgedragen.
Geef hierbij aan hoe de diverse registers van beide PIA's moeten zijn ingesteld in geval van interrupt-I/O.
- 5 Schrijf een interrupt service routine voor PIA-I/O, die ervoor zorgt dat PIA-M van voorbeeld 10.9 input data krijgt aangeboden en dat de handshake-procedure wordt afgehandeld (zie ook fig. 10.41).
Neem aan dat het te verzenden byte op de locatie met het adres DATA is opgeslagen.
- 6 Via een ACIA moeten de 8-bits gegevens op de geheugen-locaties \$8000 t/m \$8700 verzonden worden. Er wordt gecommuniceerd zonder pariteit en met één stopbit. De transportsnelheid bedraagt 1200 Baud.
We kunnen beschikken over een 1,2 kHz oscillator en over een 76.8 kHz oscillator.
Het basisadres van de ACIA is \$4000.
De IRQ-interruptvector moet worden opgeborgen op \$DFF8 en \$DFF9.

Het zenden mag pas beginnen na het ontvangen van het karakter XON (= DC1, zie appendix A).

Bij het ontvangen van XOFF (= DC3, zie appendix A) wordt met zenden gestopt, totdat opnieuw XON wordt ontvangen.

Ontwerp het bijbehorende programma, waarbij het zenden op basis van programmed I/O en het ontvangen op basis van interrupt-I/O plaatsvindt.

Er hoeft niet te worden gelet op overrun of framing error.

- 7 Mechanische contacten hebben de eigenschap dat ze bij het schakelen terugveren tegen het andere contact. Men noemt dit denderen (bouncing). Hierdoor wordt het contact een aantal malen achter elkaar gemaakt en verbroken, totdat het contact uiteindelijk niet meer terugveert en de verbinding gesloten blijft.
Door dit denderen ontstaat er een pulstrein (zie fig. 10.78).

We willen nu de dendertijd T_d meten. We doen dit met behulp van een timer.

Deze wordt ingesteld op de maximale tijd van 65.535 μ s.
De timer wordt gestart bij de eerste flank op SWITCH.

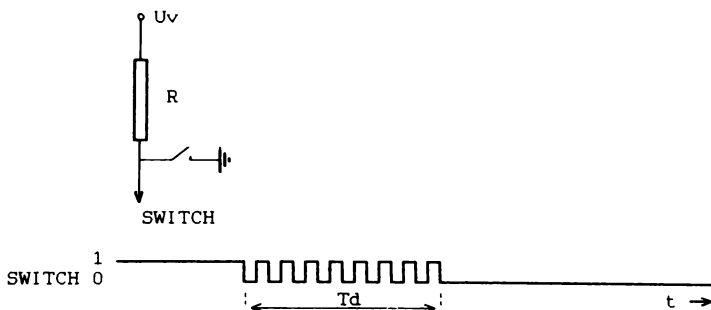


Fig. 10.78 Het denderen van contacten

Het signaal SWITCH wordt ontvangen via de CA1-ingang van een PIA. Bij elke neergaande flank op CA1 wordt de tellerstand van de timer in het geheugen opgeborgen (op TELLER). Zo bevat TELLER aan het einde van de meting de tellerstand die optrad bij de laatste interrupt.

De meettijd is groter dan de dendertijd!

Het IRQ-vectoradres is \$DFF8.

In het hoofdprogramma wordt gewacht op het einde van de meting (de tellerstand is 0). Daarna wordt de dendertijd berekend en via PA en PB zichtbaar gemaakt. PA is de meestwaardige byte.

Het voordeel van het gebruik van de timer is dat de tijdmeting niet verstoord wordt door het opbergen van de tellerstand in het geheugen. Deze instructies kosten immers tijd. De timer loopt hierbij echter gewoon door.

Schrijf de software die nodig is om de dendertijd te meten.

- 8 Een FDC en een data-acquisitiestation zijn aangesloten op een DMAC. Beide werken in de single byte mode. De aanvragen voor DMA verlopen via DRQ2. De DMA-END moet worden genegeerd. De FDC moet altijd de hoogste prioriteit hebben. Er wordt data (128 bytes) gelezen van een diskette en opgeborgen vanaf adres \$3000 in het geheugen. De DMA-END moet worden verwerkt op basis van interrupt-I/O. De gegevens van het data-acquisitiestation worden opgeborgen vanaf geheugenlocatie \$9000. Het gaat hierbij om 30 bytes. Deze 30 bytes worden telkens overschreven met nieuwe gegevens van het station.

Wat moet de inhoud zijn van de diverse registers van de DMAC na initialisatie?

11 MICROPROCESSOR INTERFACES

11.1 Specifieke doelstelling

Na het bestuderen van hoofdstuk 11 moet u:

- * De signalen van de microprocessor kennen.
- * De signalen van de microprocessor kunnen toepassen.
- * Een priority interrupt controller kunnen gebruiken.
- * De tijddiagrammen van de signalen van microprocessor en andere componenten (RAM, ROM, PIA, ACIA, PTM, DMAC) begrijpen.
- * De microprocessor op de juiste wijze kunnen aansluiten op zijn omringende componenten.
- * De maximaal toelaatbare vertraging van bus en adresdecoder kunnen bepalen aan de hand van de timingdiagrammen.
- * Kunnen berekenen of aan alle timing-eisen van de componenten in het microprocessorsysteem wordt voldaan.

11.2 Inleiding

De timing van digitale schakelingen is altijd afhankelijk van de volgende factoren:

- de flanksteilheid van hetingangssignaal
- de spanningsniveaus waarop gemeten wordt
- de belasting van de uitgangen van de schakeling
- temperatuur en voedingsspanning
- circuitspreiding

In de datasheets van de componenten wordt vastgelegd hoe en onder welke omstandigheden de timing is gespecificeerd. Een pulsbreedte gemeten op 1 V zal een andere waarde opleveren dan wanneer diezelfde pulsbreedte wordt gemeten op 3 V, tenzij de flanken oneindig steil zijn (zie fig. 11.1).

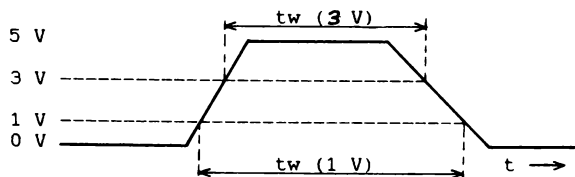


Fig. 11.1 Tijdmetering is afhankelijk van meetniveaus en flanksteilheid

De MC6809 is, net als de meeste andere microprocessoren, TTL-compatibel, dat wil zeggen, hij is rechtstreeks aansluitbaar op TTL-schakelingen. TTL is Transistor-Transistor-Logic. Spanningen kleiner dan 0.8 V worden altijd gezien als logisch 0, spanningen groter dan 2.0 V als logisch 1. Vandaar dat de timing bij de MC6809 wordt gespecificeerd met betrekking tot meetpunten op 0.8 V, dan wel op 2.0 V, afhankelijk van het gewenste logische niveau (0 of 1). Voor spanningen die tussen 0.8 V en 2.0 V liggen, kan niet met zekerheid gezegd worden of deze als 0 of als 1 worden geïnterpreteerd door de TTL-schakeling. Dit hangt af van circuitspreiding, voedingsspanning en dergelijke.

Bij vele TTL-schakelingen wordt het spanningsniveau van 1.5 V als referentie gekozen voor het specificeren van de timing.

Circuitspreiding wordt door de fabrikant opgegeven in de zin dat hij minimale en/of maximale waarden specificeert.

Temperatuur en voedingsspanning hebben invloed op het gedrag van de halfgeleiderschakeling. Deze invloed is veelal begrepen in de gespecificeerde minima en maxima.

Treden in uw schakeling afwijkende flanken op van de door de fabrikant gespecificeerde flanken, dan moet u hiervoor corrigeren.

Flanken mogen doorgaans niet te slap zijn, omdat anders het gevaar bestaat dat de schakeling gaat oscilleren. De schakeling verandert van logisch niveau als de ingang een bepaalde spanning, de omklapspanning of drempelspanning overschrijdt. In de schakeling treden dan spanningsveranderingen op, waarbij er stromen gaan lopen. Deze stroomveranderingen veroorzaken in de parasitaire zelfinducties van de verbindingen kleine stoorspanningen. Deze kunnen tot gevolg hebben dat de ingangsspanning weer beneden de omklapspanning komt, waardoor er opnieuw geschakeld wordt, enzovoorts. Om dit te voorkomen moet ervoor worden gezorgd dat de ingangsspanning voldoende ver van de omklapspanning is verwijderd, op het moment dat de schakeling reageert. Derhalve moet de spanningsflank voldoende steil zijn.

Bij ingangen met hysteresis hoeft dit niet, omdat deze ingangen verschillende drempelspanningen hebben bij het omhoog en omlaag schakelen. Zo is de drempelspanning bij het omhoog schakelen bijvoorbeeld 1.6 V en bij het omlaag schakelen 1.2 V.

Ook voor een afwijkende belasting moet de gespecificeerde timing worden gecorrigeerd. Het is duidelijk dat een grotere belastingscapaciteit een grotere vertraging met zich meebrengt. Een ingang van een schakeling heeft een capaciteit van ongeveer 5 pF, een uitgang ongeveer 10 pF, terwijl de bedradingscapaciteit in de orde van grootte van 1 pF/cm ligt.

Vanzelfsprekend specificeren de fabrikanten hun schakelingen

tussen ingangen en uitgangen. De gebruiker moet dus zelf de looptijden bij gebruik van lange verbindingen meenemen in de timingberekeningen. Een looptijd van een kabel ligt in de orde van grootte van 5 ns/m. Het berekenen van de correcties op de timing ligt buiten het kader van dit boek, zodat hierop niet verder wordt ingegaan.

11.3 Microprocessorsignalen

11.3.1 Voeding

Een (actieve) schakeling kan alleen functioneren, indien deze wordt voorzien van voedingsspanning. Derhalve zijn er tenminste twee voedingsaansluitingen nodig. Bij de MC6809 zijn dit:

$V_{cc} = +5\text{ V} \pm 5\%$

$V_{ss} = 0\text{ V}$

11.3.2 Databus

De 8 signalen D0 t/m D7 kunnen input of output zijn. De microprocessor kan immers data binnenhalen en wegsturen.

11.3.3 Adresbus

Via de 16 adresuitgangen A0 t/m A15 maakt de processor kenbaar met welk device hij wil communiceren.

Gebruikt de processor de bus niet (bijvoorbeeld als hij intern bezig is met rekenwerk), dan zet hij \$FFFF op de adresbus. Dat het hier om een zogenaamd dummy adres gaat, wordt aangegeven door het signaal Bus Status BS = 0.

11.3.4 Read/Write

Het uitgangssignaal $R/\bar{W}$ geeft de richting van het datatransport aan. Voor $R/\bar{W} = 0$ zet de processor data op de databus, voor $R/\bar{W} = 1$ haalt de processor data van de databus.

Als de microprocessor de bus niet gebruikt, geldt $R/\bar{W} = 1$. De databus staat dan in de input mode, zodat de datazenders zijn afgeschakeld van de bus.

11.3.5 Klokken

Oscillator

De MC6809 heeft een eigen oscillator. Deze kan worden aangestuurd met een parallelresonant kristal, dat aangesloten

wordt tussen de ingangen XTAL en EXTAL.

Er kan ook gebruik worden gemaakt van een externe oscillator. Deze moet dan worden aangesloten op de ingang EXTAL, waarbij XTAL aan 0 V moet worden gelegd.

Nadat de voedingsspanning na inschakelen de waarde van 4,75 V bereikt heeft, duurt het hoogstens 100 ms totdat de interne oscillator correct functioneert. Dit impliceert dat de externe resetschakeling de RESET-ingang van de microprocessor gedurende deze tijd 0 moet houden om te voorkomen, dat de processor ongewenste acties gaat uitvoeren.

De kristalfrequentie of de frequentie van de externe oscillator wordt inwendig door vier gedeeld. Op deze wijze ontstaan er acht nauwkeurig gedefinieerde momenten, waarop de processor achtereenvolgens diverse acties kan ondernemen binnen één cyclus van de interne klok (zie fig. 11.2).

De frequentie van het kristal of van de externe oscillator mag maximaal 4 MHz, 6 MHz of 8 MHz bedragen voor respectievelijk de MC6809, de MC68A09 en de MC68B09. De interne klokfrequentie bedraagt dus maximaal 1 MHz, 1,5 MHz, respectievelijk 2 MHz. De cyclustijden zijn dan minimaal 1 μ s, 0,667 μ s en 0,5 μ s. De klokfrequentie is begrensd vanwege de eindige snelheid van de schakeling. Zou de klokfrequentie te hoog worden, dan worden de inwendige parasitaire capaciteiten niet voldoende opgeladen of ontladen, om de niveaus van 0 naar 1 (of omgekeerd) te laten gaan. De microprocessor werkt dan niet meer.

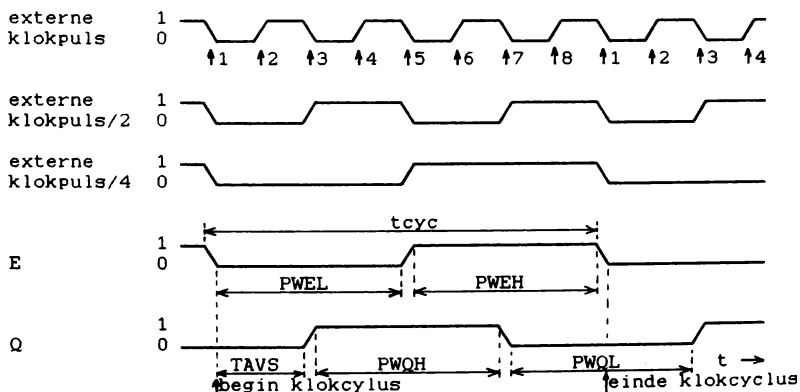


Fig. 11.2 Klokpulsen

De minimaal toelaatbare frequentie van het kristal of van de externe oscillator is 0,4 MHz. Dit betekent dat de minimale frequentie van de interne processorklok 0,1 MHz bedraagt en de cyclustijd maximaal 10 μ s is.

Dynamische geheugenelementen

Dat de microprocessor niet met tragere klokken kan werken, komt doordat de microprocessor dynamisch is. Dit betekent dat de registers van de processor opgebouwd zijn uit dynamische geheugenelementen. Een dynamisch geheugenelement bestaat in feite uit een capaciteit. Een capaciteit kan lading vasthouden. De afwezigheid van lading in de capaciteit kan worden gedefinieerd als een 0, de aanwezigheid als een 1. De lading van de capaciteit (die zeer klein is bij dynamische halfgeleidergeheugens) blijft niet lang behouden. Ten gevolge van lekstromen (stromen in de gesperde transistors) kan de lading langzaam toe- of afnemen, zodat na enige tijd het spanningsniveau kan veranderen van 0 naar 1 of omgekeerd. Hierdoor verandert de informatie in het dynamische geheugen (de microprocessorregisters). Om dit te voorkomen moet deze informatie tijdig worden opgefrist. Derhalve wordt de informatie in de registers regelmatig gelezen en teruggeschreven. Deze zogenaamde refresh van de registers wordt door de microprocessor zelf verzorgd (self-refresh). Alle acties binnen de processor, dus ook de refresh, gebeuren synchroon, onder besturing van de klokpuls. Heeft de klok een te lage frequentie, dan vindt de refresh plaats, nadat de informatie is weggelekt. Vandaar dat de klokfrequentie voor dynamische geheugenschakelingen aan een minimale waarde gebonden is.

Processorsnelheid

Vergelijkt u de snelheden van de microprocessoren van diverse fabrikanten, dan is het niet voldoende te letten op de kristalfrequentie en de deelfactor. Van groot belang is ook hoeveel klokcycli de uitvoering van de verschillende instructies kost. Er zijn speciale programma's ontwikkeld voor het weergeven van de snelheid van processoren, de zogenaamde benchmarkprogramma's (benchmark = referentiepunt). Zo'n benchmarkprogramma bevat een bepaalde samenstelling van diverse instructies. Het vergelijken van de verschillende processoren met behulp van een benchmarkprogramma geeft echter geen goed uitsluitsel omtrent de juiste keuze van een processor. Op de eerste plaats komt de verhouding tussen de diverse instructies van het benchmarkprogramma doorgaans niet overeen met uw applicatieprogramma's. Indien u vrijwel alleen rekent, moet u een eigen benchmarkprogramma schrijven, waarin vrijwel alleen rekeninstructies voorkomen. Gaat het vooral om input/output-instructies, dan moeten juist deze instructies in meerderheid in uw benchmarkprogramma voorkomen.

Ook is bij de keuze van het type microprocessor van belang welke componenten bij deze processor verkrijgbaar zijn. Ook dit bepaalt de efficiëncy van de processor. Zo zijn er diverse coprocessoren leverbaar, die de processor ondersteunen bij het uitvoeren van zijn taak. Een fast Fourier

processor kan zeer snelle Fourriertransformaties uitvoeren. Een floating point processor kan snel berekeningen doen met floating point getallen (zie paragraaf 2.12), waarbij goniometrische, logaritmische en exponentiële berekeningen worden uitgevoerd. De processor delegeert deze specifieke functies aan een of meerdere coprocessoren.

Klokuitgangen

De uitgangssignalen E en Q worden gebruikt om de externe schakelingen synchroon te laten lopen met de microprocessor. Als de processor bijvoorbeeld van de databus leest, moet het geheugen of de I/O ervoor zorgen, dat deze data tijdig op de bus aanwezig is. De tijddiagrammen voor het lezen en schrijven worden gegeven in fig. 11.3 en 11.4. De Quadratuur klok Q loopt een kwart cyclus voor op de Enable klok E (zie fig. 11.2). Een klokcyclus begint bij de neergaande flank van E.

Timing bij het lezen door de processor

Specificatie van de timing van de MC6809 (zie ook fig. 11.2); de referentiepunten zijn 0,8 V voor het 0-niveau en 2,0 V voor het 1-niveau:

tijd van E L tot Q J : $200 \text{ ns} < t_{AVS} < 250 \text{ ns}$

cyclustijd: $1,0 \mu\text{s} < t_{cyc} < 10 \mu\text{s}$
 $< 16 \mu\text{s}$ als $MRDY=1$ of als $DMA/BREQ=0$

pulse width E low: $430 \text{ ns} < PWEL < 5000 \text{ ns}$

pulse width E high: $450 \text{ ns} < PWEH < 15500 \text{ ns}$

pulse width Q low: $450 \text{ ns} < PWQL < 15500 \text{ ns}$

pulse width Q high: $430 \text{ ns} < PWQH < 5000 \text{ ns}$

E, Q rise, fall time: $t_r, t_f < 25 \text{ ns}$

address hold time: $t_{AH} > 20 \text{ ns}$ (geldt voor A_i en $R/\bar{W}$,
niet voor BA en BS)

address valid to Q: $t_{AQ} > 50 \text{ ns}$ (geldt voor A_i , $R/\bar{W}$,
 BA en BS)

read data setup time: $t_{DSR} > 80 \text{ ns}$

read data hold time: $t_{DHR} > 10 \text{ ns}$

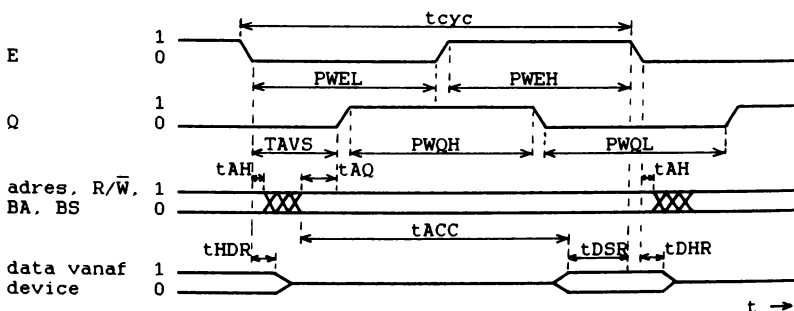


Fig. 11.3 Tijddiagram bij het lezen

Aan het begin van een klokcyclus wordt het adres op de adresbus gezet en worden de signalen R/W, BA en BS op de controlbus gezet. De vorige waarden van Ai en R/W blijven echter nog minstens 20 ns geldig. De nieuwe waarden van Ai, R/W, BA en BS zijn uiterlijk 50 ns voor de opgaande flank van Q geldig.

De microprocessor schakelt zijn datazenders af bij het lezen. Deze zenders komen in de hoogimpedante toestand te staan. Er zijn dan geen zenders actief op de databus. In het tijddiagram is dit aangegeven met het tussenniveau. Dit betekent niet dat de spanning tussen het 0-niveau en het 1-niveau in ligt. Het niveau is in principe ongedefinieerd, hoewel het in de praktijk veelal 1 is vanwege pullup weerstanden op de datalijnen.

Adres en R/W gaan via de bussen naar het geadresseerde geheugen of I/O-device. Geheugen of I/O-device schakelt zijn datazenders in en zet de data op de databus. Slechts één device mag zijn zenders activeren, opdat er niet meer dan één zender actief is op een bepaalde lijn. Dit betekent dat elk adres uniek moet zijn (zie ook paragraaf 10.2).

De data moet tenminste 80 ns voor de neergaande flank van E op de aansluitpennen van de microprocessor aanwezig zijn. Op deze flank wordt de data door de processor ingelezen. Dit betekent dat de in fig. 11.3 aangegeven tijd tACC de tijd is gedurende welke toegang moet worden verkregen tot geheugen of I/O. Binnen deze tijd moet het volgende gebeuren:

- Het adres moet over de bus worden vervoerd en via de adresdecoder het juiste device selecteren.
- Het geselecteerde device moet vervolgens data op de databus zetten.
- deze data moet tenslotte via de databus naar de processor overgebracht worden.

De accestijd tACC is gelijk aan (zie fig. 11.2):

$$tACC = tcyc - tDSR - tf(E) - (tAVS - tAQ)$$

Komt de data te laat, dan leest de processor verkeerde gegevens in. De meest ongunstige situatie treedt op als het adres zo laat mogelijk wordt aangeboden (dus als tAQ minimaal is, dus 50 ns) en als de data zo vroeg mogelijk beschikbaar moet zijn (dus als tcyc minimaal, dus 1 μ s is). De setup tijd tDSR mag natuurlijk groter zijn (de data mag vroeger komen), maar dat hoeft niet. Dit betekent dat in de formule van tACC de minimale waarde tDSR (80 ns) mag worden ingevuld.

We vinden zo:

$$tACC(\text{min.}) = 1000 - 80 - 25 - (250 - 50) = 695 \text{ ns}$$

Verder moet de data tenminste tot 10 ns na de neergaande flank van E gehandhaafd blijven, opdat deze correct wordt ingelezen.

Timing bij het schrijven door de processor

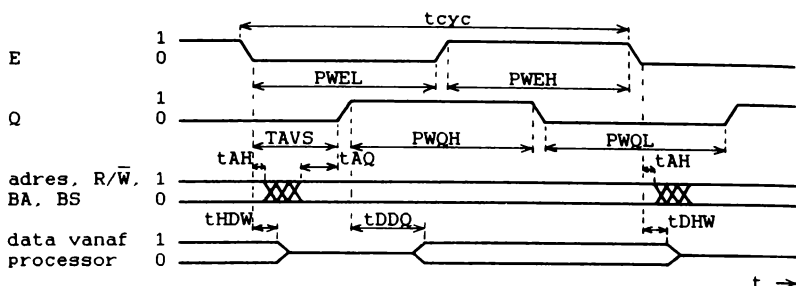


Fig. 11.4 Tijddiagram bij het schrijven

Specificaties van de MC6809:

$t_{DHW} > 30 \text{ ns}$ (write data hold time)

$t_{DDQ} < 200 \text{ ns}$ (data delay time from Q)

Bij het schrijven activeert de microprocessor zijn datazenders na de opgaande flank van Q. Het duurt dan maximaal 200 ns, voordat de data op de uitgangen van de microprocessor geldig is (zie fig. 11.4).

De data blijft stabiel tot tenminste 30 ns na de neergaande flank van E. Daarna worden de datazenders weer hoogimpedant. De datazenders van de microprocessor zijn altijd hoogimpedant als $E=Q=0$, afgezien van de tijd t_{DHR} (zie fig. 11.3 en 11.4).

11.3.6 Bus Available, Bus Status

Het uitgangssignaal Bus Available BA geeft aan dat de bus beschikbaar is. De microprocessor heeft zijn R/W-, adres- en datazenders afgeschakeld (hoogimpedant gemaakt). Andere devices (bijvoorbeeld andere processoren of DMA-Controllers) mogen nu de bus gebruiken (busmaster worden).

Wordt BA gelijk aan 0, dan verklaart de microprocessor hiermee dat hij de bus gaat gebruiken. Alle andere devices moeten hun zenders afschakelen. De processor wacht echter nog 1 klokcyclus alvorens de bus daadwerkelijk te gebruiken door zijn zenders te activeren. Op deze manier wordt voorkomen, dat er twee zenders tegelijk actief zijn op een buslijn. De cyclus gedurende welke geen enkele zender op de bus actief is, heet dode cyclus (zie ook paragraaf 10.13).

Het uitgangssignaal Bus Status BS geeft samen met BA de toestand van de microprocessor aan (zie fig. 11.5).

Normal of running mode

Dit is de gewone toestand, waarbij de processor busmaster is en bezig is met het uitvoeren van een programma.

BA	BS	MPU-toestand
0	0	normal (running)
0	1	interrupt/reset acknowledge
1	0	synchronisation acknowledge
1	1	halt/bus grant acknowledge

Fig. 11.5 De toestanden van de microprocessor

Interrupt/reset acknowledge

Deze toestand treedt op tijdens het ophalen van de interruptvector door de microprocessor (zie ook paragraaf 9.3). Deze toestand geldt gedurende de twee opeenvolgende klokcycli, tijdens welke het hoog-deel en het laag-deel van het startadres van de interrupt service routine worden opgehaald.

Daar de adressen van de interruptvectoren alleen verschillend zijn voor wat betreft A3, A2, A1 en A0 (zie paragraaf 9.7.9), kunnen deze signalen worden gebruikt om te bepalen om welke interrupt acknowledge het gaat. A0 geeft hierbij aan of het om het hoog-deel (A0 = 0), dan wel het laag-deel (A0 = 1) van het startadres gaat. Fig. 11.6 laat de decodeerschakeling zien voor de IRQ-acknowledge.

Om te voorkomen dat er ten onrechte een interrupt acknowledge gegeven wordt (bijvoorbeeld als BA BS van 00 naar 11 gaat), moet IRQACK pas worden vrijgegeven als BA en BS stabiel zijn. Dit is het geval bij de opgaande flank van Q (zie fig. 11.3). Deze flank kan derhalve worden gebruikt voor het inlezen van de acknowledge in een flipflop. Deze flipflop is niet nodig als het device het signaal interrupt acknowledge alleen gebruikt als IRQAK stabiel is. Bij het veranderen van BA, BS en Ai kunnen korte pulsen op IRQAK ontstaan.

IRQACK kan worden gebruikt om de IRQ-aanvrager te melden dat zijn interruptaanvraag wordt afgehandeld. De aanvrager dient hierop zijn aanvraag te beëindigen.

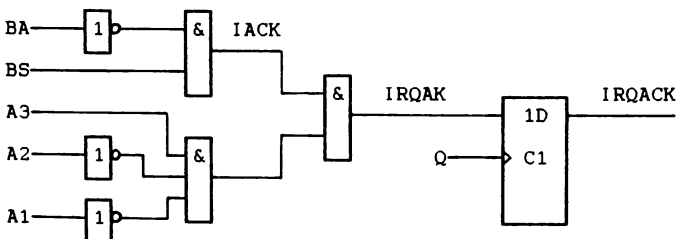


Fig. 11.6 Decodeerschakeling voor IRQ-acknowledge

Bij meerdere IRQ-aanvragers kan IRQACK in een daisy-chain worden toegepast (zie ook paragraaf 9.6). De aanvragers kunnen

eventueel zelf de interruptvector leveren (en wel het hoog-deel als $A0=0$ en het laag-deel als $A0=1$). Hierbij moet het geheugen gedisabled zijn. Dit kan eenvoudig geschieden met behulp van het signaal $\overline{IACK}$ (de inverse van $IACK$). In paragraaf 11.4 laten we zien hoe de interruptvectoren worden gegenereerd met behulp van een priority interrupt controller.

Op dezelfde wijze als in fig. 11.6 kunnen ook interrupt acknowledge signalen worden gerealiseerd voor de andere interrupts.

Synchronization acknowledge

De processor is in de sync-mode, indien hij wacht op externe synchronisatie door een hardware-interrupt.

De processor komt in deze toestand na het decoderen van een SYNC-instructie (zie paragraaf 9.7.11).

De sync-mode wordt verlaten na het detecteren van een interrupt (die tenminste gedurende 1 klokcyclus aanwezig moet zijn). De processor voert daarna de volgende instructie uit, tenzij er een NMI of een RESET is gedetecteerd of een niet-gemaskeerde IRQ of FIRQ die langer duurt dan 3 klokcycli. In dat geval wordt eerst de bijbehorende interrupt service routine uitgevoerd.

In sync-mode zijn de $R/\overline{W}$ -, adres- en datazenders van de bus ontkoppeld. Nadat BA weer 0 is geworden (na het detecteren van een interrupt), duurt het nog één klokcyclus, voordat de microprocessor deze zenders weer activeert. Er treedt dus een dode cyclus op.

Halt/bus grant acknowledge

Ook in deze mode heeft de microprocessor zijn $R/\overline{W}$ -, adres- en datazenders van de bus afgeschakeld. De microprocessor geeft met $BA=BS=1$ te kennen, dan een ander device busmaster mag worden. Ook hier treedt na het 0 worden van BA een dode cyclus op.

De microprocessor komt in deze toestand na het 0 worden van de ingangen $\overline{DMA/BREQ}$ of $\overline{HALT}$.

11.3.7 DMA/Bus Request

Indien de $\overline{DMA/BREQ}$ -ingang tenminste 200 ns voor de neergaande flank van E gelijk aan 0 is geworden, schakelt de MC6809 aan het einde van de betreffende klokcyclus in de DMA-mode. Dit hoeft niet te gebeuren aan het einde van een instructie! De interne klokken worden stopgezet, maar de externe klokken E en Q lopen gewoon door. Doordat de interne klokken tijdens DMA stilstaan, verandert er binnen in de processor niets, zodat de uitvoering van de instructie weer kan worden voortgezet, als de DMA-mode is beëindigd. Vandaar dat een DMA-aanvraag aan het

einde van elke klokcyclus kan worden gehonoreerd, ook als dit niet samenvalt met het einde van een instructie!

Als de interne klokken stilstaan, kan de processor geen self-refresh doen. Om nu te voorkomen dat de data in de microprocessorregisters verloren gaat bij een langdurige DMA-toestand, vraagt de microprocessor na 15 klokcycli automatisch de besturing van de bus terug. Dit doet hij door 15 cycli na het zetten van BA en BS deze uitgangen weer 0 te maken.

1 cyclus later worden de zenders van R/ $\bar{W}$, adres en data weer geactiveerd, zodat er een dode cyclus is ontstaan, waarin de vorige busmaster zijn zenders van de bus moet loskoppelen. Na deze dode cyclus volgt de self-refresh-cyclus van de microprocessor. Tijdens deze cyclus werkt de processor tevens aan het programma, waar hij mee bezig was. Daarna schakelt hij opnieuw naar de DMA-mode, mits $\overline{\text{DMA/BREQ}}$ gelijk aan 0 is. 1 cyclus later activeert de nieuwe busmaster zijn zenders, zodat ook hier een dode cyclus is ontstaan, waarin de processor de gelegenheid krijgt om zijn zenders hoogimpedant te maken (zie ook paragraaf 10.13).

Op deze wijze blijven er 14 aaneengesloten klokcycli over voor het busgebruik door een andere busmaster. De eerste cyclus tijdens DMA-mode is immers een dode cyclus en nadat gedurende 15 klokcycli BA en BS hoog zijn geweest, treedt weer een dode cyclus op. De processor is zodoende 16 klokcycli lang geen busmaster. Gedurende die 16 cycli staan de interne klokken stil.

De 14 zogenaamde DMA-cycli kunnen worden gebruikt voor het overdragen van data buiten de processor om (direct memory access) of voor het refreshen van een extern dynamisch geheugen.

Een DMA/Bus Request wordt zelfs gehonoreerd als $\overline{\text{RESET}} = 0$! Een reset kan de DMA-mode niet onderbreken!

De DMA-mode wordt (behalve voor self-refresh) verlaten, indien $\overline{\text{DMA/BREQ}}$ tenminste 200 ns voor de neergaande flank van E hoog is.

11.3.8 Halt

Als de $\overline{\text{HALT}}$ -ingang 0 is (bij de MC6809 op tenminste 200 ns voor de neergaande flank van Q), stopt de microprocessor met het uitvoeren van het programma aan het einde van de huidige instructie. De halt-toestand kan oneindig lang gehandhaafd blijven, zonder dat er informatie in de processor verloren gaat. De inwendige klokken en E en Q lopen gewoon door.

De halt-toestand wordt verlaten, als $\overline{\text{HALT}}$ hoog is op tenminste 200 ns voor de neergaande flank van Q.

In de halt-toestand reageert de processor niet op interrupts, hoewel RESET en NMI in interne flipflops worden opgeslagen voor latere afhandeling. De microprocessor komt zelfs in de halt-toestand als $\overline{\text{HALT}}$ gelijk aan 0 wordt, terwijl $\overline{\text{RESET}} = 0$ is.

Een DMA/Bus request wordt altijd gehonoreerd! In de halt-mode zijn BA en BS reeds geset, zodat DMA grant actief is. Is $\overline{\text{DMA/BREQ}}$ gelijk aan 0 als $\overline{\text{HALT}}$ laag wordt, dan zal de microprocessor het einde van de huidige instructie bereiken (door af en toe gedurende een of meerdere klokcycli busmaster te worden). Daarna komt de processor in de halt-mode. De interne klokken lopen nu door.

De halt-mode kan worden gebruikt bij

- direct memory access
 - memory refresh
 - multi-processorsystemen
- Slechts één processor is in running mode.
- hardware en software debugging
- De processor kan telkens worden gestopt, zodat de instructies in een zogenaamde step-by-step mode kunnen worden uitgevoerd.

11.3.9 Memory Ready

De ingang MRDY kan worden gebruikt om de microprocessor te laten communiceren met langzame geheugens. Dit gebeurt door de klokcyclus te vergroten.

Als MRDY (bij de MC6809) tenminste 200 ns voor de neergaande flank van E gelijk is aan 0, dan wordt deze neergaande flank van E, alsmede de daaropvolgende opgaande flank van Q, uitgesteld totdat MRDY weer terugkeert naar het 1-niveau (zie fig. 11.7). Dit geldt eveneens voor de interne klokken van de microprocessor. Op deze manier wordt de accesstijd vergroot. Het verlengen van de klokcyclus gebeurt met veelvoud van de periode van de externe oscillator (of van het kristal). Let hierbij op dat de cyclustijd niet groter mag worden dan 16 μs . De processor kan de klok niet laten doorlopen als $\text{MRDY}=0$. Is MRDY te lang laag, dan gaat de data in de processorregisters verloren!

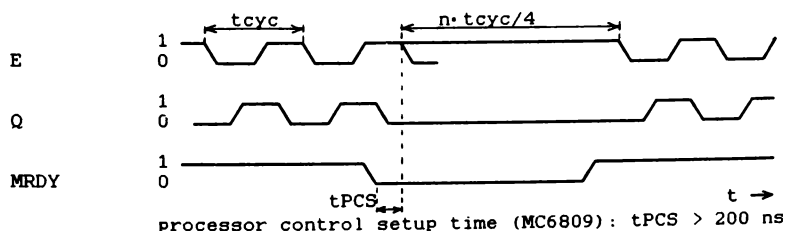


Fig. 11.7 Timing van MRDY

Het signaal MRDY kan worden afgeleid uit het adres. Wordt een langzaam device geadresseerd, dan moet MRDY 0 worden gemaakt. Is het device klaar met het opbergen van de data (bij MPU write) of met het gereed zetten van de data op de databus (bij MPU read), dan moet MRDY worden geset.

De klokcyclus kan niet worden vergroot, als de microprocessor de bus niet gebruikt tijdens het uitvoeren van instructies. Dit is gedaan om te voorkomen dat de processorodeloos wordt vertraagd als hij intern bezig is met rekenwerk.

Als de besturing van de bus (met behulp van HALT of DMA/BREQ) is overgedragen aan een ander device (bijvoorbeeld een DMAC), dan kan MRDY de klokcyclus wel verlengen. Zodoende is het mogelijk met langzame geheugens of I/O-devices direct memory access te plegen.

Bij gelijktijdig gebruik van DMA/BREQ en MRDY moet er extra zorg worden besteed om te voorkomen dat de klokperiode groter wordt dan 16 μ s. In dit geval moet MRDY in ieder geval weer 1 worden voor het begin van de dode cyclus, voorafgaand aan de self-refresh-cyclus van de microprocessor.

11.3.10 Hardware interrupts

De hardware interrupts worden ingelezen op de neergaande flank van de klok Q. Ze moeten (bij de MC6809) tenminste 200 ns voor de neergaande flank van Q de gewenste waarde hebben. Voor het effect van de interrupts wordt verwezen naar paragraaf 9.7.

RESET

Een 0 op de RESET-ingang gedurende tenminste 1 klokcyclus, zal de microprocessor resetten. Zolang RESET=0, staat er \$FFFE op de adresbus en geldt BA=BS=0. Nadat RESET gelijk aan 1 is geworden, haalt de processor de interruptvector op uit de locaties \$FFFE en \$FFFF. Gedurende het ophalen van de interruptvector (dus gedurende 2 klokcycli) is BA=0 en BS=1. Bij het inschakelen van de voedingsspanning moet RESET laag blijven, totdat de klokoscillator correct functioneert (zie paragraaf 11.3.5).

NMI

Een neergaande flank op NMI start een non-maskable interrupt. NMI moet echter tenminste 1 klokcyclus van E laag zijn teneinde gedetecteerd te worden.

De NMI-interrupt wordt opgeslagen in een flipflop, mits de stack pointer S geladen is (zie paragraaf 9.7.4). De interrupt wordt uitgevoerd, zodra de huidige instructie beëindigd is. Tussen de neergaande flank van RESET en de opgaande flank van BS (RESET Acknowledge) wordt NMI niet herkend en evenmin opgeslagen in een flipflop.

FIRQ

Een 0 op de $\overline{\text{FIRQ}}$ -ingang initialiseert een fast interrupt (mits de F-bit in het condition code register gereset is). Om herkend te worden, moet $\overline{\text{FIRQ}}$ tenminste 1 klokcyclus laag zijn en laag blijven tot het einde van de huidige instructie.

IRQ

Een laag niveau op de $\overline{\text{IRQ}}$ -ingang leidt het afhandelen van een interrupt in (mits de I-bit in het condition code register 0 is). Om herkend te worden, moet $\overline{\text{IRQ}}$ tenminste 1 klokcyclus 0 zijn, evenals 0 blijven tot de huidige instructie is uitgevoerd.

11.3.11 Synchronisatie

Wordt niet aan de gespecificeerde setup-tijd en hold tijd van een signaal voldaan, dan kan in principe het volgende gebeuren, indien dit signaal niet door de microprocessor zelf wordt gesynchroniseerd:

1. Het signaal wordt toch gedetecteerd en heeft de gewenste actie tot gevolg.
2. Het signaal wordt aanvankelijk niet gedetecteerd. In dit geval zal het signaal bij de volgende actieve klokflank worden gedetecteerd. Ten opzichte van die volgende flank is wel aan de setup tijd voldaan. Het signaal is hier reeds ongeveer een hele klokperiode geldig. De gewenste actie wordt nu alsnog uitgevoerd, zij het een klokperiode later dan in geval 1. Dit is geen enkel probleem, immers was het signaal in geval 1 net na de hold tijd, dus iets later geldig geweest, dan was ook hier de betreffende actie na de volgende actieve klokflank uitgevoerd.
3. De microprocessor komt in een ongedefinieerde toestand. Dit is natuurlijk niet acceptabel. Om een ongedefinieerde toestand te voorkomen, moet het signaal extern moeten worden gesynchroniseerd op de microprocessorklok.

Bij signalen, die binnen de processor worden gesynchroniseerd, hoeft eigenlijk niet aan de setup tijd en hold tijd te worden voldaan. In dit geval kan de processor immers niet in een ongedefinieerde toestand komen. Alleen de situaties 1 en 2 kunnen hier optreden. Wil men er echter zeker van zijn dat de gewenste actie meteen na de huidige actieve flank van de klokpuls uitgevoerd wordt, dan moet het signaal wel voldoen aan de setup tijd en aan de hold tijd van het signaal ten opzichte van die klokflank.

De specificatie zou duidelijk moeten vermelden welke signalen intern worden gesynchroniseerd en welke niet. Dit gebeurt helaas niet altijd.

Bij de MC6809 worden de interruptsignalen en MRDY intern gesynchroniseerd.

Voor $\overline{\text{DMA/BREQ}}$, $\overline{\text{HALT}}$ en Di gebeurt dit niet. Deze signalen

moeten derhalve extern met behulp van extra hardware worden gesynchroniseerd, opdat aan de setup tijd en aan de hold tijd wordt voldaan.

11.3.12 Signalen van de MC6809E

De MC6809, MC68A09 en MC68B09 hebben elk een E-versie. Deze is vooral bedoeld voor gebruik in multiprocessorsystemen. Hiertoe is hij uitgerust met enkele extra signalen (LIC, BUSY en AVMA) voor informatie over de processorstatus. De signalen MRDY, E, Q, XTAL EXTAL en $\overline{\text{DMA/BREQ}}$ zijn niet aanwezig. In plaats daarvan kent de MC6809E de signalen BUSY, LIC, Ein, Qin, AVMA en TSC. De signalen BUSY, LIC en AVMA zijn geldig op tCD na de opgaande flank van Q ($t_{CD} < 300 \text{ ns}$).

De instructiesets van al deze processoren zijn identiek.

Klokingangen

De MC6809 heeft geen interne klokoscillator. Derhalve moeten de systeemklokken E en Q door een externe oscillator worden opgewekt en worden aangesloten op de ingangen Ein en Qin van de microprocessor. Op deze wijze loopt de microprocessor synchroon met een externe klok. Dit is vooral belangrijk in een multiprocessorsysteem, waarin de diverse processoren synchroon moeten lopen.

Omdat de MC6809E geen interne oscillator heeft, is een MRDY-signaal zinloos. De MC6809E kan de klokcyclus niet oprekken. Het signaal MRDY is derhalve niet op deze processor aanwezig. Het oprekken van de klokcyclus moet gebeuren door de klokgenerator.

De signalen Ein en Qin moeten dezelfde eigenschappen hebben als de signalen E en Q van de MC6809 (zie fig. 11.3). Er zijn enkele kleine verschillen:

- De breedten van de 0-puls van E en van de 1-pulsen van E en Q moeten liggen tussen 450 ns en 9500 ns.
- De afstand tussen twee willekeurige opeenvolgende flanken van E en Q moet tenminste 200 ns bedragen.
- Het tijdstip waarop het nieuwe adres en de signalen $R/\overline{W}$, BA en BS geldig zijn, is niet meer gespecificeerd ten opzichte van de opgaande flank van Q, maar ten opzichte van de neergaande flank van E. De vertragingstijd tussen de neergaande flank van E en deze uitgangssignalen bedraagt maximaal 200 ns.

Three-state control

De ingang TSC vervangt in feite de ingang $\overline{\text{DMA/BREQ}}$ van de MC6809. TSC=1 zorgt ervoor dat adres-, $R/\overline{W}$ - en data-uitgangen van de microprocessor in de hoogimpedante toestand gezet worden:

- Als Ein=0 beïnvloedt TSC deze uitgangen onmiddellijk.

- Als $\text{Ein}=1$ heeft TSC geen invloed op de betreffende uitgangen.
- Als TSC nog 1 is tijdens de opgaande flank van Ein, blijven de genoemde uitgangen hoogimpedant tot aan het einde van de klokcyclus (tot aan de neergaande flank van Ein).

TSC wordt gebruikt ten behoeve van direct memory access en dynamic memory refresh en bij multiprocessorsystemen. De klokken Ein en Qin moeten worden stilgezet om te voorkomen dat de processor verder gaat met het ophalen en uitvoeren van instructies, terwijl de buszenders hoogimpedant zijn. Het stoppen van de klokken moet gebeuren aan het begin van de klokcyclus, waarbij $\text{TSC}=1$ is (zie fig. 11.8). De klokken op de processoringenangen moeten dus worden stilgezet als $\text{Ein}=\text{Qin}=0$. De externe klokgenerator moet derhalve ook worden aangestuurd door het signaal TSC.

Als de klokgenerator behalve de signalen Ein en Qin ook het signaal TSC levert, is altijd aan de voorwaarde voldaan, dat $\text{TSC}=1$ wordt als $\text{Ein}=\text{Qin}=0$. De klokken moeten hiertoe (met behulp van een stuursignaal) alleen gestopt kunnen worden op het 0-niveau.

De datazenders van de processor zijn altijd hoogimpedant als $\text{Ein}=\text{Qin}=0$ (zie fig. 11.8).

Specificatie van de TSC-timing van de MC6809E (zie fig. 11.8): de referentiepunten zijn 0,8 V voor het 0-niveau en 2,0 V voor het 1-niveau:

- $t_{\text{PCS}} > 200 \text{ ns}$ (TSC setup time)
- $t_{\text{TSD}} < 120 \text{ ns}$ (TSC delay time)
- $t_{\text{TSV}} < 210 \text{ ns}$ (TSC to valid level time)
- $t_{\text{TSR}} < 200 \text{ ns}$ (TSC release to high impedance time)
- $t_{\text{DDQ}} < 200 \text{ ns}$ (Data delay time from Q)

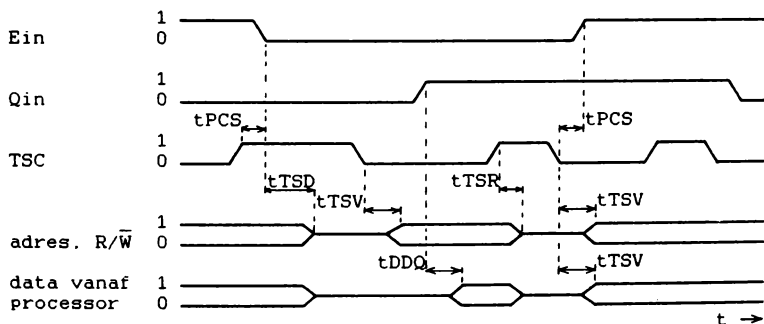


Fig. 11.8 Tijddiagram van Three State Control

Last instruction cycle

De output LIC is 1 gedurende de laatste klokcyclus van elke instructie (zie fig. 11.9). De eerste cyclus na het laag worden

van LIC is de cyclus, waarin de processor de volgende opcode ophaalt uit het geheugen. Het signaal LIC kan in een multiprocessorsysteem worden gebruikt om aan de externe busarbitration logica te melden, dat de processor bijna klaar is met het uitvoeren van een instructie. Dit betekent dat de besturing van de bus aan het begin van de volgende klokcyclus eventueel aan een andere processor kan worden overgedragen.

Er is in een multiprocessorsysteem extra hardware nodig die als taak heeft om de busbesturing toe te wijzen aan een van de processoren. Deze hardware wordt de busarbitration logica genoemd. Hij treedt op als scheidsrechter tussen de diverse processoren.

LIC is ook 1 als de microprocessor zich in de halt-toestand bevindt aan het einde van een instructie (niet bij CWA1 en RESET), in de sync-toestand en tijdens het automatisch op de stack opbergen van de registerinhouden bij een interrupt. In de halt- en in de sync-toestand voert de microprocessor geen instructies uit. De besturing van de bussen kan derhalve worden overgedragen aan een ander device.

Ook bij het automatisch op de stack opbergen van registerinhouden wordt geen echte instructie uitgevoerd. Vandaar dat in deze gevallen LIC=1 is. De processor mag bij het 0 worden van LIC worden onderbroken, opdat een ander device busmaster wordt.

BUSY

Het uitgangssignaal BUSY is gelijk aan 1

- tijdens de toegang tot de eerste byte bij een bewerking van 2-bytes getallen (bijvoorbeeld bij STU, LDY, SUBD), behalve bij immediate addressing
- bij het ophalen van de high-byte van het effectieve adres bij een indirect addressing instructie (bijvoorbeeld bij LDA [\$24A0], JSR [-28,X])
- bij het ophalen van de high-byte van een interruptvector (bij een hardware of software interrupt)
- gedurende de read- en modify-cycli van de zogenaamde read-modify-write instructies (ASL, ASR, LSL, LSR, ROL, ROR, CLR, INC, DEC, COM en NEG)

In multiprocessorsystemen wordt het signaal BUSY gebruikt voor het uitstellen van het aanwijzen van een andere busmaster (als BUSY=1). Zo wordt voorkomen dat de bovengenoemde operaties verkeerd kunnen aflopen.

2-bytes gegevens

Bij de eerste drie situaties gaat het telkens om 2-bytes gegevens in het datageheugen (uitgezonderd bij de immediate addressing instructies, waar de data zich in het programmeergeheugen bevindt).

Stel dat de high-byte van een gegeven wordt binnengehaald.

Vervolgens wordt de besturing van de bussen door de busarbitration logica aan een andere processor overgedragen. De busarbitration logica geeft de besturing weer aan de eerste processor terug, die vervolgens de low-byte binnenhaalt. Zo is er een totaal verkeerd gegeven geladen. Immers de high-byte en de low-byte horen niet bij elkaar.

Doordat de waarde van BUSY alleen verandert na de opgaande flank van Q (binnen maximaal 300 ns) zal BUSY telkens 1 zijn aan het begin van de klokcyclus, waarin de low-byte van de bedoelde 2-bytes gegevens wordt opgehaald of weggeschreven. Dit heeft tot gevolg dat het transporteren van de genoemde 2-bytes gegevens niet kan worden onderbroken, mits de busarbitration logica de busbesturing alleen overdraagt aan een ander device als BUSY=0 is! Dan is het niet mogelijk dat een verkeerd getal wordt verwerkt.

Read-modify-write instructies

Read-modify-write instructies zijn instructies, waarbij binnen één en dezelfde instructie de volgende acties worden ondernomen:

1. de inhoud van een geheugenlocatie wordt gelezen
2. de gelezen waarde wordt gewijzigd
3. de veranderde waarde wordt op dezelfde locatie teruggeschreven.

Bij INC \$A5DD leest de microprocessor eerst de inhoud van de geheugenlocatie \$A5DD (read), verhoogt deze waarde daarna (in de Arithmetic and Logic Unit) met 1 (modify) en schrijft tenslotte de nieuwe waarde terug op de geheugenlocatie met het adres \$A5DD (write).

Doordat BUSY hoog is tijdens het uitvoeren van dit soort instructies, kan bij multiprocessorsystemen worden voorkomen, dat de ene processor een flag leest en modificeert, die door een andere processor tegelijkertijd wordt gewijzigd.

Bij de instructie INC \$A5DD wordt BUSY geset bij het lezen van (\$A5DD). BUSY blijft 1 tot na het wijzigen van de gelezen waarde in de processor (zie fig. 11.9). In fig. 11.9 is aangenomen dat de increment-instructie zich op de geheugenlocaties \$3140 t/m \$3142 bevindt, en dat (\$A5DD) = \$E7. De extended addressing instructie INC heeft als opcode \$7C. \$FFFF op de adresbus geeft aan dat de processor intern bezig is en de bussen niet gebruikt.

Gedurende klokcyclus c+4, dus bij het lezen van (\$A5DD) wordt BUSY geset. De processor wordt niet meer onderbroken (door de busarbitration logica), totdat BUSY gereset is. Dit gebeurt na het wijzigen van de binnengehaalde waarde, dus tijdens klokcyclus c+6. In deze klokcyclus wordt de veranderde waarde opgeborgen in het geheugen. Aan het einde van deze cyclus kan de besturing pas worden overgedragen aan een ander device. Op deze wijze wordt voorkomen dat eenzelfde geheugenlocatie door twee verschillende devices tegelijkertijd wordt gewijzigd, mits de busarbitration logica de busbesturing alleen overdraagt aan een ander device als BUSY=0 is!.

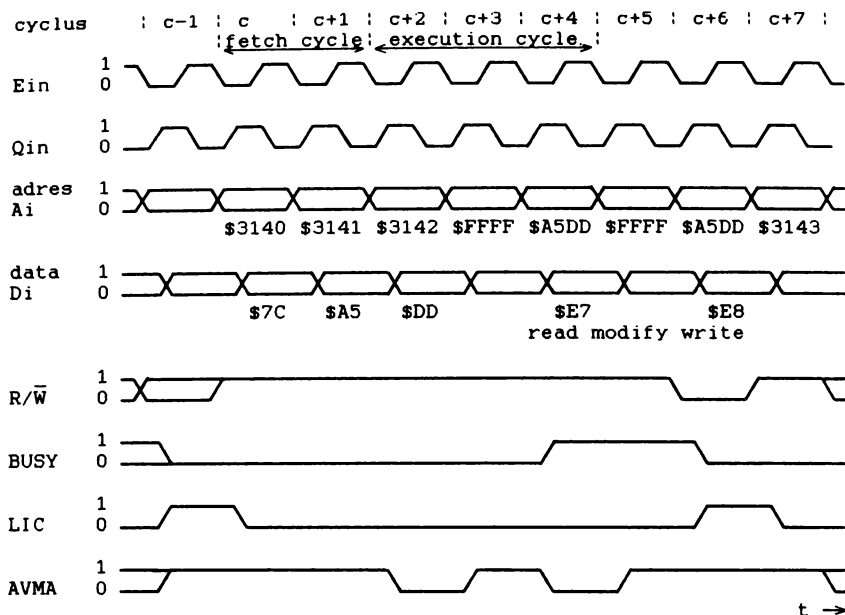


Fig. 11.9 Timing van het signaal BUSY (alsmede van LIC en AVMA) bij de instructie INC \$A5DD

Deze niet te onderbreken read-modify-write instructies worden toegepast bij semaforen (zie ook paragraaf 10.14.7). Stel op een gegeven moment is er nog één plaats vrij in een vliegtuig naar Madrid. Nu willen twee reizigers ongeveer tegelijkertijd een reis naar Madrid boeken met het bewuste vliegtuig. Ze staan aan verschillende loketten. Via het ene loket wordt in de computer gekeken of er nog plaats is. De lokettiste geeft aan de reiziger te kennen dat er inderdaad nog plaats is en vraagt zijn naam, adres, enzovoorts om de reis te boeken. Intussen heeft echter ook de andere lokettiste gekeken of er nog plaats is, hetgeen inderdaad het geval was. Nu zullen beide lokettisten dezelfde plaats reserveren en beide zullen ervoor zorgen dat het bit "RESERVED" voor de betreffende plaats geset wordt, zodat later niet nog iemand dezelfde plaats reserveert. Het kwaad is echter reeds geschiedt.

Het hierboven beschreven probleem is op te lossen met behulp van semaforen.

Een semafoor geeft aan dat een bepaald deel van het gemeenschappelijk geheugen in bewerking is door een processor en dus niet door andere processoren mag worden gebruikt. Dit is te vergelijken met een seinpaal, die aangeeft of er een trein op een bepaald baanvak rijdt. Is dit het geval (het sein staat

hoog), dan moeten andere treinen wachten met het berijden van dat baanvak, totdat het sein veilig (laag) is.

Een semafoor moet door alle processoren in een multiprocessorsysteem gelezen en geset kunnen worden. Een semafoor moet zich derhalve op een locatie in het gemeenschappelijke geheugen bevinden.

Bij multiprocessorsystemen kan het volgende probleem optreden. Een bepaalde processor test of een geheugenblok vrij is voor gebruik. Hij leest hiertoe op een bepaald moment de betreffende semafoor (een flag in het gemeenschappelijke geheugen). Is de semafoor gereset, dan is het geheugenblok toegankelijk. De processor zal nu de semafoor zetten om kenbaar te maken dat hij het geheugenblok gaat bewerken en dat het derhalve niet door andere processoren mag worden gebruikt. Tijdens het testen en zetten van de semafoor gebruikt de processor de bus niet. In deze tijd kan een andere processor de bus gebruiken om bijvoorbeeld diezelfde semafoor te testen. Deze tweede processor zal ook constateren dat het geheugenblok vrij is, niet wetende dat de eerste processor juist bezig is met het zetten van de semafoor.

Nu zouden twee processoren tegelijkertijd eenzelfde geheugenblok gaan gebruiken, zodat er met verkeerde, incomplete informatie gewerkt gaat worden.

Dit kan worden voorkomen door de semafoor te testen en te zetten met een niet te onderbreken test-and-set instructie. Bij deze instructie wordt de semafoor eerst gelezen en daarna binnen dezelfde, niet te onderbreken instructie geset. Is de semafoor bij het testen reeds geset, dan betekent dit dat de betreffende processor moet wachten. Het geheugengebied is immers in bewerking door een andere processor. Was de semafoor gereset, dan wordt hij geset om aan andere processoren kenbaar te maken dat het geheugengebied bewerkt wordt en dat voor hen de toegang verboden is. Is de processor klaar met het bewerken van het geheugengebied, dan reset hij de semafoor en kan een andere processor toegang krijgen.

Doordat de test-and-set instructie niet kan worden onderbroken, is het niet mogelijk dat meerdere processoren tegelijk de semafoor testen en zetten en derhalve tegelijk toegang krijgen tot het bewuste geheugengebied.

De processoren uit de MC68000-familie hebben zo'n instructie, de MC6809(E)-processoren echter niet.

Nu kan hetzelfde worden bereikt met een niet te onderbreken read-modify-write-instructie, zij het op een omslachtigere manier. Fig. 11.10 laat de te volgen procedure zien.

In de aanvangstoestand is de semafoor 0. Wil een bepaalde processor het gemeenschappelijke geheugengebied bewerken, dan moet hij eerst de semafoor verhogen en daarna pas testen. Bij de omgekeerde volgorde houden we het eerder genoemde probleem.

dat twee processoren beide de semafoor testen en beide tot de ontdekking komen, dat het gebied nog vrij is.
Na het verhogen kan de semafoor 1 zijn of hoger.

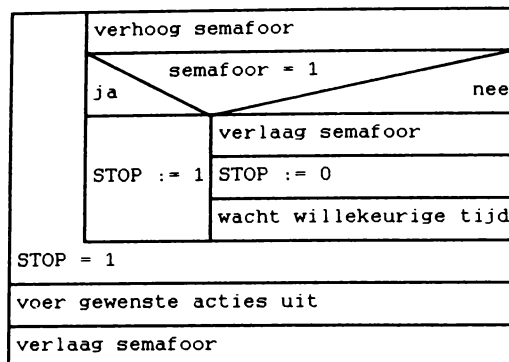


Fig. 11.10 Procedure bij semaforen met behulp van niet te onderbreken read-modify-write instructies

Is de semafoor 1, dan is dit veroorzaakt door de processor die deze 1 detecteert. Deze processor kan vervolgens de gewenste acties in het betreffende geheugengebied gaan uitvoeren. Er is immers door het verhogen van de semafoor aan alle andere processoren reeds medegedeeld, dat het geheugengebied in bewerking is.

Zijn de betreffende acties uitgevoerd, dan verlaagt de processor de semafoor. Hiermee wordt het geheugengebied vrijgegeven voor gebruik door andere processoren.

Is de semafoor hoger dan 1, dan hebben meerdere processoren deze verhoogd en moet de processor die dit detecteert, de semafoor weer verlagen. Na enige tijd verhoogt hij de semafoor opnieuw en test daarna nogmaals de waarde.

De random (willekeurige) wachttijd wordt slechts gebruikt om de kans te verkleinen dat bij de volgende test weer meerdere processoren tegelijk de semafoor hebben verhoogd. Dit wordt herhaald totdat de semafoor bij de test de waarde 1 heeft. Op deze manier kan het nooit voorkomen dat twee processoren beide ontdekken dat de semafoor de waarde 1 heeft. Dit is te danken aan het niet onderbreken van de read-modify-write-operatie bij de instructie INC SEMAFO. Hiertoe moet vanzelfsprekend gelden dat de busarbitration logica de busbesturing alleen overdraagt aan een andere processor, indien BUSY=0 is!

Dit niet onderbreken van de read-modify-write is essentieel. Zou de onderbreking wel mogelijk zijn, dan bestaat het gevaar dat de semafoor (die de waarde 0 heeft) door een processor

wordt binnengehaald, teneinde hem te verhogen. Vervolgens wordt die processor onderbroken en wordt de semafoor (die nog 0 is) door een andere processor gelezen en verhoogd (dus 1 gemaakt). Hierna wordt de waarde die door de eerste processor was binnengehaald (en die toen nog 0 was), eveneens verhoogd (en dus 1 gemaakt). Deze 1-waarde wordt nogmaals opgeborgen in de semafoor. Nu testen beide processoren de semafoor en beide vinden ze de waarde 1. Dit betekent dat beide processoren in het geheugengebied gaan werken.

Het programma voor het zetten en testen van de semafoor ziet er als volgt uit:

```

10 PROGR    ---
20          ---
30          BSR      SEMA          set en test semafoor
40          ---          voer gewenste geheugen-
50          ---          bewerkingen uit
60          DEC      SEMAFO        verlaag semafoor (geef geheugen
70 *          ---          vrij)
80          ---
90          ---
100         SWI
110 *
120 *****
130 * Subroutine SEMA *
140 *****
150 *
160 *De subroutine SEMA verhoogt de semafoor SEMAFO.
170 *Er wordt pas teruggekeerd naar het aanroepende programma,
180 *indien de semafoor 1 is (dus indien het geheugengebied
190 *vrij is)
200 *
210 *
220 STOP     RMB      1
230 *
240 SEMA     INC      SEMAFO
250         LDA      SEMAFO          spring als
260         CMPA     #1              de semafoor gelijk is aan 0
270         BNE      BEZET          (geheugen is bezet)
280 VRIJ     LDA      #1
290         STA      STOP
300         BRA      TEST
310 BEZET    DEC      SEMAFO
320         CLR      STOP
330         BSR      DLYRAN          wacht willekeurige, random tijd
340 TEST     TST      STOP
350         BEQ      SEMA
360         RTS
370 *
380 DLYRAN    ---
390          ---

```

400	RTS	
410	END	PROGR

De locatie SEMAFO voor de opslag van de semafoor is een geheugenlocatie, die voor elke processor op elk moment toegankelijk is.

Advanced valid memory address

Het 1 worden van de uitgang AVMA geeft aan dat de microprocessor tijdens de volgende klokcyclus de bus wil gaan gebruiken (zie fig. 11.9). De busarbitration logica kan dit signaal gebruiken om de processor de besturing van de bussen te geven. Op deze wijze kan worden voorkomen, dat een processor de bussen te vroeg krijgt toegewezen. AVMA maakt zo efficiënte multiprocessorsystemen mogelijk.

AVMA is 0 in de halt- of sync-toestand van de processor. In deze gevallen gebruikt de processor de bussen immers niet en kan een ander device busmaster blijven.

11.4 Priority Interrupt Controller

Opbouw van de PIC

De Priority Interrupt Controller (MC6828) wordt gebruikt om interruptvectoradressen te vertalen (zie ook paragraaf 9.6, met name fig. 9.16 en 9.17). Als de microprocessor het startadres van de interrupt service routine ophaalt, verandert de PIC het vectoradres. Het vectoradres is het adres van de locatie, waar de microprocessor het startadres van de interrupt service routine vandaan haalt. Op deze wijze zorgt de PIC ervoor, dat elke interruptaanvrager zijn eigen interruptvector krijgt.

Op de PIC kunnen 8 interruptaanvragers worden aangesloten. Deze krijgen een prioriteit, afhankelijk van de ingang van de PIC, waarop de aanvrager is aangesloten. Fig. 11.11 toont de opbouw van de MC6828.

Interrupt IN7 heeft de hoogste prioriteit en IN0 de laagste.

Selected mode

De PIC moet alleen een adresvertaling doen, als de microprocessor de interruptvector adresseert. De PIC werkt in de zogenaamde selected mode, als

$$A4 \cdot A3 \cdot \overline{A2} \cdot \overline{A1} \cdot CS1 \cdot \overline{CS0} \cdot R/\overline{W} = 1,$$

dus als de processor de IRQ-vector ophaalt.

Hiertoe moeten natuurlijk de Chip Select ingangen CS1 en $\overline{CS0}$ geactiveerd zijn, terwijl er door de processor op het vectoradres gelezen wordt, dus $R/\overline{W}=1$ en $A4A3A2A1 = 1100$.

Het gaat hierbij om de IRQ-interrupt, aangezien de IRQ-vector opgeslagen is op de adressen \$FFF8 en \$FFF9. De chip select

ingangen moeten dus geactiveerd zijn als A5 t/m A15 gelijk aan 1 zijn. De vier bits A4=1, A3=1, A2=0 en A1=0 worden door de PIC vertaald en de nieuwe waarden worden via de uitgangen Z4, Z3, Z2 en Z1 op de adresbus gezet (zie fig. 11.12). Bedenk hierbij dat er vier bits nodig zijn voor het genereren van 8 nieuwe afwijkende interruptvectoren, behorende bij de 8 interruptingangen $\overline{\text{IN0}}$ t/m $\overline{\text{IN7}}$.

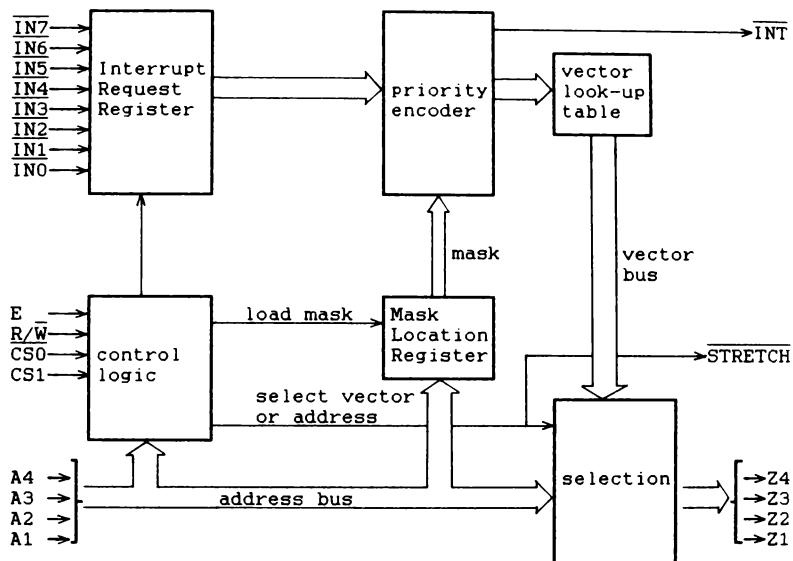


Fig. 11.11 Blokschema van de PIC

hoogste actieve en enabled interrupt	output, mits PIC in selected mode Z4 Z3 Z2 Z1				equivalent adres	vectoradres van
geen	1	1	0	0	\$FFF8/9	default routine
IN7	1	0	1	1	\$FFF6/7	IN7-routine
IN6	1	0	1	0	\$FFF4/5	IN7-routine
IN5	1	0	0	1	\$FFF2/3	IN7-routine
IN4	1	0	0	0	\$FFF0/1	IN7-routine
IN3	0	1	1	1	\$FFEE/F	IN7-routine
IN2	0	1	1	0	\$FFEC/D	IN7-routine
IN1	0	1	0	1	\$FFEA/B	IN7-routine
IN0	0	1	0	0	\$FFE8/9	IN7-routine

Fig. 11.12 Vertaaltabel van interruptvectoradres

De adresuitgangen A1 t/m A4 van de microprocessor moeten niet rechtstreeks met de adresbus verbonden worden, maar via de PIC

(zie fig. 11.13).

De overige adresuitgangen worden wel rechtstreeks met de adresbus verbonden.

A5 t/m A15 moeten tevens worden gebruikt om (via een externe adresdecoder) CS1 en $\overline{CS0}$ te activeren.

Bit A0 gaat buiten de PIC om, omdat deze bit alleen bepaalt of de high-byte (A0=0), dan wel de low-byte (A0=1) van het startadres wordt opgehaald. Deze adresbit wordt derhalve niet vertaald.

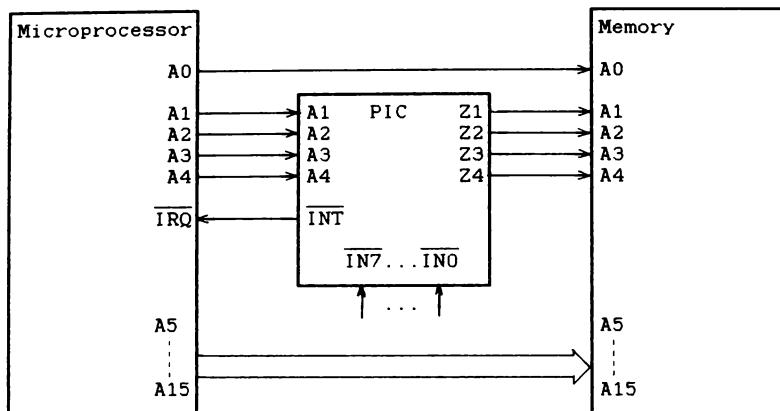


Fig. 11.13 Hardware vectoring door middel van vectoradresvertaling met de PIC (MC6828)

Bevindt de PIC zich niet in de selected mode, dan worden de adresingangen A1 t/m A4 overanderd aan de adresbus doorgegeven via de uitgangen Z1 t/m Z4.

Hiermee gaat wel een extra vertraging (van maximaal 75 ns) gepaard, zodat er minder accestijd over blijft voor het geheugen (zie ook paragraaf 11.3.5).

Default routine

Als de PIC in de selected mode staat, terwijl er geen interrupt is gegeneerd door de PIC, dan wordt het vectoradres (\$FFF8/9) ongewijzigd doorgegeven. Het gaat hierbij om een interruptaanvraag die niet via de PIC is aangesloten. Nu wordt de zogenaamde default routine uitgevoerd. Het is dus mogelijk om, buiten de PIC om, interrupts aan te sluiten op de $\overline{IRQ}$ -ingang van de processor. Deze interrupts hebben dan de allerlaagste prioriteit. Wordt er immers een (niet-geblokkeerde) interruptaanvraag gedaan via de PIC, dan zal deze het vectoradres veranderen, zodra de processor het bijbehorende startadres ophaalt. Zodoende komen de rechtstreeks op de processor aangesloten interrupts pas aan de beurt, als alle andere (niet-geblokkeerde) interrupts zijn afgehandeld.

Met behulp van de inhoud van het Mask Location Register kunnen bepaalde interruptaanvragen worden geblokkeerd. Is de inhoud van het Mask Location Register i , dan zijn de aanvragen beneden IN_i geblokkeerd. Voor $(MLR) = 0$ is geen enkele aanvraag geblokkeerd. Voor bijvoorbeeld $(MLR) = 5$ zijn de aanvragen IN_0 t/m IN_4 geblokkeerd. Voor $(MLR) > 7$ zijn alle interruptaanvragen geblokkeerd.

Interruptmasker laden

Het Mask Location Register wordt geladen op de neergaande flank van E , indien geldt: $CS1 \cdot CS0 = 1$ en $R/\bar{W} = 0$. Dit betekent dat er moet worden geschreven in een geheugenlocatie met een adres, waarbij geldt dat A_5 t/m $A_{15} = 1$ zijn (de PIC is geadresseerd). Dit betekent dat het Mask Location Register wordt geselecteerd met 32 opeenvolgende adressen (bepaald door A_0 t/m A_4). Als de processor de bus niet gebruikt is $A_1 = \$FFFF$ en $R/\bar{W} = 1$ (zie paragraaf 11.3.3 en 11.3.4). Bij het schrijven in de PIC moet het signaal $R/\bar{W}$ gelijk zijn aan 0. Dit betekent dat er niet ten onrechte in de PIC wordt geschreven, als de processor de bus niet gebruikt.

Daar de PIC geen data-ingangen heeft, wordt het Mask Location Register geladen met de informatie op de adresingangen A_1 t/m A_4 . De informatie op de databus is hierbij don't care.

Bij het schrijven in de PIC wordt het adres niet vertaald, zodat tevens het geheugen wordt geselecteerd. Hierbij kunnen problemen ontstaan:

Geval 1: De interruptvectoren staan in read only memory. Bij het schrijven in de PIC zal in dit geval ook de ROM zijn geadresseerd. Daar de ROM geen $R/\bar{W}$ -ingang heeft (hij kan alleen worden gelezen), zet hij data op de databus. De microprocessor zet bij het schrijven in de PIC eveneens data op de databus. Dit leidt tot hoge stromen in de buslijnen (zie ook paragraaf 10.2). De ROM moet derhalve geblokkeerd worden bij het opbergen van het interruptmasker.

Remedie:

Neem het signaal $R/\bar{W}$ op in de adresdecoder van de ROM, zodat de ROM alleen bij leesoperaties wordt geselecteerd en data op de databus zet.

Geval 2: De interruptvectoren staan in read/write memory. In dit geval wordt het read/write memory tegelijk met het Mask Location Register beschreven. Dit betekent dat bij het initialiseren van de interruptvectoren ook het interruptmasker van een bepaalde waarde wordt voorzien.

Remedie a:

Initialiseer het interruptmasker als laatste. Zet hierbij de juiste waarde van de interruptvector op de databus, omdat ook een van de interruptvectoren wordt geadresseerd. Dit is nogal vervelend.

Remedie b:

Leg de adressen van het Mask Location Register buiten het interruptvectorgebied. Dit betekent dat de PIC moet worden geselecteerd met

- het adres van de IRQ-vector voor de vectoradresvertaling en
- de adressen van 32 locaties (bepaald door A0 t/m A4) elders in de memory map (het adresgebied van het geheugen) voor het laden van het Mask Location Register.

De adresdecoder moet derhalve een OR-functie zijn van twee decoders: de ene voor de interruptvectoren, de andere voor de maskers in het Mask Location Register (zie fig. 11.14).

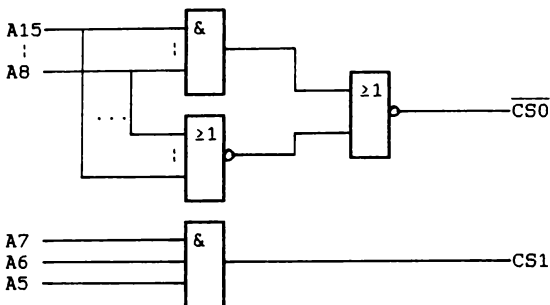


Fig. 11.14 Voorbeeld van een adresdecoder voor de PIC, waarbij de interruptvectoren in een read/write memory kunnen staan

In het voorbeeld van fig. 11.14 wordt het interruptmasker geladen via de adressen \$00E0 t/m \$00FF. Dit zijn 32 adressen, waarvan er eigenlijk maar de helft nodig zijn. Door A1 t/m A4 alle mogelijke waarden te geven, worden alleen de even adressen gebruikt (als A0=0) of de oneven (als A0=1). Voor het interruptmasker is de waarde van A0 niet van belang.

In het geheugengebied van de interruptmaskers mag zich geen ander device of geheugen bevinden (zie ook paragraaf 10.2). Dit betekent dat het eventueel aanwezige geheugen voor de betreffende 32 adressen niet moet worden geselecteerd.

Het masker kan met de adresdecoder van fig. 11.12 met behulp van de volgende instructies worden geladen in de PIC:

STA	\$00E0	masker := 0, dus enable alle interrupts
STA	\$00E1	masker := 0, dus enable alle interrupts
STA	\$00E2	masker := 1, dus disable IN0
STA	\$00E8	masker := 4, dus disable IN0 t/m IN3
STA	\$00EE	masker := 7, dus disable IN0 t/m IN6
STA	\$00F0	masker := 8, dus disable alle interrupts
STA	\$00FF	masker := 15, dus disable alle interrupts

De inhoud van accumulator A is hierbij niet van belang.

Interrupt Request Register

Het Interrupt Request Register wordt niet beïnvloed door het

interruptmasker. Ook de geblokkeerde interrupts worden ingelezen, zodat deze worden afgehandeld, nadat het interruptmasker gereset is. De $\overline{INT}$ -uitgang wordt dan geactiveerd.

Het Interrupt Request Register wordt op de neergaande flank van E geladen met de informatie op de interruptingangen $\overline{IN0}$ t/m $\overline{IN7}$, mits de PIC niet in de selected mode staat ($A4 \cdot A3 \cdot \overline{A2} \cdot \overline{A1} \cdot CS1 \cdot CS0 \cdot R/\overline{W} = 0$). Dit laatste voorkomt, dat er een interruptaanvraag met een hogere prioriteit binnen komt, terwijl de PIC bezig is met het vertalen van het vectoradres. Er zou dan een ongedefinieerde toestand kunnen ontstaan.

Indien in het Interrupt Request Register een interrupt wordt geladen die niet geblokkeerd is, wordt $\overline{INT}$ actief binnen 240 ns (t_{INT}) na de neergaande flank van E (zie ook fig. 11.15). De microprocessor detecteert een interrupt tijdens de neergaande flank van Q, mits de interrupt tenminste 200 ns voor die flank aanwezig is (zie paragraaf 11.3.10). Aan deze voorwaarde is voldaan. De afstand tussen de neergaande flank van E en de neergaande flank van Q bedraagt minimaal $200 + 430 = 630$ ns. Dit is groter dan propagatietijd + setuptijd van de interrupt ($240 + 200 = 440$ ns).

Wordt $\overline{INT}$ actief, maar is deze interrupt geblokkeerd, dan blijft $\overline{INT}$ gelijk aan 1 (inactief), althans voor wat betreft $\overline{INT}$. $\overline{INT}$ wordt wel geladen in het Interrupt Request Register. Als een interruptmasker wordt geladen in het Mask Location Register, dan wordt $\overline{INT}$ binnen maximaal 360 ns na de neergaande flank van E gelijk aan 0 (dus actief). Dit is op tijd met betrekking tot de volgende neergaande flank van Q, immers $(200 + 360) \text{ ns} < (200 + 430) \text{ ns}$.

Interruptprioriteiten

Er kunnen meerdere interruptaanvragen tegelijkertijd aanwezig zijn. Ze worden alle ingeklokt in het Interrupt Request Register. $\overline{IN7}$ heeft altijd de hoogste prioriteit en $\overline{IN0}$ de laagste. In de vector look-up table wordt de vector, behorende bij de (niet-geblokkeerde) interruptaanvraag met de hoogste prioriteit, opgezocht en op de inwendige vectorbus gezet. Tevens wordt de interruptaanvraag doorgegeven aan de microprocessor door de uitgang $\overline{INT}$ van de PIC actief (= 0) te maken.

Zodra de microprocessor het adres van de interruptvector op de adresbus zet, wordt dit door de PIC vertaald in het vectoradres, behorende bij de interruptaanvraag met de hoogste prioriteit. Op deze wijze wordt het startadres opgehaald van de interrupt service routine, die hoort bij de interruptaanvraag van de hoogste prioriteit.

Adresvertaling door de PIC

Indien een adres door de PIC wordt vertaald, treedt er een

extra vertraging van het adres op. Deze bedraagt maximaal 125 ns (in plaats van 75 ns). Hierdoor zou de data wel een te laat door het geheugen op de databus gezet kunnen worden. Om dit te voorkomen, genereert de PIC een signaal STRETCH. Dit signaal is alleen actief (= 0) in de selected mode, dus als er een adresvertaling plaatsvindt. STRETCH=0 geeft derhalve aan dat de PIC zich in de selected mode bevindt. De vertraging tussen de ingangen A1 t/m A4, $\overline{CS0}$, CS1 en R/W en de uitgang STRETCH bedraagt maximaal 140 ns bij het activeren van STRETCH en maximaal 125 ns bij het deactiveren van STRETCH. STRETCH wordt tussen het ophalen van de high-byte en de low-byte van de interruptvector gedurende korte tijd 1 (zie fig. 11.15). Toch kan STRETCH niet rechtstreeks worden verbonden met de MRDY-ingang van de microprocessor, teneinde de klokperiode uit te rekken en zodoende het geheugen voldoende tijd te geven om de data op de bus te zetten. STRETCH is immers afgeleid van de klok. Wordt de klokpuls direct door STRETCH zelf verbreed, dan hangt de klokpuls zichzelf op. De klokpuls E blijft dan gelijk aan 1, zolang MRDY (en dus STRETCH) 0 blijft. STRETCH blijft 0 zolang E 1 blijft. Hiermee hangt het systeem op. Het probleem is eenvoudig op te lossen door STRETCH via een one-shot te verbinden met MRDY. Een one-shot geeft een eenmalige puls af als hij wordt geactiveerd. De one-shot moet in dit geval worden geactiveerd door de neergaande flank van STRETCH. De breedte van het uitgangssignaal van de one-shot hangt niet af van de breedte van de ingangspuls, maar wordt alleen bepaald door de waarden van de extern aan te sluiten weerstand en condensator.

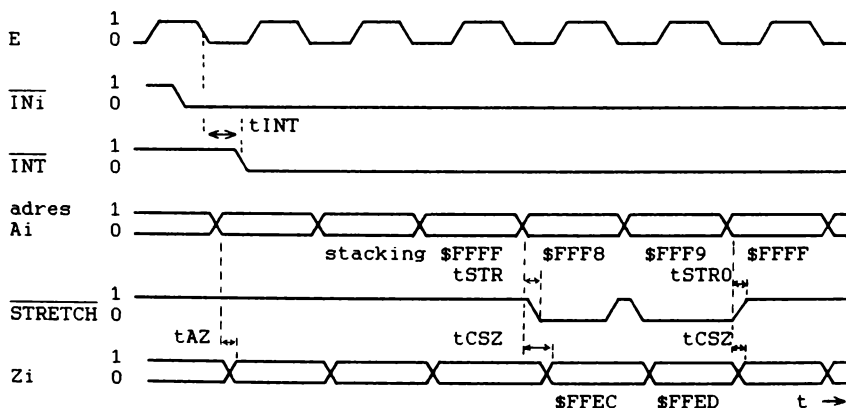


Fig. 11.15 Timing van de PIC (bij vertaling van IN2-vectoradres)

Specificatie van de timing van de PIC (zie fig. 11.15):

$t_{AZ} < 75 \text{ ns}$ (Ai to Zi delay)

$t_{CSZ} < 125 \text{ ns}$ (select to Zi delay;

select = $A4 \cdot A3 \cdot \overline{A2} \cdot \overline{A1} \cdot CS1 \cdot \overline{CS0} \cdot R/\overline{W}$)

```

tSTR < 140 ns (select to STRETCH delay)
tSTRO < 125 ns (deselect to STRETCH delay)
tINT < 240 ns (E to INT delay in non-masked mode)
      < 360 ns (E to INT in masked mode; deze delay treedt
      bij het laden van een lager masker)

```

Alleen de uitgang INT van de PIC is van het open-collectortype. Een open-collectoruitgang kan alleen actief zijn voor het 0-niveau. Hij maakt de uitgang in dat geval ongeveer 0 V. Is de uitgang 1, dan is hij hoogimpedant (zwevend) en dus in feite afgeschakeld (inactief). In dat geval moet de lijn hoog worden gemaakt met behulp van een externe pull-up-weerstand (die de uitgang met een hoog niveau verbindt, bijvoorbeeld met de voedingsspanning). Er mogen meerdere open-collectoruitgangen met dezelfde lijn verbonden zijn. Er treedt dan geen probleem op als sommige uitgangen 0 zijn (ongeveer 0 V), terwijl andere tegelijkertijd 1 zijn (zwevend). De lijn is dan 0 en er gaan geen hoge stromen lopen, omdat de actieve uitgangen per definitie hetzelfde niveau (= 0) hebben. Het 0-niveau overheerst dus. Zo ontstaat een wired-and-functie (zie ook paragraaf 9.4).

Natuurlijk kunt u op elke interruptingang van de PIC meerdere aanvragers aansluiten. In dat geval moet er in de diverse interrupt service routines een polling worden uitgevoerd. Dit is nodig om te achterhalen welke aanvrager op de betreffende interruptingang de interrupt heeft gegenereerd.

Meerdere PIC's aansluiten

Op de (adres)bus kan slechts één PIC worden aangesloten zonder extra maatregelen te treffen. Dit komt doordat de uitgangen Z1 t/m Z4 altijd geactiveerd zijn. Dit betekent dat er meerdere zenders op dezelfde lijn actief zijn bij meer dan één PIC. Bovendien levert elke PIC dezelfde interruptvectoren.

Wilt u toch meerdere PIC's aansluiten, dan moet er extra hardware worden toegevoegd:

- De PIC-uitgangen STRETCH moeten via externe open-collectorpoorten met elkaar worden doorverbonden.
 - De uitgangen Z1 t/m Z4 moeten via tri-state buffers met de adresbus worden verbonden en er moet telkens één en slechts één buffer geactiveerd zijn. De buslijnen A1 t/m A4 moeten immers worden voorzien van informatie.
 - Een deel van de adreslijnen A5 t/m A15 (bij 4 PIC's bijvoorbeeld A5 en A6) moet worden vertaald in unieke adressen per PIC. Deze vertaling moet alleen geschieden, indien de bijbehorende PIC in de selected mode staat, dus als $A4 \cdot A3 \cdot \overline{A2} \cdot \overline{A1} \cdot CS1 \cdot CS0 \cdot R/\overline{W} = 1$.
 - De ingangen E, R/ $\overline{W}$ en A1 t/m A4 van de diverse PIC's worden alle verbonden met de overeenkomstige uitgangen van de processor.
- * Bij het gebruik van PIC's voor de verschillende hardware

interrupts zijn de volgende maatregelen te treffen:

- De $\overline{\text{INT}}$ -uitgangen van de PIC's moeten worden aangesloten op de bijbehorende interruptingangen van de processor.
 - De externe vectoradresvertalers moeten worden geactiveerd door de signalen Interrupt Acknowledge IRQACK, FIACK en NMIACK voor wat betreft de interrupts IRQ, FIRQ en NMI (zie ook fig. 11.6 van paragraaf 11.3.6).
 - Met dezelfde signalen kunnen ook de externe buffers worden geactiveerd. Er moet altijd één adreszender per buslijn geactiveerd zijn. Dit kan worden bereikt door de adresbuffers van interrupt controller te activeren als alle andere niet actief zijn. Zo kunnen de buffers van de IRQ-PIC worden geactiveerd door het signaal NMIACK·FIACK.
 - De ingangen $\overline{\text{CS0}}$ en CS1 van elke PIC moeten worden verbonden met de uitgangen van de decoder.
- * Bij het gebruik van meerdere PIC's voor dezelfde soort hardware interrupt, bijvoorbeeld voor IRQ, ontstaan er meer interruptniveaus, elk met een eigen prioriteit. Bij 4 PIC's krijgen we 32 niveaus. Er dienen nu de volgende maatregelen te worden genomen:
- De (open-collector)uitgangen $\overline{\text{INT}}$ van de PIC's moeten alle worden aangesloten op de $\overline{\text{IRQ}}$ -ingang van de processor.
 - Er moet een externe prioriteitschakeling worden toegevoegd voor het activeren van een bepaalde PIC. Hiervoor kan bijvoorbeeld een daisy chain worden toegepast (zie ook paragraaf 9.6).
 - De ingangen $\overline{\text{CS0}}$ en CS1 van de PIC's moeten worden verbonden met de uitgangen van de externe prioriteitschakeling.
 - De externe vectoradresvertalers moeten eveneens worden geactiveerd door die signalen.
 - Met dezelfde signalen kunnen ook de externe buffers worden geactiveerd. Er moet altijd één adreszender per buslijn geactiveerd zijn. Dit kan worden bereikt door de adresbuffers van de interrupt controller met de laagste prioriteit te activeren, als alle andere niet actief zijn.

Wordt een PIC gebruikt in een systeem met meerdere busmasters (bijvoorbeeld DMA-controllers, dynamic memory refreshschakelingen of meerdere processoren), dan moeten de uitgangen Z1 t/m Z4 altijd via externe tri-state buffers worden verbonden met de adresbus. Deze buffers worden geactiveerd door de busarbitration hardware (zie ook paragraaf 11.3.12).

Vectoradresvertaling voor MC6809-interrupts

De MC6828 is ontwikkeld voor microprocessoren als de MC6800 en de MC6802. Hierbij is \$FFF8 het laagste vectoradres. In paragraaf 9.7.9 hebt u gezien, dat de MC6809 wel lagere vectoradressen kent, en wel voor de interrupts FIRQ, SWI2 en SWI3. Deze adressen vallen samen met de vectoradressen van IN7, IN6 en IN5. Dit betekent dat de PIC zonder meer gebruikt kan worden in systemen, waarin deze twee groepen interrupts niet beide gebruikt worden.

Wilt u deze interrupts ook gebruiken, dan moet er een externe adresvertaling worden toegevoegd, teneinde de overlapping van de vectoradressen te vermijden. Dit kan worden gerealiseerd door bijvoorbeeld A5 altijd 0 te maken bij een INi-interrupt. In dat geval liggen de high-bytes van de vectoradressen van de IRQ-interrupts tussen \$FFC8 en \$FFD6. Het adres van de default routine ligt nu niet meer op de oorspronkelijke waarde \$FFFF8/9, maar op \$FFD8/9. De schakeling om dit te realiseren wordt gegeven in fig. 11.16

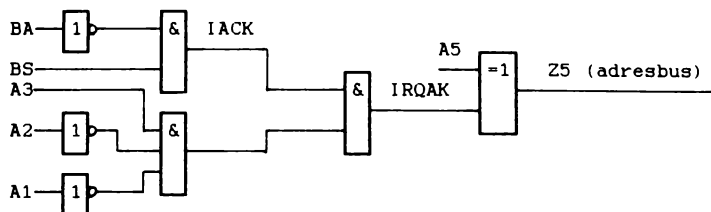


Fig. 11.16 Schakeling voor het verleggen van de interruptvectoren

De exclusive-OR in fig. 11.16 geeft de waarde van A5 onveranderd door naar zijn uitgang, indien IRQAK=0, terwijl A5 wordt geïnverteerd als IRQAK=1 (dus bij het ophalen van de interruptvector).

Wilt u de PIC niet gebruiken voor IRQ-interrupts, maar bijvoorbeeld voor FIRQ-interrupts, dan treedt er nog een tweede probleem op. De PIC voert alleen een adresvertaling uit voor A4A3A2A1 = 1100, dus indien het vectoradres eindigt op \$8 of \$9. Dit is het vectoradres voor de IRQ-interrupt. Voor de FIRQ-interruptvector is A4A3A2A1 = 1011 (zie paragraaf 9.7.9). De PIC kan bij dit adres worden geactiveerd door de inversen van A3, A2 en A1 aan te sluiten op de corresponderende PIC-ingangen. De PIC-uitgangen Z3, Z2 en Z1 moeten dan ook worden geïnverteerd. Gebeurt dit niet, dan kloppen de geheugenadressen niet meer bij een gewone geheugenaccess, dus als er geen startadres van een interrupt service routine wordt opgehaald door de processor. Houdt hierbij rekening met de extra vertraging van deze drie adreslijnen! Bedenk bovendien dat de vertaalde interruptvectoradressen door het inversen van Z1, Z2 en Z3 ook gewijzigd worden. De FIRQ-defaultroutine heeft nu als vectoradres \$FFF6/7. Voor IN7 t/m IN4 liggen de vectoradressen tussen \$FFF8/9 en \$FFFE/F en voor IN3 t/m IN0 tussen \$FFE0/1 t/m \$FFE6/7 (zie ook fig. 11.12). De overlapping met de andere interruptvectoradressen, die ook hier optreedt, kan worden voorkomen met een soortgelijke schakeling als gegeven in fig. 11.16.

11.5 Read Only Memories

Read only memories zijn geheugens, waarin alleen gelezen kan worden. De informatie is in deze geheugens van tevoren aangebracht. Bij de zogenaamde mask-programmed ROM's is dit tijdens de fabricage gebeurd. De afgelopen jaren zijn er diverse andere vormen van ROM's ontwikkeld:

- PROM's (Programmable ROM's)

Dit zijn ROM's, die door de gebruiker zelf eenmalig zijn te programmeren. Het programmeren vindt plaats door (met behulp van korte stroompulsen, die door een speciale PROM-programmer worden geleverd) verbindingen (fuses) in de PROM op te blazen. Op deze wijze worden er nullen geprogrammeerd. Laat men de fuses intact, dan blijven op de betreffende locaties enen staan.

- EPROM's (Erasable Programmable ROM's)

Deze ROM's worden ook door de gebruiker zelf geprogrammeerd met behulp van een zogenaamde EPROM-programmer. Hierbij worden geen fuses opgeblazen, maar wordt er in de gates van de (MOS-)geheugentransistoren wel of geen lading opgeslagen. Hierdoor kunnen de transistoren wel of niet geleiden, waardoor de 1- en 0-niveaus ontstaan. De lading wordt met hogere spanningen aangebracht. Doordat deze lading niet kan wegvloeien, blijft de informatie in het geheugen-IC behouden, ook al valt de voedingsspanning weg. De lading kan alleen worden weggehaald met behulp van ultraviolet licht (of röntgenstraling). Hiertoe zijn deze geheugen-IC's voorzien van een glazen venster. Het wissen van een EPROM duurt ongeveer 20 minuten.

- EAROM's of EEPROM's (Electrically Alterable ROM's of Electrically Erasable PROM's)

Deze ROM's kunnen zowel elektrisch worden beschreven als gewist. Hierbij zijn hogere spanningen nodig. Het schrijven en wissen gaat factoren langzamer (ongeveer 100 tot 1000 keer) dan het lezen. Vandaar dat deze elementen niet geschikt zijn als read/write geheugen. Ze worden toegepast als zogenaamde read mostly memories (RMM's).

Een 512x4 geheugen is een geheugen dat 512 woorden bevat van elk 4 bits. Het geheugen heeft 4 uitgangen (ten behoeve van de 4 bits van het geselecteerde woord). Het aantal adresingangen bedraagt 8 (ten behoeve van de selectie van de 512 woorden). Het totale aantal bits bedraagt 2048. Men spreekt hierbij van een woordgeorganiseerd geheugen, omdat er telkens meerdere bits tegelijk worden uitgelezen.

Bij een bitgeorganiseerd geheugen, selecteert elk adres slechts één bit. Zo'n geheugen heeft slechts één uitgang).

Een 32k8 geheugen is een geheugen dat bestaat uit 32768 woorden van elk 8 bits.

Geheugens zijn niet microprocessor-gebonden. Ze kunnen worden

toegepast bij verschillende soorten processoren van verschillende fabrikanten.

In deze paragraaf wordt de AM27256 32k8 EPROM van Advanced Micro Devices behandeld.

Het programmeren van deze EPROM (met behulp van een EPROM-programmer) duurt ongeveer 5 minuten. Op het programmeren gaan we niet verder in. Dit kan met de huidige software ondersteunde programmers op een eenvoudige wijze geschieden.

De 32k8 EPROM heeft 8 tri-state data-uitgangen O0 t/m O8. Deze worden met de databus van de microprocessor verbonden.

Voor het selecteren van het gewenste geheugenwoord, zijn er 15 adresingangen A0 t/m A14. Deze zijn met de adresbus verbonden.

Om de EPROM te kunnen blokkeren als er een ander device (bijvoorbeeld read/write memory of een PIA) wordt geselecteerd, is er een Chip Enable ingang $\overline{CE}$. Deze wordt via een adresdecoder met de overige adreslijnen verbonden. In dit geval is er slechts één adreslijn (A15) over. De adresdecoder bestaat nu uit een doorverbinding of uit een inverter. Hiermee wordt bepaald in welk gebied van de memory map de EPROM ligt. Voor $\overline{CE}=A15$ ligt de EPROM in het gebied van \$0000 t/m \$7FFF. Als $\overline{CE}=A15$ ligt hij tussen \$8000 en \$FFFF. Als $\overline{CE}=1$ zijn alle uitgangen hoogimpedant, ongeacht de toestand van A_i en $\overline{OE}$. In dit geval dissipeert de EPROM ongeveer een kwart van het vermogen dan wanneer $\overline{CE}=0$.

De output enable ingang $\overline{OE}$ kan worden gebruikt om de datazenders van de EPROM af te schakelen (hoogimpedant te maken). De ingang $\overline{OE}$ kan worden verbonden met de inverse van R/ $\overline{W}$. Op deze wijze wordt bereikt, dat de EPROM alleen wordt geactiveerd als R/ $\overline{W}=1$, dus bij het lezen. Als $\overline{OE}=1$ zijn alle uitgangen hoogimpedant, ongeacht de waarden van A_i en $\overline{CE}$.

Behalve Vcc (= 5 V) en GND (= 0 V) heeft een EPROM nog een voedingsspanningsaansluiting Vpp. Deze is nodig om de EPROM te programmeren en moet hiertoe 12,5 V zijn. In de zogenaamde read mode moet Vpp gelijk zijn aan 5 V.

Raadpleeg voor de timing van de AM2756 fig. 11.17. Overigens bestaan er ook snellere versies (de AM27256-1 en -2) en langzamere (de AM27256-3 en -4).

De specificaties van de timing van de AM27256 is als volgt (met als referentiepunten 0,8 V en 2,0 V):

- tACC < 250 ns (address to output delay)
- tCE < 250 ns ($\overline{CE}$ to output delay)
- tOE < 100 ns ($\overline{OE}$ to output delay)
- tOH > 0 ns (output hold time from first of A_i, $\overline{CE}$ or $\overline{OE}$)
- tDF < 60 ns (first of $\overline{OE}$ or $\overline{CE}$ to output float)

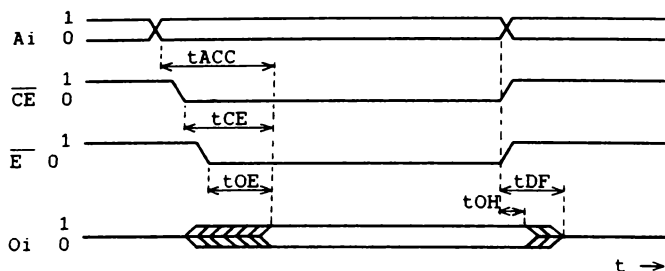


Fig. 11.17 Timing van EPROM (AM27256)

Fig. 11.18 toont de verbindingen tussen de microprocessor en de EPROM. Hierbij wordt de EPROM geselecteerd met adressen tussen \$8000 en \$FFFF.

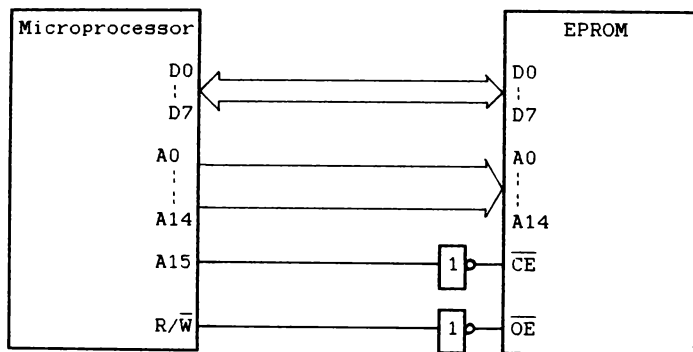


Fig. 11.18 Verbindingen tussen processor en EPROM

Voorbeeld 11.1 (Timingberekeningen bij het lezen in een EPROM)

Laten we eens nagaan of aan de timing-eisen wordt voldaan, indien de EPROM wordt aangesloten aan de microprocessor, zoals in fig. 11.18 is aangegeven.

Er kan alleen worden gelezen in de EPROM. De microprocessor leest op de neergaande flank van de klok E de data in. De data moet daartoe tenminste 80 ns voor die flank tot minstens 10 ns na die flank aanwezig zijn op de ingangen van de microprocessor (zie fig. 11.3).

Veronderstel dat voor de propagatietijd van de signalen over de bussen het volgende geldt:

databus: 5 ns tot 50 ns

adresbus: 4 ns tot 40 ns

controlbus: 3 ns tot 35 ns

Verder is gegeven dat de propagatietijd van de inverter tussen 1 en 6 ns ligt.

Eerst gaan we na of aan de setuptijd (80 ns) is voldaan. Hoe later de data beschikbaar is, des te ongunstiger is het. Het signaal $\overline{OE}$ is het minst kritisch. $\overline{CE}$ zal het laatst aankomen, omdat er nog een extra vertraging zit tussen adres en $\overline{CE}$ (namelijk de adresdecoder, in dit geval een inverter).

De data komt op zijn laatst bij de microprocessor aan op (zie ook fig. 11.3):

t = 0 ns (verzend adres op neergaande flank van E)
+ 25 ns (maximale flanktijd van E)
+ 250 ns (maximale tijd van E neergaand tot Q opgaand)
- 50 ns (minimale tijd tussen adres geldig en Q opgaand)
+ 40 ns (maximale looptijd over adresbus)
+ 6 ns (maximale propagatietijd van adresdecoder)
+ 250 ns (maximale propagatietijd van EPROM)
+ 50 ns (maximale looptijd over databus)
= 571 ns na de neergaande flank van E.

Dit betekent dat de data tenminste 1000 ns - 571 ns = 429 ns voor de volgende neergaande flank van E aanwezig is op de ingangen van de microprocessor. Dit is meer dan de vereiste 80 ns, dus aan de setuptijd is ruimschoots voldaan.

Om na te gaan of onder de ongunstigste omstandigheden ook aan de holdtijd (10 ns) wordt voldaan, moeten we uitrekenen wanneer de data op zijn vroegst ongeldig wordt op de ingangen van de microprocessor. Dit is op:

t = 0 ns (verzend adres op neergaande flank van E)
+ 0 ns (minimale flanktijd van E)
+ 20 ns (minimale holdtijd van adres en $R/\overline{W}$)
+ 3 ns (minimale looptijd van $R/\overline{W}$ over controlbus)
+ 1 ns (minimale propagatietijd van adresdecoder)
+ 0 ns (minimale holdtijd van $\overline{OE}$ van EPROM:
EPROM wordt gedeselecteerd)
+ 5 ns (minimale looptijd over databus)
= 29 ns na de neergaande flank van E.

Dit betekent dat de data tenminste 27 ns na de neergaande flank van E nog aanwezig is op de ingangen van de microprocessor. Dit is meer dan de vereiste 10 ns, dus aan de holdtijd is eveneens voldaan.

11.6 Read/Write Memories

Read/write memories worden doorgaans als RAM (Random Access Memories) aangeduid, omdat de eerste halfgeleidergeheugens willekeurig toegankelijk waren. Dit in tegenstelling tot een magneetband, die eerst vooruit of achteruit gespoeld moet worden om een willekeurig adres te bereiken.

De eerste halfgeleidergeheugens waren read/write geheugens. Het

nadeel van deze geheugens is dat alle informatie verdwenen is, nadat de voedingsspanning is weggefallen. De latere ROM's behouden hun informatie ook zonder voedingsspanning. Ze kunnen echter niet worden beschreven.

Zo ontstond de benaming RAM versus ROM. De ROM is overigens ook van het random access type.

Er is onderzoek gaan naar read/write halfgeleidergeheugens die hun informatie behouden, nadat de voedingsspanning is verdwenen. Dit zijn de zogenaamde NOVRAM's (Non-Volatile RAM's). Tot op heden gaat het schrijven hierin factoren langzamer dan het lezen, zodat er eigenlijk sprake is van RMM's (Read Mostly Memories). Hiertoe kunnen de EAROM's of de EEPROM's worden gerekend.

De RAM's worden onderverdeeld in

- SRAM's (Static RAM's)

Hier bestaan de geheugenelementen uit onderling teruggekoppelde transistors. De informatie blijft behouden zolang de voedingsspanning aanwezig is.

- DRAM's (Dynamic RAM's)

De geheugenelementen zijn kleine capaciteiten. Ze verliezen hun lading ten gevolge van lek en moeten derhalve regelmatig (elke 2 ms) opnieuw worden beschreven met hun informatie. Ze moeten worden opgefrist (refreshed). Ze nemen minder ruimte in beslag en dissiperen minder dan de DRAM's. Vandaar dat er meer dynamische geheugencellen dan statische op een chip kunnen worden ondergebracht. Ze zijn trager dan statische cellen (ongeveer een factor twee tot drie).

We beschouwen in deze paragraaf de IMS1420 van INMOS. Dit is een 4k4 SRAM. Hij heeft 4 bidirectionele datalijnen I/O1 t/m I/O4 en 12 adresingangen A0 t/m A11.

De datalijnen I/O1 t/m I/O4 worden verbonden met de databus van de microprocessor. Deze heeft echter 8 datalijnen. Om toch een byte-georiënteerd geheugen te krijgen, worden de er twee SRAM's met elk vier uitgangen in cascade geschakeld. Hierbij worden de signalen I/O1 t/m I/O4 van de ene SRAM verbonden met de lijnen D0 t/m D3 van de databus en de signalen I/O1 t/m I/O4 van de andere SRAM worden aangesloten op D4 t/m D7 (zie fig. 11.19).

De adresingangen van de SRAM zijn natuurlijk verbonden met de overeenkomstige adreslijnen van de adresbus.

De chip Enable ingang $\bar{E}$ dient voor de selectie van de chip.

Deze wordt derhalve via een adresdecoder met de overige adreslijnen verbonden. Bij een 16k4 SRAM zijn dit de lijnen A12 t/m A15.

Als $\bar{E}=0$ is de chip geselecteerd en kan er data worden gelezen of geschreven.

Als $\bar{E}=1$, is de chip niet geselecteerd. In deze zogenaamde standby mode is de dissipatie ongeveer een derde van die in de geselecteerde toestand. De uitgangen I/O1 t/m I/O4 zijn dan hoogimpedant.

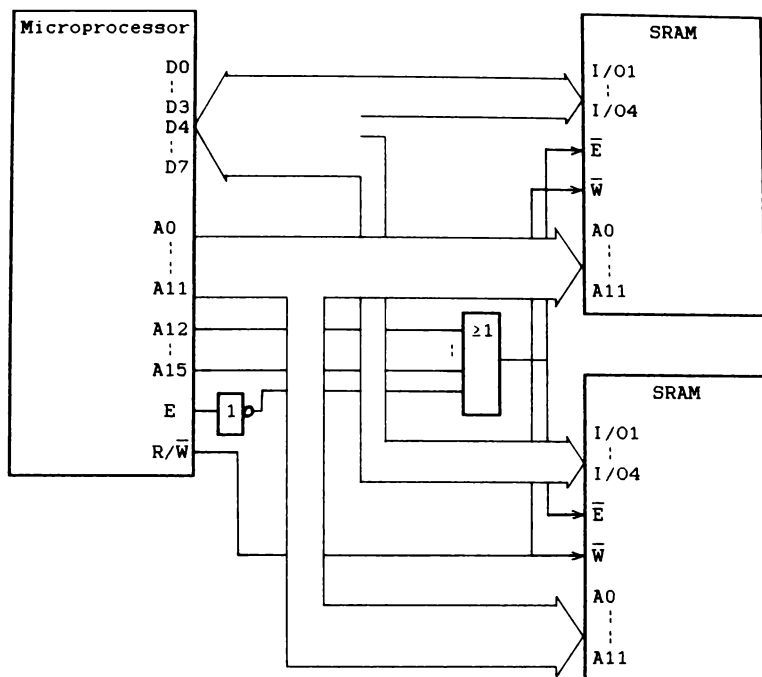


Fig. 11.19 Verbindingen tussen processor en SRAM

Om te bepalen of er wordt gelezen, dan wel geschreven in de SRAM, is er een ingang Write enable $\bar{W}$. Deze ingang wordt aangesloten op de $R/\bar{W}$ -lijn van de controlbus.

Voor $\bar{W}=0$ wordt er geschreven en fungeren I/O1 t/m I/O4 als inputs.

Voor $\bar{W}=1$ wordt er gelezen en werken I/O1 t/m I/O4 als outputs.

Schrijven in de SRAM

Er dient te worden gezorgd dat het adres niet wordt gewijzigd tijdens het schrijven. Verandert het adres, dan bestaat de kans dat de informatie op meerdere adressen in het geheugen wordt opgeborgen. De data mag wel veranderen, mits deze voor het inactief worden van $\bar{W}$ of $\bar{E}$, lang genoeg geldig is geweest. De laatst geldende data op I/O1 t/m I/O4 staat dan in het geheugen.

Om te schrijven moet de SRAM zijn geselecteerd ($\bar{E}=0$) en de ingang $\bar{W}$ moet gelijk zijn aan 0. De ingang die het laatst actief wordt, bepaalt het begin van de schrijfoperatie. De ingang die het eerst inactief wordt, bepaalt het einde van de schrijfoperatie.

Zolang $\overline{E}=\overline{W}=0$ moet het juiste adres aanwezig zijn op de adresingangen. Aangezien $R/\overline{W}$ ongeveer tegelijkertijd verandert met A_i (zie fig. 11.4), kan het signaal $R/\overline{W}$ niet worden gebruikt om de schrijfoperatie te besturen. Als $R/\overline{W}=0$ mag het schrijven niet beginnen, omdat het adres nog kan veranderen. Schrijven zou dan tot gevolg hebben dat op diverse geheugenadressen de inhoud wordt gewijzigd. Dit betekent dat het schrijven moet worden uitgesteld, totdat A_i stabiel is. Er is dus een ander signaal nodig dat de schrijfoperatie bestuurt, en wel via de SRAM-ingang $\overline{E}$. Hiertoe mag $\overline{E}$ pas 0 worden nadat het adres stabiel is. $\overline{E}$ moet vervolgens weer 1 worden voordat het adres verandert. Fig. 11.20 laat de timing van de bijbehorende schrijfoperatie zien.

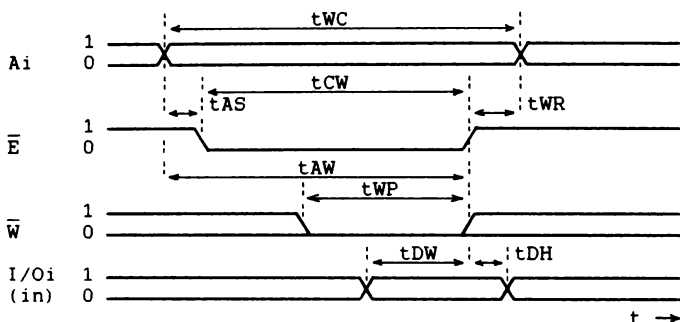


Fig. 11.20 Timing van het schrijven in een SRAM

Bij het schrijven moet voldaan zijn aan de volgende eisen (gemeten op het 50%-niveau; zie ook fig. 11.20):

- $t_{WC} > 50$ ns (Write Cycle time)
- $t_{CW} > 45$ ns (Chip Enable to end of Write)
- $t_{AS} > -5$ ns (Address Setup to beginning of Write)
- $t_{WR} > 5$ ns (Address Hold after end of Write)
- $t_{AW} > 40$ ns (Address Setup to end of Write)
- $t_{WP} > 45$ ns (Write Pulse width)
- $t_{DW} > 20$ ns (Data Setup to end of Write)
- $t_{DH} > 5$ ns (Data Hold after end of Write)

De I/O_i -zenders van de SRAM worden hoogimpedant op hoogstens 25 ns na het 0 worden van $\overline{W}$.

Fig. 11.4 laat zien dat de microprocessor data op de databus zet na de opgaande flank van Q . Deze data is in ieder geval stabiel tijdens het laatste deel van de 1-puls van Q en tijdens de 1-puls van E . Het adres is reeds 50 ns voor de opgaande flank van Q geldig. E en Q kunnen beide worden gebruikt voor het besturen van de schrijfoperatie.

Bij het lezen moet de SRAM geselecteerd zijn bij de neergaande flank van E , omdat de microprocessor op deze flank de data inleest. Derhalve is E het aangewezen signaal voor het

vrijgeven van de adresdecoder (zie fig 11.19). In fig. 11.19 geldt dat $\bar{E}=0$ als $A12=A13=A14=A15=0$ en $E=1$.

Voorbeeld 11.2 (Timingberekeningen bij het schrijven in de IMS1420 SRAM)

Neem voor de propagatietijd van de signalen over de bussen de volgende waarden:

databus: 5 ns tot 50 ns
adresbus: 4 ns tot 40 ns
controlbus: 3 ns tot 35 ns

Verder geldt dat de propagatietijd van de decoder (inverter + OR) tussen 2 en 12 ns ligt.

Daar de pulsbreedte van E minimaal 450 ns bedraagt, wordt ruim voldaan aan de eisen die gesteld zijn aan tWC, tCW, tAW en tWP.

tAS > -5 ns betekent dat het adres maximaal 5 ns na de neergaande flank van $\bar{E}$ geldig moet zijn. Om na te gaan of hieraan wordt voldaan, rekenen we uit wanneer het adres op zijn laatst geldig is. Dit is op (zie ook fig. 11.4)

t = 0 ns (neergaande flank van E)
+ 250 ns (maximale afstand van E neergaand tot Q opgaand)
- 50 ns (minimale tijd tussen Ai geldig en Q opgaand)
+ 25 ns (maximale flanktijd van Q)
+ 40 ns (maximale looptijd over adresbus)
= 265 ns na de neergaande flank van E.

$\bar{E}$ op de SRAM wordt op zijn vroegst actief op:

t = 0 ns (neergaande flank van E)
+ 430 ns (minimale breedte van de 0-puls van E)
+ 3 ns (minimale looptijd over controlbus)
+ 2 ns (minimale propagatietijd van decoder)
= 435 ns na de neergaande flank van E.

Hierbij is verondersteld dat de verschillende adressignalen binnen de microprocessor verschillende propagatietijden kunnen hebben. In de praktijk zijn deze verschillen verwaarloosbaar klein.

U ziet dat de SRAM geactiveerd wordt op tenminste $434 - 265 = 196$ ns na het geldig zijn van het adres. Dit is meer dan de vereiste - 5 ns.

Di moet op de ingangen van de SRAM aanwezig zijn op tenminste 20 ns voor de opgaande flank van $\bar{E}$.

De data is zo laat mogelijk geldig op (zie ook fig. 11.4):

t = 0 ns (neergaande flank van E)
+ 250 ns (maximale afstand tussen E neergaand en Q opgaand)
+ 25 ns (maximale flanktijd van Q)
+ 200 ns (maximale vertragingstijd van data)
+ 50 ns (maximale looptijd over de databus)
= 525 ns na de neergaande flank van E.

De volgende opgaande flank van $\bar{E}$, waarop de schrijfoperatie stopt, komt op zijn vroegst na:

- t = 0 ns (neergaande flank van E)
- + 1000 ns (minimale cyclustijd)
- + 3 ns (minimale looptijd over de controlbus)
- + 2 ns (minimale propagatietijd van decoder)
- = 1005 ns

De tijd tussen het 1 worden van $\bar{E}$ en het geldig zijn van de data op de SRAM bedraagt tenminste $1004 - 525 = 479$ ns. Dit is aanzienlijk meer dan de vereiste 20 ns.

Adres en data moeten tenminste 5 ns na het 0 worden van E stabiel blijven.

Het adres verandert op zijn vroegst op:

- t = 0 ns (neergaande flank van E)
- + 20 ns (minimale adres holdtijd)
- + 4 ns (minimale looptijd over adresbus)
- = 24 ns na de neergaande flank van E.

De data verandert op zijn vroegst op:

- t = 0 ns (neergaande flank van E)
- + 30 ns (minimale write data holdtijd)
- + 5 ns (minimale looptijd over databus)
- = 35 ns na de neergaande flank van E.

De opgaande flank van $\bar{E}$ verschijnt op zijn laatst op:

- t = 0 ns (neergaande flank van E)
- + 35 ns (maximale looptijd over de controlbus)
- + 12 ns (maximale propagatietijd van decoder)
- = 47 ns na de neergaande flank van E.

Dit is voor wat betreft het adres $47 - 24 + 5 = 28$ ns te laat en voor de data $47 - 35 + 5 = 17$ ns.

Nu is het niet erg realistisch om voor de propagatietijd van de ene bus de minimale waarde te nemen en tegelijkertijd voor een andere bus de maximale waarde. De verschillende ingangen van device en decoder bevinden zich immers op gelijk afstand van de microprocessor, zodat de looptijden vrijwel gelijk zijn. Ten gevolge van de niet-oneindig steile flanken van de signalen en de verschillende drempelspanningen van de diverse ingangen moet er toch met "looptijdverschillen" worden gerekend. De drempelspanning is de spanning waarbij een ingang van logisch niveau verandert. Spanningen beneden de drempelspanning worden als 0 gezien. Spanningen boven de drempelspanning worden als 1 gezien. Bij TTL (Transistor-Transistor-Logic) ligt de drempelspanning tussen 0,8 V en 2,0 V. Bij een opgaande flank schakelt een ingang met een drempelspanning van 0,8 V vroeger dan een ingang die een drempelspanning van 2,0 V heeft (zie fig. 11.21). Zo ontstaan er als het ware looptijdverschillen ($t_2 - t_1$), ook al komt een bepaald signaal op hetzelfde moment

op de verschillende ingangen aan. Deze verschillen zijn te verwaarlozen, indien de ingangen zich op dezelfde chip bevinden. De drempelspanningen van schakelingen op dezelfde chip zijn nagenoeg identiek.

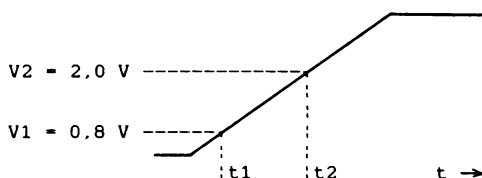


Fig. 11.21 "Looptijdverschillen" ten gevolge van verschillen in drempelspanning

Vertraging ten gevolge van capaciteiten

De schijnbare looptijden ten gevolge van de flanktijden ontstaan door het opladen van capaciteiten.

Uit $i = C \cdot dV/dt$ volgt dat $(t_2 - t_1) = (V_2 - V_1) \cdot C / I$,

mits de capaciteit wordt opgeladen of ontladen met een constante stroom I .

C is de som van de capaciteiten van de buslijn (1 pF/cm), de ontvangers (5 pF), de zenders (10 pF) en de bedrading tussen zenders en/of ontvangers en buslijn (1 pF/cm).

Voor een bus van 30 cm met 1 zender, 5 ontvangers en 3 cm bedrading voor elke ontvanger is dit totaal 80 pF.

De zenders van digitale schakelingen zijn doorgaans geen stroombronnen, maar spanningsbronnen. Door te rekenen met de gemiddelde uitgangsstroom in het traject waarin de capaciteit wordt opgeladen of ontladen, kan een goede indruk worden verkregen omtrent de grootte van de vertraging.

Een microprocessor kan geen grote stromen leveren. Gerekend moet worden met een minimale stroom van ongeveer 5 mA. Dit betekent dat bij een belasting van 80 pF een flank ontstaat van $80/5 = 16$ ns/V. Het opladen van de buslijn van 0 V tot 2 V kost dan ongeveer 32 ns. Het ontladen van 3,6 V (het hoog-niveau van een TTL-uitgang) naar 0,8 V kost 45 ns. Als er pullup weerstanden naar +5 V worden toegepast, kost het ontladen zelfs 67 ns. Indien pullup weerstanden nodig zijn (voor het verkrijgen van een gedefinieerd niveau bij zwevende zenders, bijvoorbeeld op interruptlijnen) kunnen ze beter met een lagere spanning, bijvoorbeeld 3 V verbonden worden. Dit laatste kan ook worden gerealiseerd met een weerstandsdeling vanuit +5 V. Dan is er geen aparte 3 V voedingsspanning nodig.

U ziet dat de vertragingstijden vrijwel volledig worden veroorzaakt door de flanktijden van de signalen, die op hun beurt weer ontstaan ten gevolge van de capacitieve belasting.

Leidt de meest ongunstige (worst case) berekening niet tot het

gewenste resultaat, dan zijn er een aantal mogelijkheden die een oplossing kunnen bieden.

Rekening houden met maximale verschillen binnen hetzelfde IC
Afwijkingen tussen de zenders van hetzelfde type binnen eenzelfde IC zijn zeer klein. Ook de diverse ontvangers binnen één IC zijn vrijwel identiek. Dit geldt ook voor de IC's die uit eenzelfde plak silicium zijn vervaardigd. Deze zijn immers tegelijk in één procesgang gemaakt en derhalve vrijwel identiek.

Tussen zenders van een verschillende soort kunnen de eigenschappen verschillend zijn. Wel is hierbij sprake van een zekere correlatie. Het zal niet voorkomen dat de ene zender een maximale stroom levert en de andere een minimale. De moeilijkheid is echter een afschatting te maken van de maximale verschillen binnen hetzelfde IC.

Nemen we aan dat de verschillende zenders van een microprocessor van dezelfde soort zijn, dan kunnen we er van uit gaan dat de eigenschappen onderling niet meer dan ongeveer 20 % zullen afwijken.

Dit betekent dat bij een maximale vertraging van het signaal E geldt dat de vertragingen van de adres- en datasignalen ten hoogste 4/5 van hun maxima zullen bedragen, dus respectievelijk 32 ns (in plaats van 40 ns) en 40 ns (in plaats van 50 ns). De hold tijd van het adres bedraagt dan minstens $20 + 32 - 47 = 5$ ns, hetgeen voldoende is.

De hold tijd van de data bedraagt nu tenminste $30 + 40 - 47 = 23$ ns. Dit is meer dan de vereiste 5 ns.

Buffer gates toepassen

Indien de buslooptijden te groot zijn, kan men zijn toevlucht zoeken tot het gebruik van buffer gates. Deze worden tussen de processoruitgangen en de bus geplaatst. De buffer gates kunnen grotere uitgangsströmen leveren, bijvoorbeeld tenminste 35 mA. De flanksteilheid wordt dan bij 80 pF gelijk aan 2.3 ns/V in plaats van 16 ns/V. Dit is een factor 7 sneller, zodat de vertraging van de databus van 50 ns daalt naar 7 ns. Natuurlijk gaat een deel van de winst verloren door de propagatietijd van de buffer gates. Is deze maximaal 15 ns, dan is de winst nog 28 ns. Er zijn snelle buffers verkrijgbaar.

Bij grotere capacitieve belastingen (meer aangesloten zenders en/of ontvangers) worden in de praktijk altijd buffers toegepast.

Statische aanpak

Minima en maxima zijn geen absolute waarden. Er is altijd een (kleine) kans dat een waarde buiten de gespecificeerde grenswaarden ligt. De eigenschappen van bouwstenen hangen van een groot aantal eigenschappen af. Dit betekent dat er een normale verdeling ontstaat. De kans van voorkomen van een bepaalde waarde is het grootst bij het gemiddelde en wordt

kleiner naarmate de afwijking tot de gemiddelde waarde groter wordt. De meeste waarden liggen dus rond het gemiddelde. De spreiding is een maat waarmee men kan aangeven in welk gebied de meeste waarden liggen (zie fig. 11.22). De kans dat een waarde verder dan 3 maal de standaarddeviatie (spreiding) van het gemiddelde afwijkt, bedraagt 0,26 %.

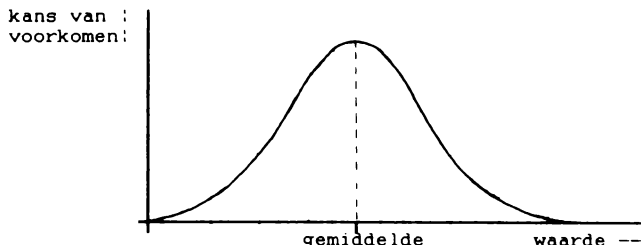


Fig. 11.22 Normale kansverdeling.

Worden een aantal waarden opgeteld, dan is het gemiddelde van de somwaarde gelijk aan de som van de gemiddelden van de opgetelde waarden. De kans dat alle waarden minimaal zijn, is uiterst gering. Voor de spreiding geldt nu dat de kwadraten mogen worden gesommeerd. Voorwaarde is dat de waarden van elkaar onafhankelijk zijn.

De berekening (voorbeeld 11.2) van de hold tijden van data en adres gaat nu als volgt:

	gemiddelde $\pm$ p.spreiding
databus	28 ns $\pm$ 22 ns
adresbus	22 ns $\pm$ 18 ns
controlbus	19 ns $\pm$ 16 ns
decoder	7 ns $\pm$ 5 ns
hold tijd Ai	105 ns $\pm$ 95 ns (10 ns minimaal en 250-50 ns maximaal, zie fig. 11.3)
hold tijd Di	35 ns $\pm$ 5 ns (maximum geschat op 40 ns, zie fig. 11.4)
hold tijd Ai SRAM	3 ns $\pm$ 2 ns

De waarde van p is niet bekend. We handhaven in onze berekeningen dezelfde factor. Dit wil zeggen dat overschrijdingen van de grenswaarden in onze schakeling niet vaker voorkomen dan dat de componenten niet aan de specificaties voldoen.

Er moet gelden (voorbeeld 11.2):

$$t(\text{Ai processor}) + t(\text{Ai bus}) > t(\text{E bus}) + t(\text{decoder}) + t_{\text{WR}}$$

(zie fig. 11.20), ofwel

$$t(\text{Ai processor}) + t(\text{Ai bus}) - t(\text{E bus}) - t(\text{decoder}) - t_{\text{WR}} > 0$$

Voor bovenstaande formule geldt:

gemiddelde = $105 + 22 - 19 - 7 - 3 = 98 \text{ ns}$

$(p\text{-spreiding})^2 = 95^2 + 18^2 + 16^2 + 5^2 + 2^2 = 9.634 \text{ ns}^2$, dus

$(p\text{-spreiding}) = 98 \text{ ns}$

De minimale waarde van de formule is dus $98 - 98 = 0 \text{ ns}$. Dit is juist voldoende.

In feite zijn de vertragingen van E en Ai niet onafhankelijk van elkaar. In dit geval wordt de situatie gunstiger, omdat bij een grote vertraging van E geen zeer kleine delay van Ai kan voorkomen.

Verder moet gelden dat

$t_{DHW} + t(\text{Di bus}) > t(\text{E bus}) + t(\text{decoder}) + t_{DH}$, ofwel

$t(\text{Di processor}) + t(\text{Di bus}) - t(\text{E bus}) - t(\text{decoder}) - t_{DH} > 0$

Voor bovenstaande formule geldt:

gemiddelde = $35 + 28 - 19 - 7 - 3 = 34 \text{ ns}$

$(p\text{-spreiding})^2 = 5^2 + 22^2 + 16^2 + 5^2 + 2^2 = 794 \text{ ns}^2$, dus

$(p\text{-spreiding}) = 28 \text{ ns}$

De minimale waarde van de formule is dus $34 - 28 = 6 \text{ ns} > 0 \text{ ns}$.

Dit is correct.

Verlagen van klokfrequentie

Bij te grote buslooptijden brengt het verlagen van de klokpulsfrequentie uitkomst, indien buffer gates niet voldoende helpen.

Externe hardware

Indien de looptijdverschillen serieuze problemen veroorzaken, moet uiteindelijk de toevlucht worden genomen tot extra hardware. Hierbij valt vooral te denken aan delayschakelingen om sommige signalen extra te vertragen ten opzichte van anderen en aan one-shots om bepaalde pulsbreedten te vergroten of te verkleinen.

Lezen in de SRAM

Bij het lezen in de SRAM geldt het tijddiagram van fig. 11.23. Hierbij moet voldaan zijn aan de volgende eisen (gemeten op het 50%-niveau):

$t_{ACS} > 55 \text{ ns}$ (Chip Enable Access time, mits adres geldig is)

$t_{LZ} > 20 \text{ ns}$ (Chip Enable $\downarrow$ to output active)

$t_{HZ} < 25 \text{ ns}$ (Chip Enable $\uparrow$ to output disable)

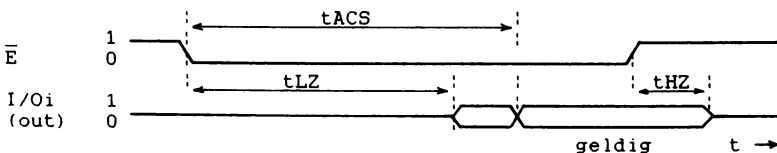


Fig. 11.23 Timing bij het lezen in een SRAM

Voorbeeld 11.3 (Timingberekening bij het lezen in de IMS1420 SRAM)

Bij het lezen in de SRAM moet voldaan zijn aan de setuptijd (80 ns) en aan de holdtijd (10 ns) van de microprocessor (zie ook fig. 11.3).

De data komt op zijn laatst na
t = 0 ns (neergaande flank van E)
+ 25 ns (maximale flanktijd van E)
+ 430 ns (maximale breedte van de 0-puls van E)
+ 25 ns (maximale flanktijd van E)
+ 35 ns (maximale looptijd over controlbus)
+ 12 ns (maximale propagatietijd van decoder)
+ 55 ns (maximale propagatietijd van de SRAM)
+ 50 ns (maximale looptijd over databus)
= 632 ns na de neergaande flank van E.

Dit is dus $1000 - 632 = 368$ ns voor de volgende opgaande flank van E, hetgeen ruimschoots voldoet aan de vereiste setuptijd van 80 ns.

De data valt op zijn vroegst weg op
t = 0 ns (neergaande flank van E)
+ 3 ns (minimale looptijd over controlbus)
+ 2 ns (minimale propagatietijd van decoder)
+ 5 ns (minimale propagatietijd van SRAM)
+ 5 ns (minimale looptijd over databus)
= 15 ns na de neergaande flank van E.

Dit voldoet aan de vereiste holdtijd van 10 ns.

De minimale propagatietijd van de SRAM is bij de berekening gesteld op het tiende deel van MHz.

De minimale propagatietijden zijn meestal niet opgegeven in de data sheets. Een veilige minimumwaarde is het tiende deel van het maximum.

11.7 I/O-timing

11.7.1 Bus timing

Voor de communicatie van een I/O-device met een microprocessor zijn de volgende signalen van belang:

- D0 t/m D7: de aansluitingen voor de datasignalen.
- R/W: Read/Write signaal om aan te geven of er moet worden gelezen of geschreven in het I/O-circuit; het Read/Write signaal geeft de richting van het datatransport aan.
- E: Enable signaal ten behoeve van de synchronisatie tussen microprocessor en I/O-circuit.
- CSi: Chip Select ingangen voor het selecteren van het I/O-

- circuit via de bus, opdat de processor in het circuit kan lezen of schrijven.
- $\overline{\text{RSi}}$: Register Select ingangen om aan te geven met welk van de I/O-registers de processor wil communiceren.
 - $\overline{\text{IRQ}}$: Interrupt request uitgang om een interruptaanvraag te kunnen doen aan de processor.

Fig. 11.24 toont de tijddiagrammen bij het lezen in het I/O-circuit, fig. 11.25 laat de timing zien bij het schrijven. De referentiepunten zijn 0.8 V voor het 0-niveau en 2,0 V voor het 1-niveau.

Net als bij de microprocessor zijn ook snellere versies te verkrijgen van de I/O-bouwstenen. Dit zijn de A-versie en de B-versie van PIA, ACIA, PTM en DMAC. Besproken wordt hier slechts de standaardversie.

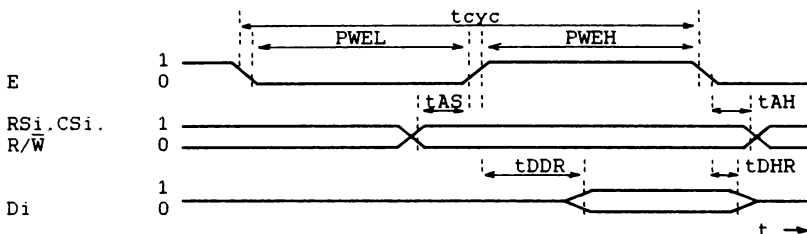


Fig. 11.24 Timing bij het lezen in een I/O-circuit

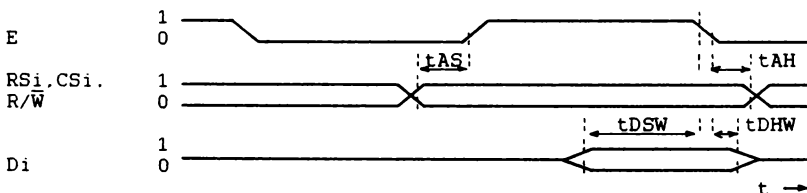


Fig. 11.25 Timing bij het schrijven in een I/O-circuit

Specificaties van MC6821 (PIA), MC6850 (ACIA), MC6840 (PTM) en MC6844 (DMAC):

- $t_{r,f} < 25 \text{ ns}$ (Enable rise, fall time)
- $t_{cyc} > 1000 \text{ ns}$ (cycle time); $t_{cyc} < 10 \mu\text{s}$
- $PWEL > 430 \text{ ns}$ (pulse width Enable low)
- $PWEH > 450 \text{ ns}$ (pulse width Enable high)
- $t_{AS} > 80 \text{ ns}$ ("address" setup time)
- $t_{AH} > 10 \text{ ns}$ ("address" hold time)
- $t_{DDR} < 290 \text{ ns}$ (data delay time, read)
- $t_{DHR} > 20 \text{ ns}$ (data hold time, read)
- $t_{DHR} < 50 \text{ ns}$ (delay to output high impedance)
- $t_{DSW} > 165 \text{ ns}$ (data setup time, write)
- $t_{DHW} > 10 \text{ ns}$ (data hold time, write)

De DMAC heeft geen aparte register select ingangen. Hiertoe dienen de adresingangen A0 t/m A4.

11.7.2 PIA-timing

Behalve voor de buszijde van de PIA gelden ook voor de I/O-zijde van de PIA eisen voor wat betreft de timing van de signalen. Zo wordt de peripheral data ingelezen op het ingangssignaal E (zie fig. 11.26). Hier geldt:

$t_{PDS} > 200 \text{ ns}$ (peripheral data setup time)

$t_{PDH} > 0 \text{ ns}$ (peripheral data hold time)

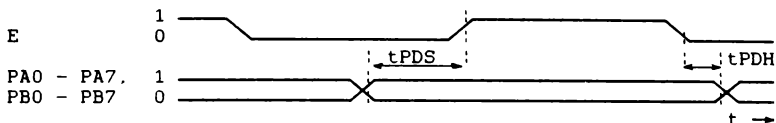


Fig. 11.26 Inlezen van peripheral data

Uit fig. 11.26 blijkt dat de data van een randapparaat synchroon moet worden aangeboden ten opzichte van de systeemklok E. Anders wordt de PIA misschien niet correct ingelezen. In het algemeen werkt de randapparatuur asynchroon ten opzichte van de microprocessor. Er moet derhalve een manier worden gezocht om de data te synchroniseren. Er zijn hiertoe diverse mogelijkheden:

- De data van de randapparatuur kan met behulp van een externe schakeling worden gesynchroniseerd, alvorens deze aan te bieden aan de peripheral inputs van de PIA (zie paragraaf 10.5).
- Het synchroniseren kan met behulp van software gebeuren. Hiertoe moet de laatst gelezen data van de peripheral registers PRA en PRB worden bewaard. Worden PRA en PRB opnieuw gelezen en blijkt de data te zijn veranderd, dan is het niet zeker of de juiste data in deze registers staat, omdat mogelijk niet is voldaan aan setuptijd of holdtijd. Door de gelezen data wederom te bewaren en na een korte tijd opnieuw de peripheral registers te lezen, kan de juiste data worden binnengehaald. De data is correct indien bij twee opeenvolgende leesopdrachten geen verandering wordt geconstateerd. Natuurlijk moet de wachttijd niet kleiner worden gekozen dan dertertijden van contacten en niet kleiner dan looptijdverschillen tussen de datalijnen onderling. Ook moet de wachttijd niet groter zijn dan de opeenvolgende datawisselingen, omdat anders data wordt gemist.
- U kunt de PIA in de pulse mode of in de handshake mode laten werken. Nu geeft het randapparaat aan wanneer de data geldig is. Het betreffende signaal wordt op een interruptingang van de PIA ontvangen, hetgeen tot gevolg heeft dat een interrupt request bit in de PIA wordt gezet (zie de paragrafen 10.8 en

10.9). Door de registers PRA en PRB te lezen nadat de betreffende interrupt request bit is geset, wordt telkens de juiste data binnengehaald in de processor.
Zie ook fig. 11.27 en 11.28.

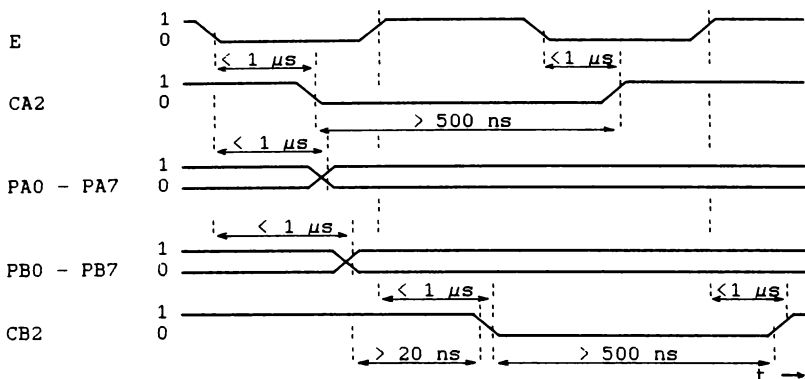


Fig. 11.27 Pulse mode van PIA

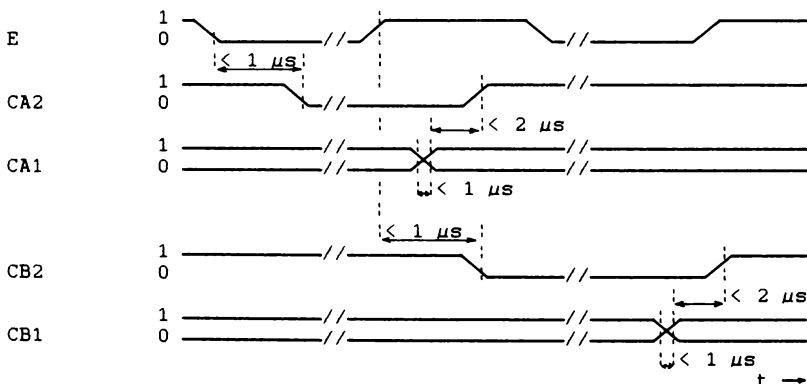


Fig. 11.28 Handshake mode van PIA

In de pulse mode wordt CA2 laag gemaakt na het lezen van PRA. CB2 wordt laag gemaakt na het schrijven in PRB (zie paragraaf 10.8). CA2 wordt hoog na de volgende neergaande flank van E, CB2 na de volgende opgaande flank van E.

In de handshake mode wordt CA2 laag na het lezen in PRA, terwijl CB2 laag wordt na het schrijven in PRB (zie ook paragraaf 10.9). CA2 en CB2 worden hoog gemaakt ten gevolge van een actieve flank op CA1, respectievelijk op CB1.

11.7.3 ACIA-timing

Fig. 11.29 toont de timing bij het zenden van data via de uitgang TxD en fig. 11.30 schetst de situatie bij het ontvangen van data via de ingang RxD (zie ook paragraaf 10.11). Aan de specificaties van setuptijd en holdtijd hoeft alleen te worden voldaan bij synchrone ontvangst van seriële data. Dit is het geval in de mode, waarbij de klok wordt gedeeld door een factor 1. Bij asynchrone ontvangst treedt de synchronisatie intern in de ACIA op met behulp van de klok RxCL met een frequentie die een factor 16 of 64 hoger is dan de bitfrequentie van het ontvangen signaal RxD.

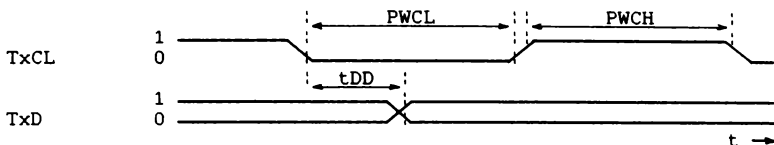


Fig. 11.29 Timingdiagram bij het zenden via de ACIA

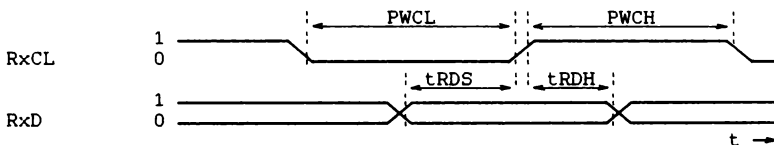


Fig. 11.30 Timingdiagram bij het ontvangen via de ACIA

Er geldt:

- $tr, f < 1 \mu s$ of 10% van pulsbreedte (kleinste van beide)
(rise, fall time van andere inputs dan E)
- $PWCL > 600 \text{ ns}$ in f/16, f/64 mode (pulse width clock low)
- $PWCH > 600 \text{ ns}$ in f/16, f/64 mode (pulse width clock high)
- $PWCL > 900 \text{ ns}$ in f/1 mode (pulse width clock low)
- $PWCH > 900 \text{ ns}$ in f/1 mode (pulse width clock high)
- $fC < 800 \text{ kHz}$ in f/16, f/64 mode (clock frequency)
- $fC < 500 \text{ kHz}$ in f/1 mode (clock frequency)
- $tTDD < 600 \text{ ns}$ (clock to data delay)
- $tRDS > 250 \text{ ns}$ in f/1 mode (receive data setup time)
- $tRDH > 250 \text{ ns}$ in f/1 mode (receive data hold time)

De maximale vertraging tussen de neergaande flank van E en het signaal op de uitgang $\overline{RTS}$ is maximaal 560 ns.

11.7.4 PTM-timing

Voor de signalen van de PTM geldt (zie ook fig. 11.31):

- $tr, f < 1 \mu s$ én $< t_{cyc}, E$
(rise, fall time van andere inputs dan E)

```

PWL > tcyce + tsu + thd asynchrone mode (input pulse width low
      Ci, Gi, RESET)
PWH > tcyce + tsu + thd asynchrone mode (input pulse width high
      Ci, Gi)
PWL > 120 ns asynchrone mode (input pulse width low C3/8)
PWH > 120 ns asynchrone mode (input pulse width high C3/8)
tsu > 200 ns synchrone mode (input setup time)
thd > 50 ns synchrone mode (input hold time)
tco < 700 ns (output delay)

```

In de asynchrone mode loopt de klok $\overline{Ci}$ asynchroon ten opzichte van E, in de synchrone mode lopen de klokken synchroon.
In de asynchrone mode vindt intern in de PTM synchronisatie plaats tussen E en de externe klokingangen en gate-ingangen.

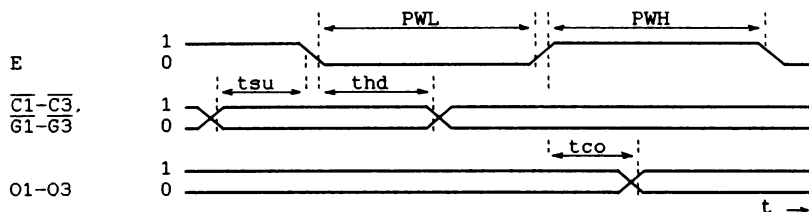


Fig. 11.31 Timingdiagram van de PTM

11.7.5 DMAC-timing

De MC6844 DMAC kent drie verschillende DMA-modes: mode 1, mode 2 en mode 3. In mode 1 wordt de uitgang $\overline{DRQ1}$ van de DMAC geactiveerd, terwijl in mode 2 en 3 de uitgang $\overline{DRQ2}$ wordt geactiveerd (zie ook paragraaf 10.14).

Mode 1: three state control steal mode

Mode 1, ook wel de three state control steal mode genoemd, is bedoeld om in systemen met MC6800-processoren of met MC6809E-processoren een DMA-cyclus aan te vragen aan de externe klokgenerator. Hierbij wordt de klok voor de microprocessor stilgezet, terwijl de systeemklok voor DMAC, geheugen en I/O doorloopt. Tevens wordt ervoor gezorgd dat de microprocessor van de bus afschakelt (door de ingang TSC van de microprocessor hoog te maken, zie paragraaf 11.3, of door de externe buffers van de processor af te schakelen). Hiervoor moet de klokgenerator zorgen. Het voordeel van deze methode is de snelle reactie. De microprocessor hoeft niet eerst de instructie af te maken alvorens de DMA-aanvraag goedgekeurd wordt. De processorklok wordt stilgezet, zodat er intern in de processor niets verandert. Zodoende kan de processor het afmaken van de onderbroken instructie gewoon voortzetten, zodra de klok na het einde van de DMA-cyclus wordt gestart.

Denk hierbij aan Doornroosje, die in een diepe slaap viel, toen

zij zich aan het spinnewiel van de boze fee prikte. Met haar viel iedereen in het paleis in slaap. De kok hield de lepel stil in de soep en de slager wilde net de slagersjongen een oorvijs geven. De klok hield op met lopen. In het paleis bleef alles precies zoals het was. Alleen buiten het paleis ging het leven gewoon door. Toen honderd jaar later een prins zich een weg baande door de doornhaag en de prinses wakker kustte, begon de klok weer te lopen. Iedereen ontwaakte en ging door met zijn bezigheden, alsof er niets was gebeurd. Dit was inderdaad het geval binnen het paleis, omdat de tijd was blijven stilstaan. De kok ging door met roeren in zijn soep en de slager gaf de slagersjongen een oorvijs, omdat deze een stuk vlees had laten vallen.

Bij het stilzetten van de microprocessorklok gebeurt eigenlijk hetzelfde. Doordat er geen klokpuls meer is, treedt er binnen de processor geen enkele verandering meer op. Er wordt geen informatie meer binnengehaald en er wordt geen informatie verplaatst of veranderd in de processor. Elke actie binnen de processor wordt veroorzaakt door een klokflank. Ontbreken de klokflanken, dan gebeurt er niets meer. Zodra de klok weer wordt gestart, wordt het proces binnen de processor voortgezet op de plaats waar het was gestopt. Er kan dus midden in een instructie worden gestopt. Het stoppen gebeurt in de praktijk aan het einde van een klokcyclus. Dit is gedaan om bij een lees- of schrijfpdracht te zorgen dat de data van geheugen of I/O wordt binnengehaald of in geheugen of I/O wordt opgeborgen, alvorens de processor te stoppen. Dit is nodig om te voorkomen dat deze data verloren gaat. Buiten de processor loopt de klok gewoon door en worden de bussen gebruikt voor datatransfer tussen geheugen en I/O, zodat de vorige data op de bus verloren gaat. Een geheugenaccess van de microprocessor moet dus worden voltooid, voordat de processorklok wordt stilgezet. Dit hoeft niet aan het einde van een instructie gebeuren. Bij een compare-instructie bijvoorbeeld vindt de geheugenaccess niet aan het einde van de instructie plaats. Na het binnenhalen en decoderen van de instructie wordt de data uit het geheugen gehaald. In de volgende cyclus wordt deze vergeleken met de inhoud van een register. Het binnenhalen van een opcode vindt zelfs helemaal aan het begin van een instructie plaats. Daarna moet eventueel de operand worden binnengehaald, waarna de instructie wordt uitgevoerd. Hiermee zijn een aantal klokcycli gemoeid. De processorklok kan aan het einde van elke klokcyclus worden gestopt. Hierbij moet rekening worden gehouden met de dynamische eigenschappen van de processor. De processor mag slechts een beperkt aantal klokcycli worden stilgezet, omdat anders de informatie in de processorregisters verloren gaat. Een klokcyclus mag niet langer dan 10 μ s duren. Dit betekent dat er gedurende maximaal 10 μ s DMA mag plaatsvinden. Nu geldt in mode 1 dat er slechts één byte wordt overgedragen door middel van DMA. Hierna wordt de besturing altijd teruggegeven aan de microprocessor. Wel zijn er rond de DMA-cyclus twee dode

cycli, zodat de processorklok gedurende drie cycli stilstaat (zie ook fig. 11.32).

De DMAC-uitgang $\overline{\text{DRQ1}}$ kan bij de MC6809-microprocessor worden toegepast door deze rechtstreeks aan te sluiten op de $\overline{\text{DMA/BREQ}}$ -ingang van de MC6809. Hierbij blijven de externe klokken E en Q doorlopen en worden intern in de processor de klokken stilgezet. Er is dus geen externe klokgenerator nodig. Bovendien worden de uitgangen Di , Ai en $\text{R}/\overline{\text{W}}$ door de processor hoogimpedant gemaakt.

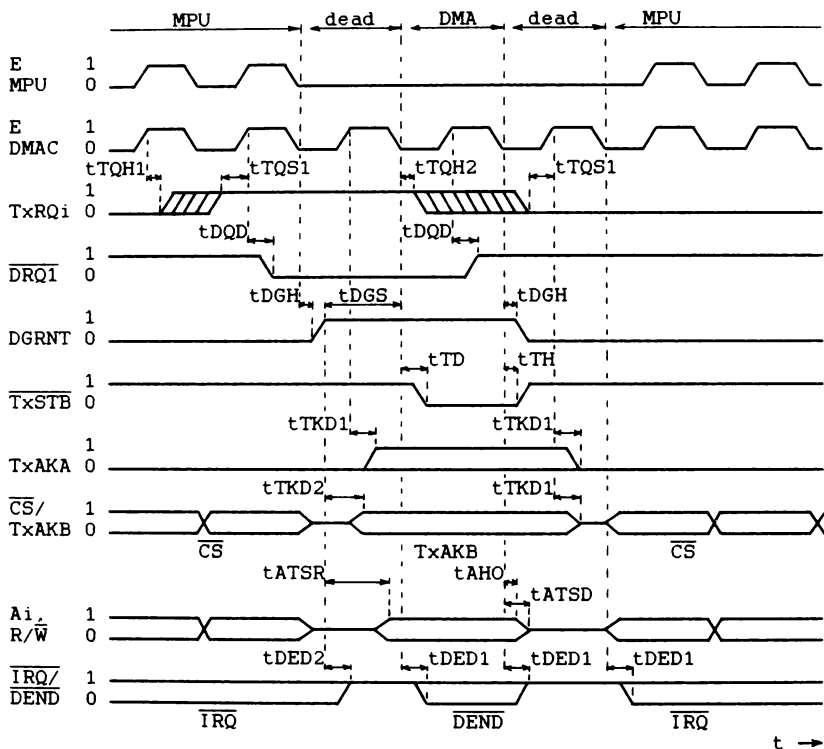


Fig. 11.32 Timing van three state control steal mode

Voor de figuren 29, 32 en 33 gelden de volgende specificaties:

- $t_{\text{TQS1}} > 120 \text{ ns}$ (TxRQ setup time at rising edge of E)
- $t_{\text{TQS2}} > 210 \text{ ns}$ (TxRQ setup time at falling edge of E)
- $t_{\text{TQH1}} > 20 \text{ ns}$ (TxRQ hold time at rising edge of E)
- $t_{\text{TQH2}} > 20 \text{ ns}$ (TxRQ hold time at falling edge of E)
- $t_{\text{DGS}} > 155 \text{ ns}$ (DGRNT setup time)
- $t_{\text{DGH}} > 10 \text{ ns}$ (DGRNT hold time)
- $t_{\text{AD}} < 270 \text{ ns}$ (address, R/ $\overline{\text{W}}$ output delay)

```

tAHO > 30 ns (address, R/ $\bar{W}$  output hold time)
tATSD < 720 ns (address, R/ $\bar{W}$  three state delay)
tATSR < 430 ns (address, R/ $\bar{W}$  three state recovery time)
tDQD < 375 ns ( $\overline{DRQ1}$ ,  $\overline{DRQ2}$  delay)
tTKD1 < 400 ns (TxAKA, TxAKB delay, E rising)
tTKD2 < 190 ns (TxAKA, TxAKB delay, DGRNT rising)
tDED1 < 300 ns ( $\overline{IRQ/DEND}$  delay, E falling)
tDED2 < 190 ns ( $\overline{IRQ/DEND}$  delay, DGRNT rising)
tTD < 270 ns ( $\overline{TxSTB}$  output delay)
tTH > 30 ns ( $\overline{TxSTB}$  output hold time)

```

Laten we de timing bekijken bij een DMA-aanvraag door een floppy disk controller (zie ook paragraaf 10.14). De FDC doet een transfer request als het Data Output Register gereed is om data te ontvangen (om deze vervolgens op de disk te schrijven) of als het Data Input Register is gevuld met data (die van de disk is gelezen). De uitgang TxRQ van de FDC wordt dan 1.

Wordt deze 1 later dan tTQH1 na de opgaande flank van E op een van de ingangen TxRQi van de DMAC ontvangen, dan zal de volgende klokcyclus (weer) een MPU-cyclus zijn, omdat de DMAC geen DMA-request zal doen. Er zal dan een klokcyclus later een DMA-aanvraag door de DMAC worden gedaan. Is TxRQi gelijk aan 1 op minstens tTQS1 voor de opgaande flank van E, dan activeert de DMAC op tDQD (< 375 ns) na deze flank zijn DMA-request uitgang $\overline{DRQ1}$, respectievelijk $\overline{DRQ2}$.

Deze DMA-aanvraag wordt naar de microprocessor of naar de externe klokgenerator gestuurd.

Wordt de ingang DMA/BREQ van de MC6809 actief (dus 0) op tenminste 200 ns voor de neergaande flank van E, dan schakelt de processor zijn uitgangen van de bus af. Bereikt de DMA-aanvraag de processor later, dan kan het gebeuren dat de processor een klokcyclus later van de bus afschakelt of de processor komt in een ongedefinieerde toestand. $\overline{DMA/BREQ}$ moet extern worden gesynchroniseerd om dit laatste te voorkomen.

De klokcyclus waarin de processor van de bus afschakelt, zal een dode cyclus zijn. De microprocessor beantwoordt het ontvangen van een DMA-aanvraag met een DMA grant door (na de neergaande flank van E) de uitgangen BA en BS beide 1 te maken (zie ook fig. 11.3 en fig. 11.33). Het signaal DMA grant zal de DMAC later dan tDGH (10 ns) na die neergaande flank van E bereiken, maar eerder dan tDGS (155 ns) voor de volgende neergaande flank van E. Zoals uit fig. 11.32 blijkt, activeert de DMAC het signaal transfer strobe pas aan het einde van de cyclus, waarin het DMA-grant signaal werd ontvangen, zodat gedurende deze cyclus de bus inderdaad niet wordt gebruikt. Na het ontvangen van DMA-grant bepaalt de DMAC alvast de waarden van TxAKA en TxAKB. Deze signalen mogen echter pas worden gebruikt als het signaal transfer strobe is geactiveerd. Samen met transfer strobe ($\overline{TxSTB}$) activeert de DMAC ook de adreslijnen en het R/ $\bar{W}$ -signaal. Uit de transfer strobe en de

signalen TxAKA en TxAKB wordt het signaal TxAKi voor device i afgeleid (bijvoorbeeld de FDC).

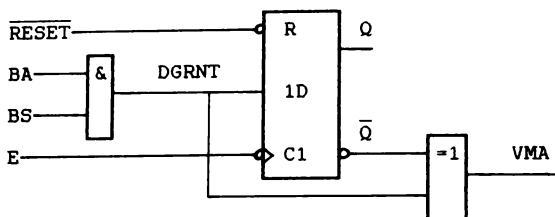


Fig. 11.33 Schakeling voor DGRNT en VMA

Uit fig. 11.32 blijkt dat tijdens de dode cycli de adreslijnen en de read/write-lijn zwevend zijn. Dit betekent dat de informatie op deze lijnen ongedefinieerd is, zodat er ten onrechte in geheugen (of I/O) zou kunnen worden geschreven (en wel aan het begin van de E-puls, zie ook paragraaf 11.6). Derhalve moet er een signaal VMA (Valid Memory Address) worden gemaakt, dat 0 is tijdens de dode cycli. VMA moet in de adresdecoders van geheugen en I/O worden opgenomen om te voorkomen dat deze tijdens dode cycli ten onrechte worden geadresseerd. Fig. 11.33 laat een schakeling zien, die ervoor zorgt dat VMA=0 tijdens dode cycli. In fig. 11.32 ziet u dat de dode cycli de klokcycli zijn waarin DGRNT van waarde verandert. Een flank kan worden gedetecteerd door de oude en de nieuwe waarde van het signaal te vergelijken. Zijn deze ongelijk, dan is er een flank geweest. Om de oude en de nieuwe waarde van DGRNT te vergelijken, moet dit signaal worden opgeslagen in een flipflop. Dit gebeurt aan het begin van een cyclus. Is de uitgang van de flipflop (de oude waarde) ongelijk aan DGRNT (de nieuwe waarde, dan is er een flank geweest en moet VMA=0 worden. De vergelijking kan eenvoudig geschieden met een exclusive-OR. Deze geeft een 1 af als de ingangen ongelijke waarden hebben. Inverteren van de uitgang van de exclusive-OR levert het gewenste signaal VMA. Hetzelfde wordt bereikt door een van de ingangen van de exclusive-OR te inverteren, hetgeen wordt gerealiseerd door de inverse uitgang van de flipflop te nemen.

Wordt de TxAK-ingang van de FDC tenminste 160 ns voor de opgaande flank van E gelijk aan 1, dan neemt de FDC, bij het lezen in het geheugen, op de volgende neergaande flank van E de data op de bus over. Deze data is dan door het geheugen op de bus gezet, net als bij een gewone leesoperatie van de microprocessor. Bij het schrijven in het geheugen zet de FDC (hoogstens 480 ns) na het ontvangen van TxAK de data op de bus. Deze wordt dan op de volgende neergaande flank van E door het geheugen overgenomen.

Hiermee is er een byte overgedragen tussen geheugen en I/O en

is de DMA-cyclus beëindigd.

Timing bij schrijven in SRAM tijdens DMA

Looptijden over databus: 5 ns tot 50 ns

adresbus: 4 ns tot 40 ns

controlbus: 3 ns tot 35 ns

van DMA-signalen: 2 ns tot 20 ns

t = 0 ns (neergaande flank van E bij SRAM, begin DMA-cyclus)

+ 35 ns (maximale looptijd van E over controlbus)

+ 270 ns (maximale delay van transfer strobe)

+ 20 ns (maximale looptijd van transfer strobe van DMAC naar TxAK van FDC)

+ 480 ns (maximale tijd tussen TxAK en data op FDC)

+ 50 ns (maximale looptijd van data van FDC naar SRAM)

+ 20 ns (maximale data setuptijd tDW van SRAM)

875 ns

Dit is kleiner dan de minimale afstand tussen twee opeenvolgende neergaande flanken van E (= 1000 ns). Er is dus aan de timingeis (setuptijd van de SRAM) voldaan.

Na het ontvangen van TxAK maakt de FDC (binnen 450 ns) TxRQ inactief (= 0). Ook al zou TxRQ gelijk aan 1 blijven, de DMAC maakt reeds tijdens de DMA-cyclus zijn uitgang DMA request inactief (binnen 375 ns na de opgaande flank van E) en na de DMA-cyclus wordt ook transfer strobe inactief. Om de uitgang DMA request opnieuw te activeren, moet TxRQ gelijk aan 1 zijn tijdens de volgende opgaande flank van E (die optreedt tijdens de dode cyclus, zie fig. 11.32). DMA request wordt dan geactiveerd na die opgaande flank van E in de dode cyclus. De processor zal hierop het signaal DMA grant activeren na de volgende neergaande flank van E. Zo ontstaat er weer een dode cyclus. Dit betekent dat er (in mode 1) maximaal één byte aaneengesloten wordt overgedragen op basis van DMA. Hierna volgen twee dode cycli, alvorens er een volgende byte kan worden overgedragen op basis van DMA.

Indien het device niet tijdig nieuwe data beschikbaar heeft of kan verwerken, volgt er na de eerste dode cyclus natuurlijk een MPU-cyclus.

In fig. 11.31 ziet u dat het signaal $\overline{CS}$ in tijd is gemultiplexed met het signaal TxAKB. Dit kan heel goed, omdat het eerste signaal alleen wordt gebruikt tijdens MPU-cycli. Alleen dan immers kan de microprocessor de DMAC adresseren. Gedurende MPU-mode is het signaal TxAKB niet nodig, omdat dit signaal wordt gebruikt om aan te geven welk device een DMA transfer mag doen. Dit vindt dus alleen plaats tijdens DMA-cycli. De adresdecoder moet een open-collectoruitgang hebben om te voorkomen dat er hoge stromen gaan lopen als TxAKB uitgang is. Tijdens DMA-cycli zal de DMAC nooit geselecteerd zijn, omdat de DMAC zelf geen data-uitwisseling op basis van DMA doet.

Overigens wordt TxAKB alleen door de DMAC geactiveerd in de 4-channel mode (zie ook paragraaf 10.14). In de 2-channel mode kunnen alleen de devices 0 en 1 een transfer request doen. Dan geeft het signaal TxAKA aan of device 0 (TxAKA=0) of device 1 (TxAKA=1) een transfer mag doen. TxAKB is hierbij overbodig en blijft derhalve ook tijdens een DMA-cyclus als ingang gedefinieerd.

Ook de DMAC-signalen $\overline{\text{IRQ}}$ en $\overline{\text{DEND}}$ zijn time-multiplexed. Tijdens MPU mode geeft deze uitgang aan of er een interrupt request uitgaat van de DMAC. Er kan immers een DMA van een device zijn beëindigd (alle bytes zijn overgedragen). Tijdens de dode cycli is de uitgang altijd hoog (inactief). Gedurende de DMA-cyclus geeft deze uitgang aan of de laatste byte met het betreffende device is wordt uitgewisseld (DMA-END). Dit signaal kan nu worden gebruikt om aan het device te melden dat alle bytes zijn overgedragen en dat de DMA block transfer voor dat device is voltooid. Het signaal DMA-END wordt altijd geactiveerd gedurende de overdracht van de laatste byte, ook al is de interrupt van het betreffende device gedisabled.

De interruptaanvraag en het signaal het signaal DMA-END moeten worden gescheiden om valse interrupts voorkomen en om te voorkomen dat voorkomen dat een device een interruptaanvraag als een DMA-END signaal interpreteert. Dit kan eenvoudig worden gerealiseerd door de interruptaanvraag alleen door te geven aan de processor als DMA-grant inactief is, dus buiten een DMA-cyclus. Verder wordt DMA-END alleen tijdens DMA-cycli doorgelaten naar de devices als het signaal transfer strobe actief is (zie fig. 11.34).

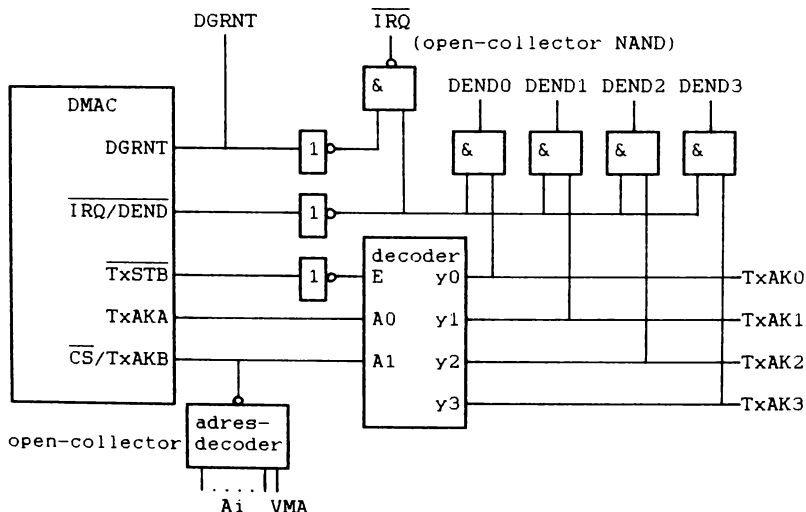


Fig. 11.34 Decodeerschakeling voor DEND_i , TxAK_i en $\overline{\text{IRQ}}$

De schakeling van fig. 11.34 kan worden toegepast in de 4-channel mode. Worden slechts devices aangesloten op de kanalen 0 en 1 van de DMAC, dan kan de schakeling worden vereenvoudigd. In de 2-channel mode wordt TxAKB niet gebruikt. De decoder voor de signalen TxAK0 en TxAK1 hoeft dan slechts één adresingang te hebben. Wordt slecht één DMA-device aangesloten, dan kan de decoder zelfs geheel vervallen. Het signaal $\overline{\text{TxSTB}}$ kan dan na invertieren dienst doen als TxAK.

Mode 2: halt steal mode

Het enige verschil tussen de halt steal mode en de three state control steal mode is dat de DMA-controller in de eerste mode (mode 1) uitgang $\overline{\text{DRQ1}}$ activeert en in de tweede mode (mode 2) de uitgang $\overline{\text{DRQ2}}$. Deze laatste uitgang wordt ook geactiveerd in mode 3 (halt burst mode). Bij de MC6809 deze kunnen DMA request uitgangen aangesloten worden op de DMA/bus request ingang $\overline{\text{DMA/BREQ}}$ of op de halt ingang $\overline{\text{HALT}}$. Het verschil is dat bij aansluiten op de DMA/bus request ingang de DMA-aanvraag aan het einde van de klokcyclus wordt gehonoreerd, terwijl bij het aansluiten op de halt ingang eerst de instructie wordt afgemaakt (zie fig. 11.35).

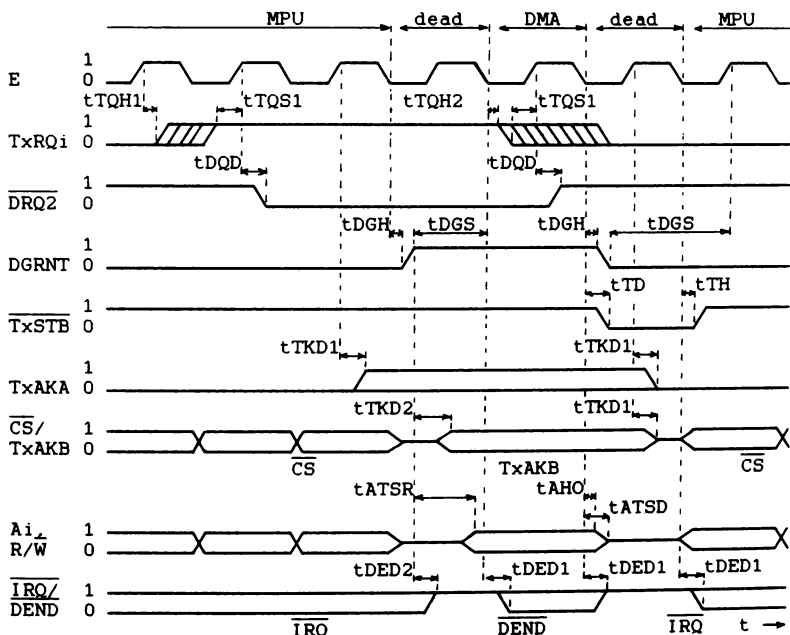


Fig. 11.35 Timing van halt steal mode

Mode 3: halt burst mode

Het gebruik van de uitgang DRQ2 heeft als voordeel dat deze in de burst mode kan worden gebruikt. Hierbij blijft deze uitgang actief totdat de laatste byte overgedragen is. Alle over te dragen bytes worden nu achter elkaar getransporteerd tussen geheugen en I/O (burst = vlaag, salvo). De burst mode mag bij de MC6809-processor worden toegepast op zowel de DMA/bus request ingang als op de halt ingang. Bij gebruik van de DMA/bus request ingang worden intern in de processor de klokken stilgezet. De processor neemt zelf na 16 cycli de besturing over de bus terug voor self-refresh (zie fig. 11.36). Op deze wijze wordt voorkomen dat de informatie in de processor tijdens burst mode DMA verloren gaat.

Hierbij treedt er een timingprobleem op voor de DMAC. Deze bemonstert het signaal DGRNT op de neergaande flank van E. Als de microprocessor de bus terugvraagt voor self-refresh, wordt DGRNT pas 0 na de neergaande flank van E. Dit betekent dat de DMAC in de dode cyclus begonnen is met een DMA-cyclus. Het Byte Count Register wordt aan het begin van de DMA-cyclus met 1 verlaagd. Als DGRNT 0 wordt, zal de DMAC alsnog van de bus afschakelen. Het Byte Count Register is inmiddels echter met 1 verlaagd en het Address Register zal aan het einde van de cyclus eveneens met 1 worden veranderd. Er kan echter geen byte worden overgedragen, omdat tijdens deze DMA/dode cyclus niet het juiste adres op de bus staat. De DMAC schakelt immers af, zodra DGRNT 0 wordt. Omdat het Byte Count Register en het Address Register wel zijn veranderd, loopt de burst mode DMA foutief af. Om dit te voorkomen moet ervoor worden gezorgd dat DGRNT eerder (en wel tDGS) voor de neergaande flank van E gelijk aan 0 wordt. Hiervoor is een externe telschakeling nodig.

Merk op dat het refreshen alleen correct functioneert als de frequentie van de E-klok 1 MHz bedraagt. Bij lagere frequenties overschrijden 16 klokcycli de maximaal toegestane periodetijd van 16 μ s. Ook bij gebruik van MRDY gaat de informatie verloren, omdat hierbij de klokcycli worden opgerekt.

Bij gebruik van de halt-ingang treden deze problemen niet op, omdat de interne klokken blijven doorlopen.

Wordt een externe klokgenerator gebruikt, dan mag er geen burst mode worden toegepast, omdat de klok te lang wordt stilgezet. Burst mode is in ieder geval veilig bij gebruik van de halt ingang van de processor. Vandaar dat mode 3 en mode 2 halt burst, respectievelijk halt steal worden genoemd. Hierbij wordt dezelfde DMA request uitgang DRQ2 van de DMAC gebruikt.

In de burst mode blijft de DMA request uitgang van de processor actief totdat de laatste byte is overgedragen, ook al zijn de TxRQi-ingangen van de DMAC niet actief. Het device activeert zijn TxRQ-uitgang alleen, indien het gereed is om data te

verzenden of te ontvangen. Dit betekent dat er in de burst mode een aantal dummy cycli kunnen ontstaan, indien het device niet elke klokcyclus een byte kan overdragen (zie ook fig. 11.36). Een gevaar van de burst mode is dat de bussen te lang bezet worden gehouden door een device, waardoor andere devices mogelijk in de problemen raken. Als een disk eenmaal is aan het lezen of schrijven, mag de data-overdracht niet te veel vertraagd worden. De disk kan niet halverwege worden gestopt. Er wordt telkens een geheel blok data overgedragen. Werkt een ander device (bijvoorbeeld een andere disk drive) in burst mode, dan kunnen er problemen ontstaan bij het datatransport.

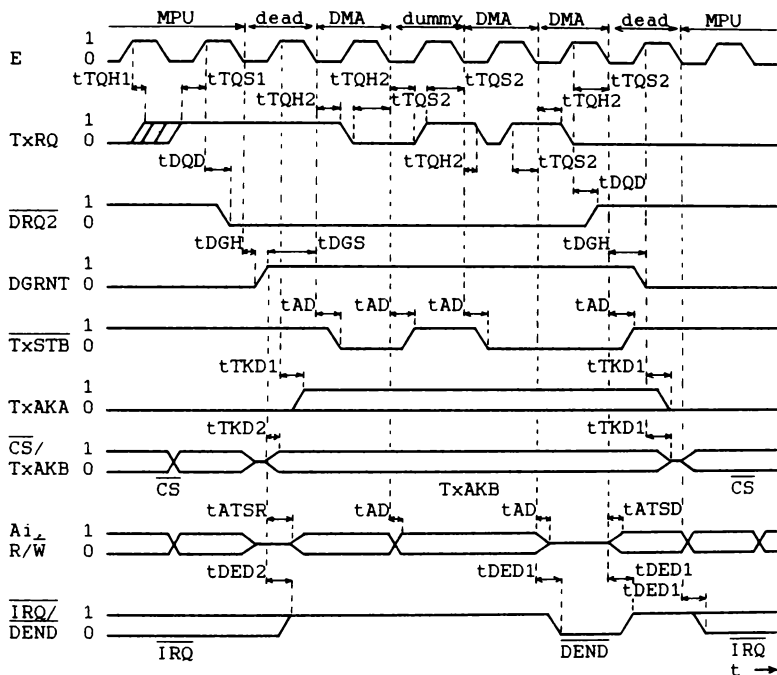


Fig. 11.36 Timing van de halt burst mode

Resumerend kunnen we de volgende voordelen en nadelen opnoemen van het gebruik van de DMA/bus request ingang en de halt-ingang van de processor voor het aanvragen van DMA transfers:

DMA/BREQ: + DGRNT wordt actief aan het einde van de huidige klokcyclus. Er wordt dus snel gereageerd op de DMA-aanvraag.

- De interne processorklokken worden stilgezet. De processor deactiveert na 16 cycli het signaal DMA grant. Dan volgt een dode cyclus en een MPU-cyclus voor self-refresh.

- HALT: + De interne processorklokken lopen door, zodat er oneindig veel bytes aaneengesloten kunnen worden overgedragen. Er gaat hierbij geen tijd verloren aan dode cycli.
- De processor maakt eerst de huidige instructie af. Dit betekent dat er soms traag wordt gereageerd op een DMA-aanvraag. Denk bijvoorbeeld aan de tijdrovende push-instructie.

Zoals al eerder is opgemerkt moeten $\overline{\text{DMA/BREQ}}$ en $\overline{\text{HALT}}$ tenminste vanaf 200 ns voor de neergaande flank van E geldig zijn op de processor-ingangen. $\overline{\text{DRQ1}}$ en $\overline{\text{DRQ2}}$ zijn geldig na maximaal 375 ns na de opgaande flank van E. Dit is te laat bij een klokpulsbreedte van minimaal 450 ns. Dit houdt in dat deze signalen extern moeten worden gesynchroniseerd. Inlezen van deze signalen in een flipflop op de opgaande flank van Q levert signalen die wel tijdig aanwezig zijn. Bovendien zijn de DMA request signalen stabiel rond de opgaande flank van Q, zodat de flipflop zelf correct wordt ingelezen.

11.8 Samenvatting

Besturingssignalen die niet intern in een bepaalde component worden gesynchroniseerd, moeten voldoen aan de voor die component gespecificeerde setuptijden en holdtijden. Gebeurt dit niet, dan kan de component in een ongedefinieerde toestand raken.

Besturingssignalen die intern in een component worden gesynchroniseerd, hoeven niet noodzakelijkerwijze te voldoen aan de setuptijden en holdtijden. In het ongunstigste geval wordt de gewenste actie een klokperiode later uitgevoerd.

Datasignalen moeten in principe eveneens aan de gespecificeerde setuptijden en holdtijden voldoen. Dit kan worden bereikt met behulp van externe synchronisatieschakelingen.

De synchronisatie van datasignalen kan echter ook plaatsvinden met behulp van strobe-signalen of handshake-signalen of softwarematig. In deze gevallen hoeft geen rekening te worden gehouden met setuptijden en holdtijden.

Om de timing sluitend te maken, kan men gebruik maken van het volgende:

- Binnen een IC geldt er een zekere correlatie tussen de eigenschappen. De ene vertraging zal niet maximaal zijn en een andere tegelijk minimaal.
- De diverse delaytijden kunnen statisch worden opgeteld,

- respectievelijk afgetrokken.
- Toepassen van buffer gates kan leiden tot kleinere buslooptijden en derhalve ook tot kleinere looptijdverschillen.
- Verlagen van de klokfrequentie laat grotere buslooptijden toe.
- Met behulp van delayschakelingen kunnen signalen ten opzichte van elkaar in de tijd worden verschoven.
- Met behulp van one-shots kunnen pulsen worden versmald of verbreed.

11.8 Opgaven

- 1 In welk geheugengebied bevindt zich de SRAM van fig. 11.19?
- 2 Hoe groot mag de delay van de databus maximaal bedragen bij het gebruik van een PIA?
Gegeven: $3 \text{ ns} < \text{delay controlbus} < 35 \text{ ns}$
 $4 \text{ ns} < \text{delay adresbus} < 40 \text{ ns}$
- 3 Hoeveel klokcycli kost het overdragen van 100 bytes via burst mode DMA minimaal bij DMA-aanvragen via
 - a. $\overline{\text{DMA}}/\overline{\text{BREQ}}$
 - b. $\overline{\text{HALT}}$

12 MODERNE PROGRAMMEER- TECHNIIEKEN

12.1 Specifieke doelstelling

Nadat u hoofdstuk 12 hebt bestudeerd, moet u in staat zijn tot het

- * Omgaan met lokaal en globaal geheugen.
- * Overdragen van parameters via de stack.
- * Schrijven van positie-onafhankelijke programma's.
- * Ontwerpen van reentrant routines.
- * Realiseren van recursieve routines.

12.2 Lokaal en globaal geheugen

12.2.1 Begrippen

In hoofdstuk 8 hebt u het belang gezien van modulaair opgebouwde programma's. Ontwerpen, testen, wijzigen en documenteren van de diverse modules waaruit een programma is opgebouwd, is eenvoudiger dan dit in één keer te doen voor het programma in zijn geheel.

Modules zijn delen van een programma, die een welomschreven taak uitvoeren.

Bij modulaair programmeren treden lokale variabelen op. Lokaal is plaatselijk. Een lokale variabele is een variabele die alleen binnen een bepaalde module wordt gebruikt.

Een voorbeeld van een lokale variabele is TELLER. Deze houdt bij hoe vaak de lus in een wachtroutine nog moet worden doorlopen, teneinde de juiste vertragingstijd te krijgen. Buiten de wachtroutine wordt TELLER niet gebruikt.

Aangezien variabelen doorgaans opgeslagen worden in het geheugen, spreekt men dienovereenkomstig van lokaal geheugen (local storage).

De lokale variabele wordt alleen gebruikt binnen een bepaalde module. Dit betekent dat hij alleen van belang is tijdens de uitvoering van die module. De betreffende geheugenruimte is derhalve alleen nodig tijdens het uitvoeren van de module. Deze

geheugenruimte kan na het uitvoeren van de module voor andere doeleinden worden gebruikt. Vandaar dat er ook wel wordt gesproken van tijdelijk geheugen (temporary storage).

Naast de lokale variabelen zijn er globale variabelen. Globaal is wereldwijd. Het gaat hier om variabelen, die door meerdere modules worden gebruikt.

Globale variabelen zijn bijvoorbeeld meetwaarden, die door een invoermodule worden binnengehaald en door een verwerkingsmodule worden gebruikt bij het nemen van allerlei beslissingen.

De ruimte waarin de globale variabelen worden opgeborgen, heet globaal geheugen (global storage). Deze geheugenruimte mag buiten een bepaalde module niet voor andere zaken worden gebruikt. Ook andere modules gebruiken immers de informatie, die erin is opgeslagen. Het gaat hierbij kennelijk om permanente geheugenruimte. Global storage wordt derhalve ook wel permanent storage genoemd.

Voor de opslag van gegevens, zowel lokaal als globaal, kunnen registers, vaste geheugenlocaties of stacklocaties worden gebruikt.

12.2.2 Locals in registers

Registers zijn erg geschikt voor het opbergen van lokale variabelen. In een andere module kunnen dezelfde registers zonder meer voor heel andere zaken worden gebruikt. De oude inhoud gaat daarbij vanzelf verloren. Zijn er veel lokale variabelen, dan is het aantal registers hiervoor ontoereikend. De opslag moet dan plaatsvinden in vaste geheugenlocaties of op de stack.

12.2.3 Globals in registers

Registers zijn niet erg geschikt voor het opslaan van globale variabelen. Registers worden in de verschillende modules voor de meest uiteenlopende bewerkingen gebruikt. Daarbij gaat de aanwezige informatie verloren. Registers die globale informatie bevatten, kunnen niet voor andere doeleinden gebruikt worden, omdat de opgeslagen informatie in meerdere modules wordt gebruikt en derhalve bewaard moet blijven. Vanwege het beperkte aantal registers is dit nogal bezwaarlijk.

12.2.4 Locals in vaste geheugenlocaties

Er kan een vast geheugengebied worden gebruikt voor de opslag van lokale variabelen. Deze geheugenruimte moet voor een

bepaalde module worden gereserveerd. Het nadeel hiervan is dat deze ruimte niet kan worden opgeheven als hij niet meer voor de module nodig is. Op deze wijze wordt er erg verkwistend omgesprongen met geheugenruimte. Dit nadeel kan ten dele worden opgeheven door deze ruimte ook door de andere modules te laten benutten voor hun lokale variabelen. Er moet dan een gebied permanent worden gereserveerd, dat zodanig groot is dat het grootste aantal lokale variabelen erin past. Eigenlijk wordt op deze wijze een globaal werkgeheugen gereserveerd. Dit werkgeheugen wordt door alle modules gebruikt voor hun lokale variabelen. Het is dus een soort globaal kladblokgeheugen. Het reserveren van een kladblokgeheugen van 100 bytes ten behoeve van de (maximaal 100 bytes) lokale variabelen kan geschieden met behulp van de assembler directive

```
KLAD    RMB    100
```

Tijdens het assembleren zorgt de assembler ervoor, dat er 100 bytes worden overgeslagen. KLAD is hierbij het adres van de eerste byte.

12.2.5 Globals in vaste geheugenlocaties

Globale variabelen kunnen eenvoudig op vaste locaties in het geheugen worden opgeslagen. De benodigde geheugenruimte hiervoor kan met behulp van de assembler directive RMB worden gereserveerd, bijvoorbeeld:

```
GLOBAL  RMB    200          reserveer 200 bytes global
                               storage
```

Vanuit elke deel van het programma kunnen deze locaties worden bereikt met behulp van instructies als

```
LDA    GLOBAL+12    lees de dertiende global
STB    GLOBAL+8      berg de negende global op
```

Bedenk dat de eerste global zich op de locatie GLOBAL = GLOBAL+0 bevindt!

Hetzelfde kan worden bereikt via een pointer:

```
LDX    #GLOBAL      initialiseer de pointer
LDA    12,X          lees de dertiende global
STB    8,X           berg de negende global op
```

Weer een andere methode voor het reserveren van globals is het gebruik van het Direct Page Register. De globals moeten hierbij op een bepaalde bladzijde in het geheugen staan (zie ook paragraaf 3.6).

Zetten we de globals op pagina 20, dan moet het direct page register op het eerste (bovenste) niveau in het programma worden geïnitieëerd met behulp van de instructies

```
LDA    #20
TFR    A,DP
```

Vervolgens kunnen de globals weer worden geadresseerd met behulp van direct addressing (zie paragraaf 4.6):

LDA	<12	lees de dertiende global
STB	<8	berg de negende global op

Het Direct Page Register wordt hierbij als pointer gebruikt. Het bevat het basisadres van de globals.

De variabelen kunnen natuurlijk ook een symbolisch adres hebben. Dit verhoogt de leesbaarheid en het programmeergemak aanzienlijk. Staan de globals op pagina 20, dan kunnen de symbolische adressen als volgt worden toegekend:

ORG	\$1400	eerste adres van pagina 20 (= \$14)	
BTW	RMB	1	eerste global is BTW
PRIJS	RMB	1	tweede global is de gasprijs per m ³
DRUK	RMB	1	derde global is de druk in de radiatoren
KAMTEM	RMB	2	vierde global is de kamertemperatuur (2 bytes)
BUITEM	RMB	2	vijfde global is de buitentemperatuur (2 bytes)
enzovoorts.			

Toegang tot deze globals kan (na het initialiseren van het Direct Page Register) worden verkregen met instructies als

LDB	<DRUK	lees de druk
STX	<KAMTEM	berg de kamertemperatuur op

De assembler laat de high-byte van het adres weg bij het vertalen van direct addressing instructies in machinetaal! Direct addressing instructies zijn snel, omdat alleen de low-byte van het adres moet worden opgehaald uit het geheugen. De high-byte staat reeds in het Direct Page Register.

Bij multitasking systemen, waarbij verschillende afzonderlijke taken worden uitgevoerd, kan het operating system aan elke taak een global storage toewijzen op een verschillende pagina. Het operating system hoeft slechts te zorgen dat het Direct Page Register op de juiste wijze wordt ingesteld, voordat de besturing aan een bepaalde taak wordt overgedragen. De verschillende taken kunnen eventueel schijnbaar tegelijk worden uitgevoerd, doordat het operating system elke taak bij toerbeurt gedurende een bepaalde tijd (time slice) laat lopen. Men spreekt hier van time-sharing. Het operating system past in ons voorbeeld telkens de inhoud van het Direct Page Register aan.

12.2.6 Locals op de stack

Een zeer flexibele methode voor het opbergen van lokale variabelen is gebruik te maken van de stack. Er wordt nu ruimte op de stack vrijgemaakt voor de opslag van de lokale variabelen. Is deze ruimte niet meer nodig (aan het einde van de module), dan wordt deze weer opgegeven.

Aan het begin van de module wordt ruimte op de stack gereserveerd ten behoeve van de lokale variabelen:

```
LEAS    -100.S           reserveer 100 local bytes op de
                           stack
```

Door de inhoud van de stack pointer met 100 te verlagen, worden in feite 100 locaties op de stack overgeslagen. Op deze locaties, die zich bevinden van adres (S) t/m adres (S)+99, worden de locals opgeborgen (zie fig. 12.1). Ze zijn (binnen die module) te bereiken met instructies als

```
LDA     12.S             laad de dertiende lokale
                           variabele
STB     8.S              berg de negende lokale
                           variabele op
```

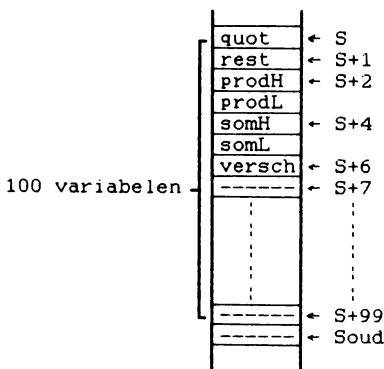


Fig. 12.1 Opslag van variabelen op de stack

Aan het einde van de module wordt de local storage opgegeven met de instructie:

```
LEAS    100.S           hef de local storage op
```

Deze ruimte kan nu weer voor andere zaken worden gebruikt (bijvoorbeeld voor het redden van registerinhouden).

Realiseer u dat interrupts geen invloed hebben op de local storage op de stack. Bij een interrupt worden weliswaar registerinhouden op de stack opgeborgen, maar dit gebeurt telkens op lagere stackadressen, dus niet in de local storage! Bij RTI worden de registers weer vanaf de stack gevuld, waarbij de stack pointer weer op de oorspronkelijke plaats wordt

teruggezet. Bij het hervatten van de uitvoering van de module is alles weer bij het oude.

Ook bij locals op de stack kunnen symbolische adressen worden gebruikt. Deze moeten nu op de volgende wijze worden toegekend:

QUOT	EQU	0	eerste local is een quotiënt
REST	EQU	1	tweede local is de rest van de deling
PROD	EQU	2	derde local is een produkt (2 bytes)
SOM	EQU	4	vierde local is een som (2 bytes)
VERSCH	EQU	6	vijfde local is een verschil

Let op dat SOM de waarde 4 krijgt toegewezen in plaats van 3, omdat de vorige local, PROD, twee bytes omvat! Zo ook krijgt VERSCH de waarde 6 (en niet 5), omdat de vorige local (SOM) uit twee bytes bestaat.

Nu kunnen de locals worden bereikt via instructies als

LDB	REST,S	lees de rest
STX	PROD,S	berg het produkt op

Enige omzichtigheid is evenwel geboden bij het gebruik van de stack voor de opslag van locals.

Wordt in de module MODA1 (zie fig. 12.2) een subroutine SUBMOD aangeroepen, dan wordt hierbij de stack pointer met twee verlaagd. Het returnadres wordt immers op de stack gezet. Bij terugkeer uit de subroutine (bij de instructie RTS) wordt het returnadres weer van de stack gehaald, waarbij de stack pointer met twee wordt verhoogd. Hierna kunnen de locals weer op dezelfde wijze als voor de subroutine call worden geadresseerd. Tot dusver is er niets aan de hand.

Wilt u echter binnen de subroutine SUBMOD de locals van de module MODA1 (van een bovenliggend niveau) gebruiken, dan dient u de offset met twee te verhogen:

SUBA1	LEAS	-35,S	creëer ruimte voor 35 locals
--			
--			
	STX	PROD,S	berg het produkt op
--			
	BSR	SUBMOD	
RETMA1	--		
	LDD	SOM,S	haal de som op
--			
EINDE	LEAS	35,S	hef de locals op
	RTS		
SUBMOD	--		
--			
	LDY	PROD+2,S	haal het produkt op
--			
	RTS		

SUBA1 is hier als subroutine aangeroepen vanuit MODA.

Een groter probleem treedt op als SUBMOD op twee verschillende niveaus wordt aangeroepen.

Veronderstel dat een programma PROG bestaat uit een aantal modules, bijvoorbeeld MODA, MODB en MODC. MODA bestaat uit de submodules SUBMOD, SUBA1 en SUBA2, die als subroutines zijn geschreven. SUBA1 op zijn beurt roept eveneens de subroutine SUBMOD aan. Er ontstaat zo de situatie van fig. 12.2, waarin SUBMOD op twee verschillende niveaus wordt aangeroepen.

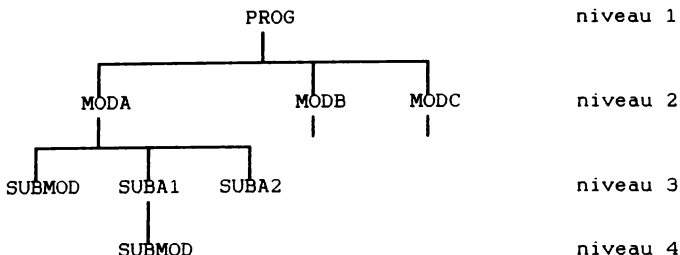


Fig. 12.2 Diverse niveaus in de programmatuur

Nu is de adressering van het produkt in SUBMOD niet meer mogelijk. Op niveau 3 zou dit moeten geschieden met de operand `PROD+2,S`. Op niveau 4 echter zou het met `PROD+4,S` moeten gebeuren. In feite wordt hier de fout gemaakt dat de variabele `PROD` als local is gedefinieerd, terwijl hij een global moet zijn. Hij wordt immers niet alleen binnen SUBMOD gebruikt, maar ook binnen een andere module SUBA1. `PROD` moet derhalve worden gedefinieerd als een global voor de modules SUBMOD en SUBA1. Het creëren van ruimte voor deze global moet op niveau 2 geschieden.

Een volgend probleem treedt op als u na een `PSHS`-instructie een local wilt adresseren. U dient dan de offset te vergroten met het aantal bytes dat gepushed is. Dit is afhankelijk van hoeveel en welke registerinhouden op de stack worden geplaatst. Na de `PULS`-instructie geldt deze verhoging van de offset niet meer. Het is zaak bij het adresseren van locals nauwlettend in het oog te houden of de stack pointer verplaatst is en zo ja, hoe ver.

Een eenvoudige manier om dit probleem op te lossen is het definiëren van een (kleine) user stack en deze te gebruiken voor het redden van registerinhouden (met behulp van `PSHU`-instructies).

12.2.7 Globals op de stack

Ook global storage kan op de stack worden gedefinieerd. Dit moet gebeuren op de hoogste stackadressen. Het reserveren van globals op de stack gaat op dezelfde wijze als het creëren van

ruimte ten behoeve van locals:

LEAS	-350,S	creëer 350 bytes ten behoeve
		van globals

Dit moet gebeuren op het bovenste niveau in het programma (bij fig. 12.2 in PROG). Dan bevinden de globals zich op de hoogste stackadressen. Op lagere stackadressen is naar believe ruimte vrij voor locals en dergelijke. Denk bij dit laatste aan het redden van registerinhouden en aan parameteroverdracht via de stack (zie de paragrafen 8.5 en 12.3).

Omdat het nu erg moeilijk is de juiste offset te bepalen, wordt er vrijwel altijd een pointer gebruikt om het globale geheugen op de stack aan te wijzen. Om zonder die pointer te kunnen werken, moet u precies weten hoeveel de stack pointer is verplaatst op het moment dat u een global wilt ophalen. U moet dus weten hoeveel registerinhouden er zijn opgeborgen op de stack, hoeveel subroutine calls er zijn geweest, hoeveel parameters er zijn overgedragen via de stack en hoeveel locals er gecreëerd zijn op de stack. Dit is vrijwel ondoenlijk. De ontwerper van een bepaalde module moet hiertoe een vrij gedetailleerde kennis hebben van alle modules van de hogerliggende niveaus. Bedenk dat een module op zijn beurt weer uit andere modules kan bestaan, enzovoorts (zie ook fig. 12.2). Het voordeel dat een ontwerper zich (vrijwel) uitsluitend kan richten op zijn eigen module, gaat zo verloren.

Maakt een interrupt service routine gebruik van globals, dan is het onmogelijk om zonder pointer te werken. Een interrupt kan immers op elk moment optreden. Hierbij is niet bekend op welk niveau dit gebeurt. Met andere woorden, het is onmogelijk om de offset te bepalen, omdat niet bekend is, over hoeveel posities de stack pointer is verplaatst tussen het creëren van de global storage en het optreden van de interrupt. Dit geldt ook voor de module SUBMOD, die op verschillende niveaus gebruik maakt van de global KAMTEM.

De pointer die naar de global storage op de stack wijst, wordt stack mark genoemd. Hiervoor wordt doorgaans een pointerregister gebruikt. Het initialiseren van de stack mark gaat als volgt:

TFR	S,U	initialiseer de stack mark U
LEAS	-350,S	creëer ruimte voor 350 global bytes

Het creëren van de ruimte ten behoeve van de globale variabelen (en eventueel constanten) moet op het bovenste niveau plaatsvinden (dus op niveau 1 in fig. 12.2).

De globals zijn vanaf elk niveau bereikbaar via de operanden -1,U t/m -350,U. Het maakt niet uit hoeveel subroutine calls of PSHS-instructies uitgevoerd zijn.

De stack mark mag in principe niet worden gewijzigd.

De globals zijn nu op elk niveau adresseerbaar met instructies als

LDA	PRIJS,U	lees de gasprijs
STY	KAMTEM,U	berg de kamertemperatuur op

Hierin geldt bijvoorbeeld

BTW	EQU	-350	
PRIJS	EQU	-349	
DRUK	EQU	-348	
KAMTEM	EQU	-347	(2 bytes)
BUIITEM	EQU	-345	(2 bytes)

Het is erg handig om adressen van I/O-devices op de stack te plaatsen. Binnen de diverse modules hoeft dan niet bekend te zijn via welke adressen de I/O bereikbaar is. Met andere woorden de ontwerpers van de programmatuur hoeven niet te weten hoe de hardware-configuratie er uit ziet. Dezelfde programma's kunnen zonder adresvertaling worden uitgevoerd in verschillende machines, mits de I/O-adressen op dezelfde posities in een globale tabel staan, die via dezelfde pointer (bijvoorbeeld U) wordt bereikt.

De I/O-adressen kunnen bijvoorbeeld als volgt in de global storage zijn opgenomen:

P/DDRA	EQU	-200	adres van PRA/DDRA van de PIA
P/DDRb	EQU	-198	adres van PRb/DDRb van de PIA
CRA	EQU	-196	adres van CRA van de PIA
CRb	EQU	-194	adres van CRb van de PIA
C/SR	EQU	-192	adres van CR/SR van de ACIA
T/RDR	EQU	-190	adres van TDR/RDR van de ACIA

enzovoorts.

Bedenk dat het opbergen van een adres 2 bytes geheugenruimte kost.

In elke module en op elk niveau kunnen de diverse I/O-devices worden bereikt met indexed indirect instructies:

LDA	[P/DDRb,U]	lees informatie op PB van PIA
STB	[T/RDR,U]	zend via TDR van de ACIA

Indexed indirect addressing is hier nodig om via offset,U het adres van een bepaald I/O-register te bereiken. De indirectie levert vervolgens de inhoud van dat register.

Het is natuurlijk ook mogelijk alleen het basisadres van het I/O-device in de global storage op te nemen. Denk hierbij bijvoorbeeld aan de DMA-Controller, die 23 adresseerbare registers heeft:

DMAC	EQU	-188	basisadres van de DMAC
------	-----	------	------------------------

Op deze wijze is er minder globale geheugenruimte nodig (voor de DMAC 2 bytes in plaats van 46).

Het adresseren van de diverse DMAC-registers gaat nu als volgt:

LDY	DMAC,U	(Y) := basisadres van DMAC
STA	22,Y	schrijf in het Data Chain Register
TST	17,Y	test Channel Control Register 1
STD	6,Y	schrijf in Byte Count Register 1

Hierbij is het wederom verstandiger om de offset te vervangen door een symbolisch adres, bijvoorbeeld:

BCR1	EQU	6	offset van Byte Count Register 1
CCR1	EQU	17	offset van Channel Control Register 1
DCR	EQU	22	offset van Data Chain Register

zodat de instructies er als volgt uit zien:

STA	DCR,Y
TST	CCR1,Y
STD	BCR1,Y

Dit is beter leesbaar dan de cijfermatige offset!

Het register BCR1 is niet bereikbaar via

STD	[DMAC+BCR1,U]
-----	---------------

Dit wordt immers door de assembler vertaald in

STD	[-182,U]	immers DMAC+BCR1 = -188 + 6 = -182
-----	----------	------------------------------------

Nu is -182,U een locatie in de global storage op de stack, waarin in ieder geval niet het adres van BCR1 is opgeslagen. Het enige adres van de DMAC dat in de global storage staat, is immers het basisadres. Dit is tevens het adres van Address Register 1.

Het opnemen van alleen de basisadressen van de I/O-devices in de global storage heeft als nadeel dat er een extra pointer nodig is om de diverse registers van de I/O-devices te selecteren.

Bij het werken met een I/O-adressentabel is het niet nodig de programma's aan te passen, indien de hardware-configuratie verandert. Er kan worden volstaan met het corrigeren van de adressentabel.

Vaak zijn er globale variabelen van een onderliggend niveau nodig. U hebt hiervan reeds een voorbeeld gezien bij SUBMOD. SUBMOD gebruikte de variabelen QUOT, REST, PROD, SOM en VERSCH, die als globals moeten worden gedefinieerd, omdat SUBMOD op twee niveaus voorkomt (niveau 3 en niveau 4). Deze globals moeten derhalve worden gedefinieerd in MODA, opdat ze op niveau 3 zowel vanuit SUBMOD als vanuit SUBA1 zijn te gebruiken (zie fig. 12.2).

Natuurlijk kunt u nu een ander register kiezen als stack mark van een onderliggend niveau. Vanwege het beperkt aantal pointerregisters is dit niet altijd gewenst.

Een andere methode is de stack mark van het bovenliggende

niveau te gebruiken, nadat deze eerst is opgeborgen. Dit gaat dan als volgt:

PSHS	U	berg de stack mark van bovenliggend niveau op
TFR	S,U	initialiseer nieuwe stack mark
LEAS	-7,S	creëer globals voor onderliggend niveau

Definiëren we in MODA

QUOT	EQU	-7	(1 byte)
REST	EQU	-6	(1 byte)
PROD	EQU	-5	(2 bytes)
SOM	EQU	-3	(2 bytes)
VERSCH	EQU	-1	(1 byte)

dan kunnen deze variabelen gebruikt worden in MODA en in elke onderliggende module (SUBMOD, SUBA1 en SUBA2 op niveau 3, maar ook in SUBMOD op niveau 4). Hierbij kan overal dezelfde offset ten opzichte van de nieuwe stack mark worden gebruikt.

Fig. 12.3 toont de opbouw van de stack, zoals deze eruit ziet in SUBMOD op niveau 4. Hierin zijn S(i) en U(j) de waarden die S en U hebben bij het verlaten van niveau i, respectievelijk niveau j. Voordat het programma PROG wordt gestart, heeft de stack pointer de waarde S(0).

Hierbij wordt de stack successievelijk als volgt opgebouwd:

```

100 PROG      TFR      S,U          initialiseer de stack mark
105 *
110           LEAS     -350,S        creëer ruimte voor 350 global
115 *                               bytes
120           LEAS     -20,S         creëer ruimte voor 20 local
125 *                               bytes; S(1)
130           ---
140           LDA      #20
150           STA      BTW,U         berg btw op (global van niveau 1)
160           ---
170           STB      VAR1,S        berg var1 op (local van niveau 1)
180           ---
190 *****
200 *                               Begin van module MODA *
210 MODA      PSHS     U             berg de stack mark op
220           TFR      S,U          initialiseer nieuwe stack mark
225 *                               U(2)
230           LEAS     -7,S         creëer 7 global bytes;
235 *                               S(2):=U(2)-7
240           ---
250           MUL
260           STD      PROD,U        berg prod op (global van MODA)
270           ---
280           PSHS     Y
290           LDY      ,U            laad de stack mark van niveau 1
300           LDX      KAMTEM,Y      lees kamtem (global van niveau 1)
310           PULS     Y

```

```

320      ---
330      BSR      SUBMOD      ga naar SUBMOD (niveau 3)
340      ---
350  *-----
360  *-          Begin van module SUBA1      -
370      ---
380      BSR      SUBA1      ga naar SUBA1 (niveau 3)
390  TOMODA      ---
400      ---
410  *-          Einde van module SUBA1      -
420  *-----
430      ---
440  *+++++
450  *+          Begin van module SUBA2      +
460      ---
470      ---
480  *+          Einde van module SUBA2      +
490  *+++++
500      ---
510      TFR      U.S      hef de globals van MODA op en
520      PULS      U      herstel stack mark van niveau 1
530  *          Einde van module MODA      *
540  *-----
550      ---
560      STD      DRUK.U      berg druk op (global van
565  *          niveau 1)
570      ---
580      ---
585  *          hier bevinden zich tevens de
590      ---          modulen
590      ---          MODB en MODC
600      ---
610      LEAS      20.S      hef de locals van PROG op
620      TFR      U.S      hef de globals van PROG op
630      SWI
640  * Einde van PROG
650  *
660  *
670  SUBA1      LEAS      -4.S      creëer 4 locals van SUBA1
680      ---
690      STB      3.S      schrijf de vierde local van SUBA1
700      ---
710  *-----
720  * Begin van module SUBMOD op niveau 4 *
730      BSR      SUBMOD      ga naar SUBMOD (niveau 4)
740  RETSA1      ---
750      ---
760  * Einde van module SUBMOD op niveau 4 *
770  *-----
780      ---
790      LEAS      4.S      hef de locals van SUBA1 op
800      RTS
810  *

```

```

820 *
830 SUBMOD LEAS -15,S      creëer 15 locals van SUBMOD
840 ---
850 LDD PROD,U           lees prod (global van MODA)
860 ---
870 STX 11,S             schrijf twaalfde local van SUBMOD
880 ---
890 RTS

```

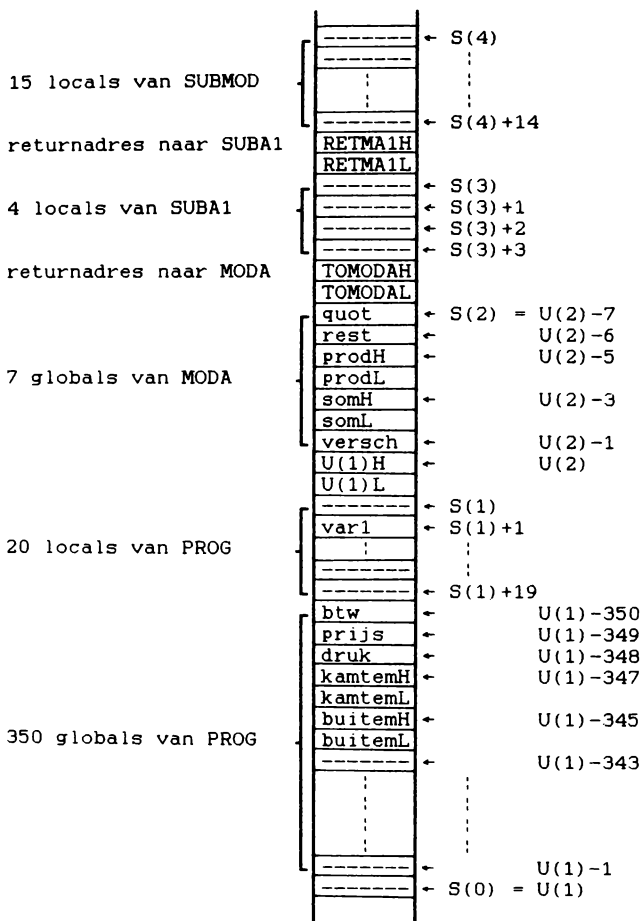


Fig. 12.3 Globals en locals op de stack

De regels 110 en 120 kunnen natuurlijk worden vervangen door de ene instructie

```
LEAS -370,S
```

waarmee ruimte wordt gecreëerd voor zowel 350 global bytes als 20 local bytes.

Vanwege de leesbaarheid is gekozen voor twee aparte instructies.

Met de instructies van de regels 280 t/m 310 wordt toegang gekregen tot globals van het bovenliggende niveau. Dit kan op dezelfde wijze gebeuren elders binnen MODA, dus ook in SUBA1, SUBMOD en SUBA2.

Bevat register Y geen zinvolle informatie, dan kunnen de push- en de pull-instructie worden weggelaten. Indien er vaak toegang tot de bovenliggende global storage moet worden verkregen, kan het handiger zijn om Y als stack mark te kiezen voor de globals van MODA. Dit is het geval als Y binnen MODA niet of weinig voor andere doeleinden gebruikt moet worden. De globals van het bovenste niveau kunnen nu in het gehele programma via stack mark U worden bereikt. Moet Y voor andere zaken worden gebruikt, dan moet de inhoud van Y (de stack mark van het onderliggende niveau) tijdelijk worden opgeborgen (bijvoorbeeld met behulp van push- en pull-instructies).

Indien er geen toegang tot de bovenliggende global storage moet worden verkregen, spreekt het vanzelf om U ook als stack mark voor de lagere globals te kiezen.

Overigens kan in principe ook register X of Y als stack mark worden gebruikt.

Aan het einde van MODA moeten de globals van MODA worden opgeheven (regel 510). Ze zijn immers in de rest van het programma niet nodig. In regel 520 wordt de stack mark van niveau 1 hersteld.

Hierna zijn in PROG de globals van het bovenste niveau weer te bereiken via stack mark U (regel 560). Dit geldt dus ook in MODB en in MODC.

Regel 610 kan in principe vervallen. Door (in regel 620) de stack pointer weer boven de globals van PROG te laten wijzen, zijn tevens de onderliggende locals opgeheven. De stack pointer wijst nu weer naar de oorspronkelijke locatie.

12.3 Parameterverdracht

12.3.1 Algemeen

In paragraaf 8.5 hebt u reeds gezien dat de overdracht van gegevens van en naar modulen, subroutines en macro's als volgt kan geschieden:

1. via registers
2. via vaste geheugenplaatsen
3. via de stack
4. via een pointer

- De pointer kan hierbij zijn:
- een 16-bits register
 - het direct page register
 - 2 vaste geheugenlocaties
 - 2 stacklocaties

De interface, de input en de output, van een module, subroutine of macro is bepaald door (zie ook fig. 12.4):

- het aantal en de eigenschappen van de invoerparameters.
Het gaat hierbij om de gegevens die in een module worden ingevoerd.
- het aantal en de eigenschappen van de uitvoerparameters.
Dit zijn de gegevens die door de module (als resultaat) worden uitgevoerd.

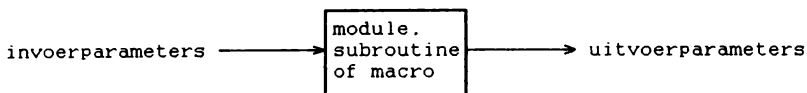


Fig. 12.4 Parameteroverdracht

De parameteroverdracht via de stack is nog niet nader uitgelegd. In paragraaf 12.3 zal worden uiteengezet hoe de diverse methoden van parameteroverdracht worden toegepast en wat de voordelen en de nadelen zijn. De wijze van toepassen wordt getoond aan de hand van voorbeeld 12.1.

Voorbeeld 12.1

In de subroutine SOMPRO wordt de som en het produkt van twee 1-byte unsigned getallen (TAL1 en TAL2) bepaald. Deze twee getallen zijn door het aanroepende programma reeds in de accumulatoren A en B geplaatst. De som moet in register X komen te staan en het produkt in Y.

12.3.2 Parameteroverdracht via registers

Vaak staan de getallen, die verder verwerkt moeten worden, in registers. Het zijn meestal resultaten van voorgaande bewerkingen en deze vinden bijna altijd in registers plaats. Het overdragen van parameters via registers gaat derhalve snel en eenvoudig. De parameters hoeven niet eerst elders te worden opgeborgen.

De parameteroverdracht via registers gaat voor de subroutine SOMPRO als volgt in zijn werk:

```

100          ORG      $4000
110 MAIN     ---
120          ---
                                     (TAL1 staat in A en TAL2 in B)
  
```

```

130      ---
140      BSR      SOMPRO
150 RETURN      ---
160      ---
170      SWI
180 *
300 *****
310 * Subroutine SOMPRO *
320 *****
330 SOMPRO  TFR      D,Y      kopieer TAL1 en TAL2 in Y
340      CLRA      (D) := TAL2
350      TFR      D,X      (X) := TAL2
360      TFR      Y,D      (A) := TAL1 en (B) := TAL2
370      EXG      A,B      (A) := TAL2 en (B) := TAL1
380      ABX      (X) := TAL2 + TAL1
390      MUL      (D) := TAL1 * TAL2
400      EXG      D,Y      (Y) := produkt
410      RTS

```

In regel 330 worden (A) = TAL1 en (B) = TAL2 opgeborgen in register Y. In regel 340 wordt (A)=0 gemaakt, zodat (D) = (B) = TAL2 wordt. Door dit in register X op te bergen (regel 350) en vervolgens (B) te laden met TAL1 (regel 360 en 370) kan de som van de twee unsigned getallen worden bepaald met de instructie ABX (regel 380). Doordat in regel 370 de instructie EXG is toegepast en niet TFR, zijn TAL1 en TAL2 in de accumulatoren behouden gebleven. Ze zijn alleen van plaats verwisseld. Derhalve kan in regel 390 de vermenigvuldiging worden uitgevoerd met de instructie MUL. Door in regel 400 de instructie EXG te gebruiken in plaats van TFR, blijven TAL1 en TAL2 behouden in de accumulatoren A en B.

U ziet dat er geen geheugenacces plaats hoeft te vinden. Derhalve is de parameteroverdracht via registers erg snel. Bij gebrek aan vrije registers moeten er registerinhouden tijdelijk worden gered (bijvoorbeeld op de stack), zodat dan enige geheugenacces nodig is.

Een tweede voordeel is dat het programma positie-onafhankelijk is. Zou het programma worden geladen vanaf adres \$5000 in plaats vanaf adres \$4000, dan wordt het programma zonder meer goed uitgevoerd.

Samenvattend geldt voor parameteroverdracht via registers:

```

Voordelen:  + eenvoudig toe te passen
              + snel
              + positie-onafhankelijk
              + kost geen geheugenruimte
Nadeel:     - beperkt aantal parameters overdraagbaar

```

12.3.3 Parameteroverdracht via vaste geheugenlocaties

Hier worden de over te dragen gegevens op vaste geheugenlocaties opgeborgen. Dit is eveneens erg eenvoudig toe te passen. Er treedt wel tijdverlies op met het opbergen en ophalen van de gegevens uit het geheugen. Het aantal parameters is niet beperkt.

De parameters worden bij voorbeeld 12.1 als volgt overgedragen:

100			ORG	\$4000
110	G	RMB	1	ruimte voor TAL1
120	H	RMB	1	ruimte voor TAL2
130	SOM	RMB	2	ruimte voor TAL1+TAL2
140	PROD	RMB	2	ruimte voor TAL1·TAL2
150	*			
160	MAIN	---		
170		---		(TAL1 staat in A en TAL2 in B)
180		STA	G	berg TAL1 op
190		STB	H	berg TAL2 op
200		---		
210		BSR	SOMPRO	
220	RETURN	---		
230		LDX	SOM	
240		LDY	PROD	
250		---		
260		SWI		
270	*			
300	*****			
310	* Subroutine SOMPRO *			
320	*****			
330	SOMPRO	CLRA		
340		LDB	G	(B) := TAL1
350		ADDB	H	(B) := TAL1 + TAL2
360		ADCA	#0	zet de carry in A
370		STD	SOM	berg de som op
380		LDA	G	(A) := TAL1
390		LDB	H	(B) := TAL2
400		MUL		(D) := TAL1·TAL2
410		STD	PROD	berg het produkt op
420		RTS		

De regels 230 en 240 zijn nodig, omdat het hoofdprogramma de som in X en het produkt in Y moet hebben.

Regel 330 en 360 dienen voor het opbergen van de carry in accumulator A. A is de meestwaardige byte van D. De carry kan ontstaan bij de optelling van de getallen TAL1 en TAL2 (regel 350). De instructie CLRA mag niet worden uitgevoerd na de optelinstructie ADDB, omdat CLRA de carry altijd reset.

Het programma is niet meer positie-onafhankelijk. Stel dat er ten gevolge van een wijziging van de hardware-configuratie geen geheugen meer aanwezig is vanaf adres \$4000, maar wel vanaf adres \$5000. Het programma kan dan worden geladen vanaf adres \$5000, maar werkt nu niet meer correct. De symbolische adressen G, H, SOM en PROD zijn door de assembler immers vertaald in \$4000, \$4001, \$4002 en \$4004. Op deze adressen kan niet meer worden geschreven en gelezen. Het programma kan nu alleen nog werken als het opnieuw wordt vertaald, nadat regel 100 is veranderd in

ORG \$5000

Wanneer het programma zich in ROM bevindt, is heeft opnieuw vertalen evenwel geen zin, omdat het programma niet meer kan worden geladen. De inhoud van de ROM kan immers niet worden gewijzigd.

Bij vaste geheugenlocaties daarentegen, blijft de ruimte gereserveerd (door de RMB-directive). Hij kan eventueel opnieuw worden gebruikt, maar het aantal parameters is vastgelegd door de RMB-directive.

Samenvattend kunnen we zeggen, dat voor de parameteroverdracht via vaste geheugenplaatsen geldt:

Voordelen: + eenvoudig toe te passen

+ aantal parameters is vrijwel onbeperkt

Nadelen: - traag door veel geheugenaccesses

- positie-afhankelijk

- niet erg flexibel

- geheugenruimte wordt niet erg efficiënt gebruikt

12.3.4 Parameteroverdracht via de stack

Bij deze methode worden de parameters, die moeten worden overgedragen, op de stack gezet, dan wel van de stack gehaald. Dit betekent dat het programmeerdeel, dat een bepaalde module aanroept, de invoergegevens voor die module op de stack moet zetten. Daarentegen moet het programmeerdeel waarvoor de uitvoergegevens van een bepaalde module bestemd zijn, deze van de stack halen. De module zelf moet zijn invoergegevens van de stack halen en zijn uitvoergegevens op de stack opbergen.

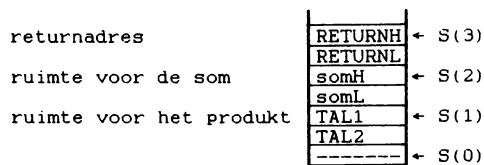


Fig. 12.5 Opbouw van de stack bij parameteroverdracht via de stack bij voorbeeld 12.1

Het overdragen van gegevens via de stack is tamelijk complex. Het verdient hierbij aanbeveling de opbouw van de stack te tekenen (zie fig. 12.5).

Bij parameteroverdracht bij subroutines treden er extra complicaties op:

- De offset ten opzichte van de stack pointer is binnen de subroutine 2 groter dan daarbuiten. Dit komt doordat het terugkeeradres op de stack staat, waardoor de stack pointer na de subroutine call met 2 is verlaagd.
- Indien aan het begin van de subroutine registerinhouden op de stack worden gezet, wordt de stack pointer verlaagd met het aantal bytes dat op de stack wordt opgeborgen. De offset binnen de subroutine moet dan met dat aantal worden verhoogd. Dit geldt ook bij modules.
- Indien het aantal uitvoergegevens groter is dan het aantal invoergegevens van een subroutine, moet het aanroepende programma ervoor zorgen dat er voldoende ruimte op de stack wordt gereserveerd. Gebeurt dit niet, dan wordt het terugkeeradres naar het aanroepende programma vernield. Het terugkeeradres staat immers ook op de stack.

Aan de hand van voorbeeld 12.1 wordt dit alles verduidelijkt. Fig. 12.5 geeft de opbouw van de stack bij de overdracht van de parameters. Hierbij worden de locaties met de getallen TAL1 en TAL2 overschreven met de waarde van het produkt.

Het programma ziet er nu als volgt uit, waarbij achter de instructies de waarde van de stack pointer is aangegeven.

100	ORG	\$4000	
110	SOM	EQU 0	offset (in hoofdprogramma) voor som
115	*		
120	PROD	EQU 2	offset (in hoofdprogramma) voor produkt
125	*		
130	TAL1	EQU 2	offset (in hoofdprogramma) voor TAL1
135	*		
140	TAL2	EQU 3	offset (in hoofdprogramma) voor TAL2
145	*		
150	*		
160	MAIN	---	S(0)
170		---	(TAL1 staat in A en TAL2 in B)
180		---	
190	PSHS	A,B	zet TAL1 en TAL2 op stack: S(1)
200	LEAS	-2,S	creëer 2 bytes ruimte voor de som; S(2)
205	*		
210		---	
220	BSR	SOMPRO	S(3)
230	RETURN	---	
240	PULS	X,Y	(X):=som, (Y):=produkt; S(0)
250		---	
260	SWI		

```

270 *
300 *****
310 * Subroutine SOMPRO *
320 *****
330 SOMPRO  CLRA
340          LDB    TAL1+2,S    (B) := TAL1
350          ADDB   TAL2+2,S    (B) := TAL1 + TAL2
360          ADCA   #0          zet de carry in A
370          STD    SOM+2,S     berg de som op
380          LDA    TAL1+2,S    (A) := TAL1
390          LDB    TAL2+2,S    (B) := TAL2
400          MUL     (D) := TAL1·TAL2
410          STD    PROD+2,S    berg het produkt op
420          RTS

```

Na het uitvoeren van de instructie op regel 200 heeft de stack pointer de waarde S(2). Er zijn nu twee locaties overgeslagen, zodat daarin de waarde van het produkt kan worden opgeborgen.

Er mag nooit data worden opgeborgen op een locatie met een lager adres dan de inhoud van de stack pointer. Deze informatie zou verloren gaan, zodra er een interrupt binnenkomt. Bij een interrupt wordt immers altijd informatie op de stack gezet (tenminste de inhoud van Condition Code Register en Program Counter). Een interrupt kan in principe op elk moment binnenkomen, waarbij een NMI-interrupt niet te blokkeren is.

Na de subroutine call (regel 220) heeft de stack pointer de waarde S(3). Bij terugkeer in het aanroepende programma staat hij weer terug op S(2). Na het uitvoeren van de pull-instructie (regel 240) is de inhoud van de stack pointer met 4 verhoogd, zodat hij weer de beginwaarde S(0) heeft.

Bij deze pull-instructie wordt eerst X geladen vanaf de stack en daarna Y, zodat (X) := som en (Y) := produkt. Indien de som en het produkt in omgekeerde volgorde op de stack stonden, hadden we in plaats van de ene pull-instructie twee afzonderlijke pull-instructies moeten gebruiken om de som in X en het produkt in Y te krijgen. Eerst hadden we dan de instructie PULS X en vervolgens PULS Y moeten toepassen!

In de subroutine (regel 340 t/m 410) is de offset voor de variabelen op de stack met twee verhoogd, opdat de juiste variabelen worden geadresseerd.

Het grote voordeel van de parameteroverdracht via de stack is dat dit positie-onafhankelijk gebeurt. De data is gerelateerd aan de stack pointer en niet aan vaste geheugenlocaties. Wordt het programma elders in het geheugen geladen, dan blijft het toch correct functioneren, omdat er geen vaste geheugenlocaties zijn gebruikt.

In paragraaf 12.4 zult u zien hoe de stack pointer op een positie-onafhankelijke wijze kan worden geïnitieerd.

Parametertransfer via de stack is erg flexibel. Er wordt juist zoveel geheugenruimte gebruikt als voor een bepaalde parameteroverdracht nodig is. Zijn de parameters niet meer nodig, dan wordt de gereserveerde ruimte vrijgegeven en kan deze voor een andere transfer (met een ander aantal parameters) opnieuw worden gebruikt. Men spreekt hier van een dynamische geheugentoewijzing. Dit is vooral handig bij het overdragen van tabellen met variabele lengten. Het gereserveerde geheugen wordt groter en kleiner afhankelijk van de benodigde ruimte. Het gereserveerde geheugengebied verandert tijdens de uitvoering van het programma. Er wordt hierbij dus op erg efficiënte wijze omgegaan met geheugenruimte.

Samenvattend gelden de volgende voordelen en nadelen bij het gebruik van de stack voor parametertransfer:

- Voordelen:
- + positie-afhankelijk
 - + aantal parameters is groot, mits de stack voldoende groot is
 - + erg flexibel
 - + efficiënt gebruik van geheugenruimte
- Nadelen:
- tamelijk complex
 - traag door veel stackaccess (met push-, pull- en indexed addressing load- en store-instructies)
 - er blijft minder stackruimte over voor subroutines en interrupt service routines

12.3.5 Parameteroverdracht via een pointer

Bij deze methode van parameteroverdracht staan de benodigde gegevens zelf in het geheugen. In principe is dit een vast geheugengebied. Indien de stack zou worden gebruikt voor de opslag van de gegevens, is er geen pointer meer nodig. Hierbij is de stack pointer immers impliciet de geheugenpointer. Het verschil met de parameteroverdracht via vaste geheugenlocaties is, dat nu niet alle adressen worden overgedragen, maar alleen het beginadres (of eindadres) van het geheugengebied. De gegevens worden niet rechtstreeks, maar via een pointer met behulp van offsets geadresseerd. In tegenstelling tot bij de parameteroverdracht via vaste geheugenlocaties hoeven niet alle adressen van de parameters in de andere module bekend te zijn, maar alleen het begin- of eindadres (en het aantal en de soort parameters).

De pointer kan hierbij zijn

- een 2-bytes register
- het direct page register
- twee vaste geheugenlocaties
- twee stacklocaties

Er is weinig verschil in de toepassing van deze drie methoden. Het ligt het meest voor de hand een register als pointer te kiezen.

In alle vier de gevallen is het programma positie-afhankelijk als de gegevens zelf in een vast geheugengebied staan. Dit geheugengebied blijft in alle gevallen in het gehele programma gereserveerd. In die zin is de methode weinig verschillend met de parameteroverdracht via vaste geheugenplaatsen. Er wordt weinig efficiënt met de gebruikte geheugenruimte omgesprongen.

Indien de pointers wijzen naar geheugenlocaties die gekoppeld zijn aan het programma en die derhalve met het programma mee worden verplaatst, kan positie-onafhankelijk worden gewerkt.

De parametertransfer via een pointer is echter vooral geschikt bij het op eenvoudige wijze overdragen van tabellen met gegevens. Een tabel kan met behulp van auto-increment-instructies worden gelezen of gevuld.

Alleen het beginadres van de tabel wordt meegedeeld. Op de eerste locatie kan eventueel de lengte van de tabel zijn vermeld. Een andere manier is de tabel af te sluiten met een bepaald karakter.

Het programma is traag ten gevolge van de te gebruiken indexed addressing methode voor het ophalen of opbergen van de data, behalve als het direct page register als pointer wordt gebruikt.

Samenvattend gelden de volgende voordelen en nadelen bij de parameteroverdracht via een pointer:

- Voordelen:
- + eenvoudig toe te passen, vooral bij tabellen
 - + aantal gegevens is vrijwel onbeperkt
 - + overdracht van de pointer kost slechts twee bytes (register of geheugenruimte)
 - + kan positie-onafhankelijk zijn
- Nadelen:
- traag door veel geheugenaccesses
 - niet erg flexibel
 - geheugenruimte wordt niet erg efficiënt gebruikt

Een 16-bits register als pointer

De oplossing van voorbeeld 12.1 met parametertransfer met een pointerregister is als volgt:

100	ORG	\$4000	
110	BUFFER	RMB	4
115	*		reserveer 4 bytes
120	SOM	EQU	0
125	*		geheugenruimte
130	PROD	EQU	2
135	*		offset (in hoofdprogramma) voor
140	TAL1	EQU	2
145	*		produkt
			offset (in hoofdprogramma) voor
			TAL1

150	TAL2	EQU	3	offset (in hoofdprogramma) voor TAL2
155	*			
160	*			
170	MAIN	---		
180		---		(TAL1 staat in A en TAL2 in B)
190		---		
200		LDU	#BUFFER	initialiseer pointer van databuffer
205	*			
210		STA	TAL1,U	zet TAL1 in buffer
220		STB	TAL2,U	zet TAL2 in buffer
230		---		
240		BSR	SOMPRO	
250	RETURN	---		
260		LDX	SOM,U	(X) := som
270		LDY	PROD,U	(Y) := produkt
280		---		
290		SWI		
295	*			
300	*****			
310	* Subroutine SOMPRO *			
320	*****			
330	SOMPRO	CLRA		
340		LDB	TAL1,U	(B) := TAL1
350		ADDB	TAL2,U	(B) := TAL1 + TAL2
360		ADCA	#0	zet de carry in A
370		STD	SOM,U	berg de som op
380		LDA	TAL1,U	(A) := TAL1
390		LDB	TAL2,U	(B) := TAL2
400		MUL		(D) := TAL1·TAL2
410		STD	PROD,U	berg het produkt op
420		RTS		

Bij deze oplossing moet worden opgemerkt dat het pointerregister U niet van inhoud mag veranderen. Is dit register toch voor andere zaken nodig, dan kan men zijn toevlucht zoeken tot het overdragen van de pointer via vaste geheugenlocaties of via de stack. Anders moet het pointerregister telkens gered worden.

Vaste geheugenlocaties als pointer

Worden vaste geheugenlocaties gebruikt, dan moet hiervoor ruimte worden gereserveerd in het geheugen. De oplossing komt er nu als volgt uit te zien:

10			ORG	\$4000
20	BUFFER	RMB	4	reserveer 4 bytes
25	*			geheugenruimte
30	SOM	EQU	0	offset (in hoofdprogramma) voor som
35	*			
40	PROD	EQU	2	offset (in hoofdprogramma) voor produkt
45	*			
50	TAL1	EQU	2	offset (in hoofdprogramma) voor

55	*			TAL1
60	TAL2	EQU	3	offset (in hoofdprogramma) voor
65	*			TAL2
70	POINT	RMB	2	reserveer ruimte voor de
75	*			datapointer
80	*			
90	MAIN	---		
100		---		(TAL1 staat in A en TAL2 in B)
110		---		
120		LDU	#BUFFER	initialiseer pointer van
125	*			databuffer
130		STA	TAL1,U	zet TAL1 in buffer
140		STB	TAL2,U	zet TAL2 in buffer
150		STU	POINT	berg de pointer op
160		---		
170		BSR	SOMPRO	
180	RETURN	---		
190		LDU	POINT	haal de pointer op
200		LDX	SOM,U	(X) := som
210		LDY	PROD,U	(Y) := produkt
220		---		
230		SWI		
240	*			
300	*****			
310	* Subroutine SOMPRO *			
320	*****			
330	SOMPRO	LDU	POINT	haal de pointer op
340		CLRA		
350		LDB	TAL1,U	(B) := TAL1
360		ADDB	TAL2,U	(B) := TAL1 + TAL2
370		ADCA	#0	zet de carry in A
380		STD	SOM,U	berg de som op
390		LDA	TAL1,U	(A) := TAL1
400		LDB	TAL2,U	(B) := TAL2
410		MUL		(D) := TAL1·TAL2
420		STD	PROD,U	berg het produkt op
430		RTS		

Deze oplossing is vrijwel identiek aan de oplossing waarbij de pointer rechtstreeks via een register werd overgedragen. De regels 150, 190 en 330 lijken overbodig. Ze zijn echter noodzakelijk, omdat in tussenliggende instructies (regel 160 en 180) de inhoud van register U wordt gewijzigd.

De stack pointer als datapointer

Het gebruik van de stack voor het overdragen van de pointer heeft geen voordelen of nadelen ten opzichte van de keuze van twee vaste geheugenlocaties. Regel 70 kan vervallen en de regels 150, 190 en 330 dienen als volgt te worden aangepast:

150	PSHS	U	berg de pointer op
-----	------	---	--------------------

190	PULS	U	haal de pointer op	
330	SOMPRO	LDU	2,S	haal de pointer op

Denk bij regel 330 aan de offset van 2 ten gevolge van het terugkeeradres RETURN, dat bij de subroutine call op de stack is gezet.

12.4 Positie-onafhankelijk programmeren

12.4.1 Inleiding

Een positie-onafhankelijk programma is een programma dat telkens correct functioneert, ongeacht de plaats in het geheugen waar het is geladen. Het programma hoeft hiertoe niet opnieuw te worden geschreven of opnieuw te worden vertaald. Het werkt zonder meer telkens correct. Een position-independent programma dat in ROM is geladen, zal zonder meer in elke module, ongeacht de adressering van die ROM, correct functioneren.

Een relocatable programma is eveneens herplaatsbaar. Hier worden de geheugenadressen echter tijdens het herladen toegekend door de relocatable loader. Dit kan alleen geschieden, indien het programma is vertaald met een zogenaamde relocatable assembler. Deze assembler genereert een lijst van labels (symbolische adressen), waaraan een voorlopig (relatief) adres wordt toegekend. De loader kan dan tijdens het laden bij al die voorlopige adressen een bepaalde offset optellen. Zodoende functioneert het programma na herladen correct. Alle adressen zijn aangepast aan het geheugengebied, waarin het programma wordt geladen.

In het source program moeten alle relocatable adressen worden opgegeven als labels. Absolute adressen kunnen niet worden veranderd. Dit is gedaan omdat er adressen kunnen zijn, die niet mogen worden veranderd als het programma elders in het geheugen wordt geladen. Denk hierbij aan de adressen van de I/O-devices.

Is een programma eenmaal in het geheugen geladen, dan werkt het niet meer, indien het betreffende geheugengebied van adres verandert. Dit betekent dat een ROM met een relocatable programma, niet in een ander geheugengebied mag worden geplaatst, als waarvoor hij bedoeld is. Is dit toch nodig, dan zal een nieuwe ROM moeten worden geprogrammeerd, waarbij deze wordt geladen met het relocatable programma, maar nu met een andere offset.

Een relocatable programma is dus niet positie-onafhankelijk!

Eenmaal geladen in het geheugen, is het positie-onafhankelijk geworden.

Resumerend kan gesteld worden dat een programma positie-onafhankelijk moet zijn, indien het in een ROM komt te staan, waarvan het basisadres niet bekend is, of waarvan het basisadres afhangt van de configuratie waarin die ROM wordt geplaatst.

Programma's moeten tenminste relocatable zijn, indien ze (bijvoorbeeld door een operating system) worden geladen in een geheugeengebied, dat op dat moment beschikbaar is.

Dit geldt ook voor modules, die voor algemeen gebruik nuttig zijn, en die derhalve op diverse adressen in het geheugen geladen zullen worden.

Position-independent code werkt altijd. Bij veranderen van de geheugenadressen hoeft er aan de code niets te wijzigen. De code hoeft dus niet te worden herladen.

Voorwaarde voor positie-onafhankelijkheid

Regel voor het ontwerpen van een positie-onafhankelijk programma:

Gebruik GEEN absolute adressen! Dit betekent:

- Gebruik geen extended addressing en geen extended indirect addressing.
- Indexed indirect addressing is alleen toegestaan, als de adressentabel positie-onafhankelijk is gevuld en alleen adressen bevat, die gerelateerd zijn aan het programmeergeheugen.
- Direct addressing mag alleen als het Direct Page Register op positie-onafhankelijke wijze is geladen en als de te gebruiken memory pages gerelateerd zijn aan de positie waar het programma wordt geladen.

Bedenk hierbij dat I/O-adressen altijd absolute adressen zijn. Ze zijn niet gerelateerd aan de adressen van het programma- en datageheugen.

Maak niet de vergissing dat het gebruik van symbolische adressen (labels) voldoende is voor het positie-onafhankelijk programmeren.

Het programma

		ORG	\$2000
CONST	FCB	18	
START	---		

	LDA	CONST	

	JMP	START	

is niet positie-onafhankelijk, omdat de assembler de labels voorziet van absolute adressen. CONST wordt immers vertaald met \$2000 en START met \$2001. Is het programma geladen aan het begin van het geheugen van een geheugenmodule met als basisadres \$2000, dan functioneert het programma juist.

Wordt het programma echter geladen vanaf bijvoorbeeld adres \$3000, dan werkt het niet meer. De constante wordt wederom opgehaald op adres \$2000, maar hij bevindt zich nu op adres \$3000. Bovendien moet aan het einde van het programma worden teruggesprongen naar het begin (dus naar \$3001). Er wordt echter gesprongen naar adres \$2001. Doordat er absolute adressen zijn gebruikt, is het programma positie-afhankelijk.

Om een programma positie-onafhankelijk te maken, moeten de volgende soorten opdrachten positie-onafhankelijk worden uitgevoerd:

1. springen in het programma
2. data ophalen en opbergen
3. wijzen naar data (pointers initialiseren)

Dit kan met behulp van de MC6809 worden gerealiseerd dankzij de volgende drie typen instructies:

1. branch- en long branch-instructies
2. program counter relative addressing instructies
3. load effective address instructies

12.4.2 Springen

Positie-onafhankelijk springen kan geschieden met behulp van relative addressing instructies (zie ook paragraaf 4.9). Dit zijn de branch en long branch instructies. Het effectieve adres, het adres waar naartoe wordt gesprongen, is de som van de operand en de inhoud van de program counter. Ten opzichte van de huidige positie in het programma, wordt er een aantal plaatsen vooruit (bij een positieve operand) of terug (bij een negatieve operand) gesprongen.

Voorbeeld 12.2 (Positie-onafhankelijk springen)

adres	instructie		
	ORG	\$2000	
\$2000	START	---	
:	:		
:	:		
\$2052	BRA	START	2-bytes instructie (zie appendix B)
\$2054	---		

De assembler vertaalt de operand START in een relatief adres. Indien de instructie BRA START is opgehaald door de microprocessor (tijdens het uitvoeren van het programma), wijst de program counter naar de volgende instructie, dus naar adres \$2054. Er moet worden teruggesprongen naar adres \$2000, met andere woorden de operand moet -\$54 (= - 84) zijn. Dit is \$AC in two's complement notatie. De assembler vertaalt de operand START dus in \$AC.

Wordt het programma geladen vanaf \$2000, dan zal bij het uitvoeren van de instructie BRA START worden gesprongen naar het adres \$2054+\$FFAC = \$2000. Dit klopt!
Denk eraan dat de tekenbits onder elkaar moeten staan bij het optellen van getallen, vandaar dat \$AC moet worden geschreven als \$FFAC (zie paragraaf 2.11).

Laden we het programma vanaf het adres \$3000, dan begint het programma op adres \$3000, terwijl de spronginstructie zich op de adressen \$3012 en \$3013 bevindt. Er wordt nu gesprongen naar adres \$3054+\$FFAC = \$3000, dus naar het begin van het programma. Dit is correct.
Inderdaad wordt de spronginstructie juist uitgevoerd, onafhankelijk van de positie van het programma in het geheugen.

Met de branch-instructies kunnen maximaal slechts 128 plaatsen terug en 127 plaatsen vooruit worden gesprongen. Dit is natuurlijk te weinig om grotere programma's positie-onafhankelijk te schrijven. De MC6809 kent long branch-instructies. Deze hebben een operand van 2 bytes, zodat er nu maximaal 32.768 plaatsen terug en 32.767 plaatsen vooruit kan worden gesprongen. Nu kan het gehele 64k-geheugen worden bereikt. Bedenk hierbij dat door adres \$FFFF met 1 te verhogen adres \$0000 wordt verkregen.

Met de instructies BSR en LBSR kunnen subroutines op positie-onafhankelijke wijze worden bereikt.

12.4.3 Data ophalen en opbergen

Met behulp van program counter relative addressing instructies kan op positie-onafhankelijke wijze data worden geadresseerd (zie ook paragraaf 4.7.5). Deze adresseringstechniek werkt hetzelfde als de relative addressing methode. In beide gevallen wordt het effectieve adres bepaald door de operand op te tellen bij de inhoud van de program counter.

Voorbeeld 12.3 (Positie-onafhankelijk lezen van een constante)

adres	instructie		
	ORG	\$2000	
\$2000	START	---	
:	:		
\$200A	LDA	<CONST,PC	3-bytes instructie (zie appendix B)
\$200D	---		
:	:		
:	:		
\$2057	CONST	FCB	18

De operand van de load-instructie wordt door de assembler

vertaald in een relatief adres, een displacement (van 1 byte vanwege het teken "<"). De load-instructie bevat 3 bytes, zodat de program counter na het ophalen van die instructie naar adres \$200D wijst. De displacement is derhalve \$2057-\$200D = \$004A = \$4A.

Bevindt het programma zich aan het begin van een ROM met als basisadres \$2000, dan wordt bij het uitvoeren van de load-instructie het effectieve adres \$200D+\$004A = \$2057 berekend. Dit betekent dat A wordt geladen met de inhoud van geheugenlocatie \$2057. Hier bevindt zich de constante 18 (= \$12).

Wordt het basisadres van de ROM gewijzigd in \$3000, dan staat de load-instructie op adres \$300A t/m \$300C en de constante bevindt zich op locatie \$3057. Tijdens de uitvoering van het programma berekent de microprocessor het effectieve adres van de load-instructie weer als volgt: (PC)+offset = \$300D+\$004A = \$3057. Hier bevindt zich inderdaad de gezochte constante CONST.

De program counter relative load-instructie is inderdaad positie-onafhankelijk. Daar de offset een 2-bytes getal kan zijn, kan elke geheugenlocatie worden bereikt. Met behulp van program counter relative store-instructies kan data op positie-onafhankelijke wijze worden opgeborgen.

12.4.4 Pointers initialiseren

Het verwijzen naar data in tabellen gebeurt via pointers. In position independent programma's moet het initialiseren van pointers op een positie-onafhankelijke manier geschieden.

Tot nu toe hebben we pointers geïnitieerd met behulp van immediate addressing load-instructies.

Voorbeeld 12.4 (Het initialiseren van een pointer met een immediate addressing instructie)

```
$2000 START ---
:
:
$2006      LDX      #TABEL      3-bytes instructie (zie
                                appendix B)
$2009      ---
:
:
$2032      STA      22,X
:
:
:
$2100 TABEL RMB    500
```

Aan het einde van het programma is er een ruimte van 500 bytes gereserveerd ten behoeve van een tabel.

De assembler vertaalt de operand van de LDX-instructie in het absolute adres (\$2100).

Tijdens de uitvoering van het programma, dat geladen is vanaf adres \$2000, wordt het indexregister X geladen met het getal \$2000. De microprocessor berekent verder het effectieve adres van de indexed addressing instructie STA als volgt: effectieve adres = (indexregister)+offset = \$2100+\$0016 = \$2116 (22 = \$16). Zodoende wordt de inhoud van A opgeborgen in de 23ste locatie van de tabel.

Wanneer het programma echter wordt geladen vanaf adres \$3000, wordt X tijdens het uitvoeren van het programma wederom geladen met het getal \$2000 en bepaalt de processor bij de STA-instructie weer het effectieve adres \$2100+\$0016 = \$2116. De tabel begint nu echter op de locatie \$3100, zodat de verkeerde geheugenplaats wordt geadresseerd. Het programma is dus niet positie-onafhankelijk. Dit komt doordat de pointer niet positie-onafhankelijk is geïnitieëerd.

Het positie-onafhankelijk initialiseren van de pointer kan evenmin gebeuren met de instructie

LDX TABEL.PC

Deze instructie is weliswaar positie-onafhankelijk, maar nu wordt niet het beginadres van de tabel in X geladen, maar de inhoud van de eerste twee locaties van de tabel.

De enige manier om een adres op positie-onafhankelijke manier te laden, is gebruik te maken van de load effective address instructie. Deze bestaat alleen als indexed addressing instructie. Ten behoeve van de initialisatie van de pointer moet hierbij de program counter worden gebruikt als indexregister. Dit betekent dat in feite de program counter relative adresseringsmethode wordt toegepast. Dit is een van de indexed addressing instructies.

Voorbeeld 12.5 (Het positie-onafhankelijk opbergen van een getal in een tabel)

Het programma van voorbeeld 12.4 komt er nu als volgt uit te zien:

```
$2000 START ---
:
$2006      LEAX   TABEL.PC      4-bytes instructie (zie
                                appendix B)
$200A      ---
:
$200F      STA    22,X
:
:
$2100 TABEL RMB   500
```


In dit geval wordt tijdens de uitvoering van het programma het effectieve adres berekend, dat in X wordt geladen met behulp van de load effective address instructie. De assembler heeft de operand vertaald in een relatief adres, namelijk in het getal $\$2100 - \$200A = \$00F6$. In dit geval mag er voor de operand niet $\$F6$ worden geschreven, omdat dit een negatief getal is, terwijl de offset positief is.

Is het programma geladen vanaf $\$2000$, dan wordt het indexregister tijdens de uitvoering van het programma geladen met het getal $\$200A + \$00F6 = \$2100$. Dit is inderdaad het beginadres van de tabel. Vervolgens wordt met de STA-instructie de locatie $(X) + \$16 = \2116 geadresseerd. Dit is de juiste plaats in de tabel.

Schuift alles 1024 locaties op (het programma begint dan op adres $\$3000$), dan wordt X tijdens het uitvoeren van het programma geladen met de waarde $\$300A + \$00F6 = \$3100$. Dit is het beginadres van de tabel. Ook de tabel is immers 1024 plaatsen opgeschoven. Vervolgens wordt de inhoud van A opgeborgen op locatie $(X) + \$16 = \3116 , hetgeen de 23ste positie in de tabel is.

Het programma is inderdaad positie-onafhankelijk geworden door het positie-onafhankelijk initialiseren van de pointer.

U hebt in voorbeeld 12.5 gezien dat ook met behulp van indexed addressing instructies op positie-onafhankelijke wijze geheugen wordt geadresseerd, mits het indexregister positie-onafhankelijk is geladen met een adres.

Ook push- en pull-instructies werken met relatieve adressen en dus positie-onafhankelijk. Er worden immers geheugenplaatsen geadresseerd, waarvan het adres in de stack pointer staat. Hierbij moet de stack pointer (U of S) met behulp van een LEA-instructie positie-onafhankelijk geladen zijn.

Voorbeeld 12.6 (Het positie-onafhankelijk initialiseren van de stack pointer)

Veronderstel dat een programma moet worden geladen in een 4 kbyte read/write geheugen. De stack moet zich op de hoogste adressen van dat geheugengebied bevinden. De stack pointer S wordt nu als volgt op de juiste wijze positie-onafhankelijk geladen:

```

                ORG      $2000
START          LEAS     <START,PCR    3-bytes instructie (zie
                                   appendix B)
                LEAS     4096,S
                ---
                ---

```

Met behulp van de eerste instructie wordt de stack pointer

geladen met het startadres van het programma, en wel op positie-onafhankelijke wijze. De operand wordt door de assembler vertaald in een offset ten opzichte van de program counter. Deze offset bedraagt -3 (= \$FD), ongeacht de waarde van de operand in de ORG-directive.

De inhoud van S wordt tijdens het uitvoeren van het programma geladen met het adres van de eerste instructie van het programma, onafhankelijk van het basisadres van het geheugen. Vervolgens wordt de inhoud van S met 4096 verhoogd, zodat S net buiten het 4 kbyte read/write memory wijst. Dit is goed, omdat de stack pointer bij een push-instructie eerst wordt verlaagd, alvorens er op de stack wordt geschreven. Op deze wijze wordt het eerste gegeven op het hoogste stackadres opgeborgen.

Het is mogelijk het direct page register als pointer te gebruiken. Deze pointer wijst dan altijd naar de eerste locatie van een geheugenpagina (van 256 bytes). Wordt het direct page register op positie-onafhankelijke wijze geladen, dan kan de direct adressering mode eveneens worden toegepast bij schrijven van positie-onafhankelijke programma's. Zo'n geheugenpagina kan worden gebruikt als local of als global storage, bijvoorbeeld voor het opslaan van tabellen.

Voorbeeld 12.7 (Positie-onafhankelijk laden van het direct page register)

Stel we willen de pagina, die direct volgt op het programma, gebruiken voor de opslag van een tabel. Als pointer wensen we het direct page register te gebruiken, dat op positie-onafhankelijke wijze moet worden geïnitialiseerd. Dit kan als volgt geschieden:

100	ORG	\$4000	deze directive is eigenlijk
110 *			niet van belang bij position-
115 *			independent code
120 BEGIN	LEAX	ACHTER,PC	zet eerste vrije adres na
125 *			programma
130	TFR	X,D	(positie-onafhankelijk) in D
140	TSTB		is dit niet het eerste adres
150	BEQ	EERSTE	van een geheugenpagina, wijs
160	INCA		dan naar volgende geheugen-
170 EERSTE	TFR	A,DP	pagina door (A) te verhogen;
175 *			zet (A) tenslotte in DP
180	---		
:	:		
:	:		
600	SWI		einde van het programma
610 ACHTER	RMB	1	reserveer een dummy byte ten
620 *			behoefte van het eerste adres
625 *			achter het programma
630	END	BEGIN	

In regel 140 wordt getest of de low-byte (dus de inhoud van B) van het adres van de eerste vrije geheugenlocatie achter het programma 0 is. Is dit zo, dan begint achter het programma precies een nieuwe geheugenpagina. Het nummer van die pagina is de high-byte van het adres, dus de inhoud van A. Is (B) ongelijk aan 0, dan moet de volgende (hele) pagina worden genomen om de tabel in op te slaan. Dit wordt gerealiseerd door (A) met 1 te verhogen. Het direct page register kan alleen gehele pagina's aanwijzen. Door tenslotte de inhoud van A in het direct page register op te bergen, bevat dit register het paginanummer van de eerste vrije geheugenpagina achter het programma. Dit alles is positie-onafhankelijk gebeurd.

Hierna kan de tabel op positie-onafhankelijke wijze worden geadresseerd met direct page addressing instructies als

STY <79

Met deze instructie wordt de inhoud van Y opgeborgen op de 80ste en 81ste locatie in de tabel. De tabel begint immers op locatie 0 van de betreffende pagina.

12.4.5 I/O-adressen

Soms zijn er in positie-onafhankelijke programma's toch absolute adressen nodig. Denk hierbij aan I/O-adressen.

Het is doorgaans verstandig deze adressen op te nemen in een I/O-tabel en deze in de global storage te zetten, zodat elk programma er gebruik van kan maken.

Wijzig de hardware-configuratie, dan hoeft alleen deze adrestabel te worden aangepast. Alle programma's functioneren verder onveranderd.

De I/O-adressen kunnen met behulp van een klein programma worden opgeborgen in de global storage.

Voorbeeld 12.8 (Het creëren van een I/O-tabel in de global storage)

In een bepaalde hardware-configuratie bevindt zich het read/write geheugen vanaf adres \$0000 t/m \$7FFF. De hoogste locaties van dit read/write geheugen worden voor de stack gebruikt. Het basisadres van de PIA is \$FF04, dat van de ACIA is \$FF08. Verder zijn 8 schakelaars te bereiken via adres \$FF00 en 8 LED's via \$FF01.

We nemen de routine om de I/O-tabel te creëren op in het monitorprogramma. Het monitorprogramma staat bij deze configuratie in de ROM vanaf adres \$F000 en wordt gestart bij een hardware reset. Hiertoe moeten de locaties \$FFFE en \$FFFF achtereenvolgens de waarden \$F0 en \$00 bevatten (zie ook paragraaf 9.7.8).

De I/O-tabel wordt in de global storage gezet, die zich vanaf de hoogste stacklocatie bevindt.

Het monitorprogramma begint nu als volgt:

10		ORG	\$F000	
20	STACK	EQU	\$8000	
30	BASSCH	EQU	\$FF00	basisadres van de schakelaars
40	BASLED	EQU	\$FF01	basisadres van de LED's
50	BASPIA	EQU	\$FF04	basisadres van de PIA
60	BASACI	EQU	\$FF08	basisadres van de ACIA
70	*			
80	*			
100	MONI	LDS	#STACK	initialiseer de stack pointer S
110		TFR	S,U	initialiseer de stack mark U
120		LDY	#BASSCH	zet de adressen van schakelaars
130		LDX	#BASLED	en LED's
140		PSHS	X,Y	op de stack
150		LDA	#4	
160		LDX	#BASPIA	zet de vier opeenvolgende
170	ZETWEG	PSHS	X	adressen van P/DDRA, P/DDRB,
180		LEAX	1,X	CRA en CRB op de stack
190		DECA		
200		BNE	ZETWEG	
210		LDY	#BASACI	zet de adressen van C/SR en
220		LEAX	1,Y	T/RDR op de stack
230		PSHS	X,Y	
240		---		
:		:		
:		:		
900		ORG	\$FFFE	
910		FDB	MONI	

Na het uitvoeren van het monitorprogramma t/m regel 230 heeft stack mark U de waarde \$8000 gekregen en stack pointer S de waarde \$7FF0. Er zijn 8 I/O-adressen, dus 16 bytes op de stack gezet, zodat de stack pointer S met 16 (= \$10) is verlaagd. Merk op dat bij regel 140 eerst de inhoud van Y op de stack wordt gezet (dus het adres van de schakelaars) en daarna de inhoud van X (het adres van de LED's). Evenzo wordt bij regel 230 eerst het adres van C/SR en daarna dat van T/RDR op de stack gezet. De assembler directives op de regels 900 en 910 zorgen ervoor dat het beginadres van het monitorprogramma (\$F000) wordt opgeborgen in de RESET-interruptvector (op \$FFFE en \$FFFF).

Wijzig de hardware-configuratie, dan hoeven alleen de assembler directives op de regels 10 t/m 60 te worden aangepast. Vervolgens moet het programma opnieuw worden geassembleerd en in een ROM geladen. Bij wijziging van het basisadres van de ROM, hoeft slechts regel 10 te worden aangepast. Verandert de plaats van de stack, dan moet regel 20 worden veranderd. Zijn de basisadressen van een van de I/O-devices gewijzigd, dan moet een van de regels 30 t/m 60 worden gecorrigeerd. De basisadressen van de I/O-devices hoeven niet opeenvolgend te zijn.

Alle gebruikersprogramma's blijven onveranderd. Vanuit deze programma's zijn de I/O-adressen te bereiken via de stack mark U. Dit is eenvoudig te realiseren door bij het assembleren van de gebruikersprogramma's telkens de volgende EQUATE-file mee te nemen:

- * EQUATE-file voor het via een symbolische offset bereiken
- * van de I/O-devices.
- *

```

SCHAK    EQU    -2
LEDS     EQU    -4
P/DDRA   EQU    -6
P/DDRB   EQU    -8
CRA      EQU    -10
CRB      EQU    -12
C/SR     EQU    -14
T/RDR    EQU    -16

```

De I/O-devices zijn in de gebruikersprogramma's dan rechtstreeks toegankelijk met behulp van instructies als

```

STA      [C/SR,U]    schrijf in controlregister van
                     de ACIA
LDA      [SCHAK,U]   lees de stand van de
                     schakelaars

```

12.4.6 Voorbeelden van positie-onafhankelijke programma's

Voorbeeld 12.9 (Zoek maximum)

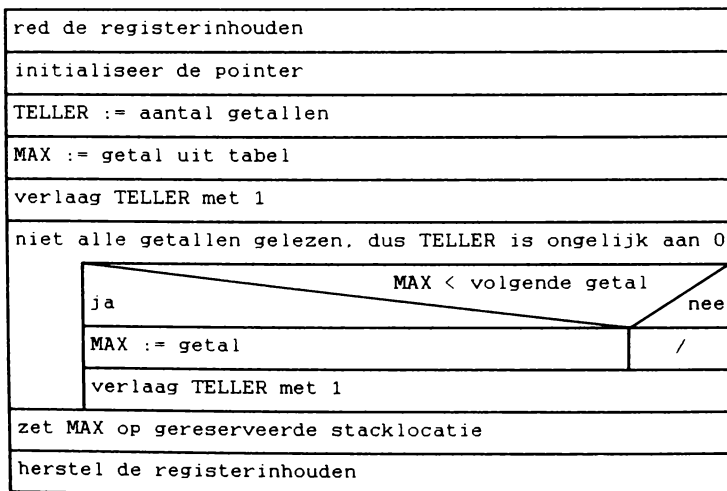


Fig. 12.6 Structogram bij het bepalen van het grootste getal uit een tabel

Ontwerp een positie-onafhankelijke subroutine die het maximale getal uit een tabel van 1-byte two's complement getallen opzoekt. Dit getal moet op de system stack worden opgeborgen. Hiertoe heeft het aanroepende programma een positie op de stack gereserveerd.

Er moet voor worden gezorgd, dat de registers voor en na het uitvoeren van de subroutine, dezelfde inhoud hebben.

De tabel, voorafgegaan door het aantal getallen dat de tabel bevat, wordt direct achter de subroutine in het geheugen geladen.

We gaan er van uit, dat de tabel tenminste één getal bevat.

Het structogram van fig. 12.6 geeft een mogelijke oplossing van het geschetste probleem.

De bijbehorende subroutine ziet er als volgt uit:

```

100 *****
110 * Subroutine MAXIM *
120 *****
130 *
140 *Progr.: J.G. Rouland
150 *Revisie 0      Datum: 24-12-1986
160 *
170 *De positie-onafhankelijke subroutine MAXIM zet het
180 *grootste two's complement getal op de stacklocatie, die
190 *door het aanroepende programma is gereserveerd, vlak voor
200 *de subroutine call.
210 *Het aantal te vergelijken getallen en de getallen zelf
220 *worden direct achter de subroutine in het geheugen
230 *geladen.
240 *
250          ORG      $5000
260 MAXIM    PSHS     CC,A,B,X      red registerinhouden
270          LEAX     AANTAL,PC      (pointer) := adres van AANTAL
280          LDA      AANTAL,PC      TELLER := aantal getallen van
285 *                               de tabel
290          LDB      A,X           MAX := laatste getal uit tabel
300          DECA     verlaag TELLER
310 TEST     BEQ      KLAAR        spring als alle getallen
315 *                               gelezen zijn
320          CMPB     A,X           vergelijk MAX met vorige getal
325 *                               in tabel
330          BGE      KLEIN        spring als MAX ≥ getal
340          LDB      A,X           MAX := getal
350 KLEIN    DECA     verlaag TELLER
360          BRA      TEST
370 KLAAR    STB      7,S           zet MAX op de gereserveerde
375 *                               stacklocatie
380          PULS     CC,A,B,X      herstel de registerinhouden
390          RTS
400 *
410 AANTAL   RMB      1            aantal getallen van de tabel

```

In regel 270 wordt X geladen met het adres van het aantal bytes, dat vlak voor de tabel getallen staat. Door dit aantal op te tellen bij het adres (regel 290) krijgen we het adres van het laatste getal in de tabel. Het eerste getal heeft als adres AANTAL+1, het laatste AANTAL+(totale aantal getallen van de tabel) = AANTAL+(AANTAL).

Door (A) te verlagen met 1, vinden we het adres van het op een na laatste getal (regel 300).

Door (A) telkens opnieuw te verlagen wijst (A)+(X) achtereenvolgens naar alle getallen in de tabel, van achteren naar voren (zie de regels 350 en 320).

Opgemerkt kan worden dat regel 350 eigenlijk kan vervallen door vanaf regel 360 terug te springen naar regel 300 in plaats van naar regel 310. Dit is nagelaten teneinde niet van het structogram af te wijken. Zou deze instructie wel weggelaten zijn, dan krijgen we een ongestructureerd programma. De zo verkregen oplossing kan niet worden weergegeven in een structogram.

In regel 370 is een hogere offset nodig dan 2, omdat niet alleen het terugkeeradres op de stack gezet is, maar omdat bovendien de inhoud van de registers CC, A, B en X op de stack gered worden. Het gaat hierbij om 5 locaties. Vandaar dat de offset 7 moet zijn. Dit is duidelijk te zien in fig. 12.7, die de opbouw van de stack laat zien.

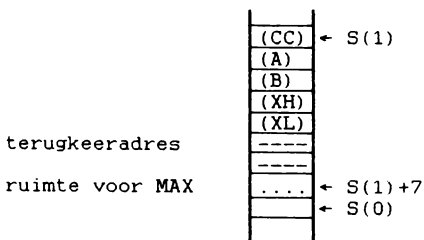


Fig. 12.7 Stackopbouw tijdens uitvoeren van subroutine MAXIM

Gaan we uit van de situatie waarbij de stack pointer de waarde S(0) heeft, dan wordt in het hoofdprogramma ruimte op de stack gecreëerd ten behoeven van MAX. Dit gebeurt met behulp van de instructie LEAS -1,S. Vervolgens wordt de subroutine MAXIM aangeroepen, waarbij het terugkeeradres op de stack wordt gezet en de stack pointer met 2 wordt verlaagd. Aan het begin van MAXIM worden de inhoud van de registers CC, A, B en X op de stack gezet, teneinde ervoor te zorgen dat de registerinhouden aan het einde van de subroutine weer hetzelfde zijn als bij het (tijdelijk) verlaten van het aanroepende programma. De stack pointer wordt hierbij met 5 verlaagd en krijgt de waarde S(1). Derhalve moet MAX (in regel 370) worden opgeborgen op de locatie met als adres S(1)+7.

Voorbeeld 12.10 (Positie-onafhankelijke sprong via jump table)
 Veronderstel dat een bepaald besturingssysteem in vier verschillende modi kan werken. De bijbehorende programma's zijn geschreven in de vorm van subroutines.
 In mode 0 vindt er een zelftest plaats, waarin het systeem zelf de werking van bepaalde functies test. In mode 1 wordt het systeem met de hand bediend, in mode 2 gebeurt dit op afstand en in mode 3 werkt het systeem helemaal automatisch.

Met behulp van twee schakelaars wordt ingesteld in welke mode het systeem moet functioneren. Het adres van de schakelaars SW0 en SW1 is bereikbaar via de offset SWITCH ten opzichte van de stack mark U.

Aan het begin van het programma wordt de jump table samengesteld. Vervolgens wordt de stand van de schakelaars binnengehaald en wordt er rechtstreeks naar de bijbehorende subroutine gesprongen, zonder dat de stand van de schakelaars hoeft te worden vergeleken met de vier mogelijke waarden.

100	CONTR	LEAX	JUMP,PC	(X) := beginadres jump tabel
110		LEAY	SELF,PC	berg startadres van zelftest
120		STY	,X	routine op in jump table
130		LEAY	MANU,PC	berg startadres van manual
140		STY	2,X	routine op in jump table
150		LEAY	REMO,PC	berg startadres van remote
160		STY	4,X	routine op in jump table
170		LEAY	AUTO,PC	berg startadres van automatic
180		STY	6,X	routine op in jump table
190		LDA	[SWITCH,U]	laad stand van schakelaars SW0
200		ANDA	#\$03	en SW1; vermenigvuldig deze
210		ASLA		met 2 t.b.v. offset in jump
215	*			table
220		JSR	[A,X]	voer de juiste subroutine uit
230		---		
240		---		
250		SWI		
260	*			
270	SELF	---		
280		---		
290		RTS		
300	*			
310	MANU	---		
320		---		
330		RTS		
340	*			
350	REMO	---		
360		---		
370		RTS		
380	*			
390	AUTO	---		
400		---		


```

410          RTS
420 *
430 JUMP     RMB      8          reserveer 8 locaties voor
435 *                               jump table
440 *
450          END      CONTR

```

Fig. 12.8 toont de jump table, zoals die in de regels 100 t/m 180 is gecreëerd.

SELFH	← JUMP
SELF L	
MANUH	← JUMP+2
MANUL	
REMOH	← JUMP+4
REMO L	
AUTOH	← JUMP+6
AUTO L	

Fig. 12.8 Jump table

Door de waarde van de schakelaars (0, 1, 2 of 3) te verdubbelen (regel 210) vinden we de offset (0, 2, 4 of 6), behorende bij het startadres van de gewenste subroutine. Springen naar die subroutine kan nu eenvoudig geschieden door een accumulator offset indexed indirecte JSR-instructie (regel 220).

Het werken met een jump table maakt het programma overzichtelijk. Er hoeven geen vergelijkingen te worden gedaan om de juiste sprong te bepalen. Vandaar dat jump tables vooral handig werken als er veel mogelijkheden zijn.

Voorbeeld 12.11 (Calculated jump)

Wordt er geen jump table gebruikt, dan geschiedt het springen via een zogenaamde calculated jump. Hierbij wordt met behulp van een aantal vergelijkingen bepaald naar welk adres gesprongen moet worden. Fig. 12.9 laat zien op welke wijze een calculated jump wordt uitgevoerd in het geval van voorbeeld 12.10.

lees de stand van de schakelaars SW0 en SW1			
SW1=0			
ja			nee
SW0=0		SW0=1	
ja	nee	ja	nee
selftest	manual	remote	automatic

Fig. 12.9 Calculated jump

U ziet dat er drie vergelijkingen moeten worden gedaan bij het testen van twee bits. Voor drie bits worden dit zeven vergelijkingen, voor vier bits vijftien, enzovoorts. Hierbij horen nog een vrij groot aantal spronginstructies. In dit voorbeeld ziet dit er als volgt uit:

100	CONTR	LDA	[SWITCH,U]	lees stand van schakelaars
110		BITA	#2	test SW1
120		BNE	SW1EEN	spring als SW1=1
130		BITA	#1	test SW0
140		BNE	MANUAL	spring als SW0=1
150		BSR	SELF	mode 0
160		BRA	VERDER	
170	MANUAL	BSR	MANU	mode 1
180		BRA	VERDER	
190	SW1EEN	BITA	#1	test SW0
200		BNE	AUTOMA	spring als SW0=1
210		BSR	REMO	mode 2
220		BRA	VERDER	
230	AUTOMA	BSR	AUTO	mode 3
240	VERDER	---		
250		---		

U ziet dat het volledig decoderen van twee schakelaars al tamelijk onoverzichtelijk is.

12.5 Reentrant routines

Reentrant routines zijn delen van programma's (zoals modules, subroutines of macro's) die correct blijven functioneren als ze opnieuw aangeroepen (binnengegaan) worden, voordat ze (ten gevolge van een vorige aanroep) helemaal uitgevoerd zijn.

Een eenvoudig voorbeeld van een routine die reentrant moet zijn, is een subroutine die zowel vanuit gewone programma's als vanuit interrupt service routines aangeroepen kan worden. Stel dat deze routine wordt aangeroepen vanuit een programma. Tijdens het uitvoeren van de subroutine treedt een interrupt op. Vanuit de bijbehorende interrupt service routine wordt deze subroutine eveneens aangeroepen. Dit betekent dat de subroutine opnieuw wordt uitgevoerd vanaf het begin, voordat hij ten gevolge van een vorige aanroep helemaal is uitgevoerd.

Ook routines die voor algemeen gebruik ontworpen zijn, moeten reentrant zijn, indien ze in een multitasking systeem worden gebruikt. Hier lopen een aantal taken tegelijk, zodat zo'n global routine door meerdere programma's "tegelijk" kan worden gebruikt, bijvoorbeeld in time sharing, waarbij elke taak beurtelings gedurende een bepaalde tijd wordt uitgevoerd. De beginadressen van global routines kunnen in een jump table

zijn opgenomen, bijvoorbeeld als globals op de stack. In dit geval worden ze vanuit de diverse programma's aangeroepen met behulp van indirect addressing via de stack mark:

JSR [ROUT,U]

Hierin is ROUT de offset (gerelateerd aan de stack mark U) van het beginadres van de gewenste subroutine

Een routine is reentrant als hij op een willekeurig moment kan worden onderbroken en vervolgens kan worden aangeroepen vanuit een ander programma, zonder verlies van gegevens voor het eerste programma.

De onderbreking van de routine wordt veroorzaakt door een interrupt. Het andere programma dat de onderbroken routine aanroept, kan bijvoorbeeld een interrupt service routine zijn.

In het algemeen is een routine niet reentrant. Er dienen speciale maatregelen getroffen te worden om ervoor te zorgen, dat de routine correct wordt uitgevoerd voor beide aanroepen.

Voorbeeld 12.12 (Niet-reentrant subroutine ADDBA)

Subroutine ADDAB voor het optellen van de inhoud van A en B. ADDAB wordt zowel vanuit het hoofdprogramma HOOFD als vanuit een interrupt service routine ISR aangeroepen en moet derhalve reentrant zijn.

De som (D) wordt op de stack gezet. Hiervoor heeft het aanroepende programma twee bytes ruimte gecreëerd.

Eerst laten we een oplossing zien, waarbij ADDAB niet reentrant is.

```

100 HOOFD    ---
110          ---
120          LDA    #25
130          LDB    #43                                t0
140          LEAS   -2,S          creëer ruimte voor de som
150          BSR    ADDAB                                t1
160          MAIN   PULS          D          haal de som op t7
170          ---
180          SWI
190 *
200 ISR      ---                                t3
210          ---
220          LDA    #76
230          LDB    #11
240          LEAS   -2,S          creëer ruimte voor de som
250          BSR    ADDAB                                t4
260          INTR   PULS          D          haal de som op
270          ---
280          RTI                                t6
290 *
300 * Niet-reentrant versie van ADDAB:
310 *
320 ADDAB     STA    HULP          red (A)
330           CLRA

```

340	SUBR	ADDB	HULP	
350		ADCA	#0	zet de carry in A
360		STD	2.S	zet som op gereserveerde
365	*			stacklocatie
370		RTS		
380	*			
390		END	HOOFD	

Veronderstel dat ADDAB wordt aangeroepen vanuit het hoofdprogramma HOOFD. Dan wordt (HULP)=25 (regel 320). Treedt tijdens het uitvoeren van de instructie CLRA (regel 330) een interrupt op, dan worden na deze instructie de inhouden van de registers op de stack gezet. Vervolgens wordt de interrupt service routine gestart. Deze roept ADDAB aan (regel 250), zodat nu (in regel 320) opnieuw HULP wordt gevuld met de inhoud van A. Deze is nu gelijk aan 76, zodat de oude waarde (25) vernield wordt. De subroutine wordt verder uitgevoerd en het resultaat van de optelling ($11+76 = 87$) wordt via de stack overgedragen aan het aanroepende programma (de interrupt service routine).

Aan het einde van de interrupt service routine worden de registerinhouden weer hersteld en wordt er teruggekeerd naar het onderbroken programma (ADDAB, regel 340). Er geldt nu (A)=0 en (B)=43, hetgeen correct is. In regel 340 wordt hierbij opgeteld (HULP). Dit is echter tijdens het uitvoeren van de interrupt service routine gewijzigd in 76, zodat als som wordt uitgerekend $43+76 = 119$ in plaats van $43+25 = 68$. Hierdoor loopt het programma verkeerd af. De interrupt service routine heeft data van het hoofdprogramma vernield. De subroutine ADDAB is niet reentrant.

Voorwaarde voor reentrancy

Een routine is alleen reentrant, indien aan de volgende voorwaarde is voldaan:

Een reentrant routine mag geen eigen werkgeheugen bezitten. Het werkgeheugen moet dus uitsluitend zijn toegewezen aan het aanroepende programma. Anders gezegd, het werkgeheugen moet zich bevinden in een gebied, dat uitsluitend wordt gebruikt ten behoeve van het aanroepende programma.

Dit betekent dat dit werkgeheugen slechts mag bestaan uit:

1. microprocessorregisters
2. vaste geheugenlocaties, mits deze alleen ten behoeve van het aanroepende programma worden gebruikt
3. stacklocaties

Registerinhouden worden bij interrupts automatisch op de stack opgeborgen en aan het einde van de interrupt service routine vanaf de stack weer teruggeschreven in de registers. Hierdoor worden de microprocessorregisters in feite automatisch toegewezen aan het aanroepende programma. De informatie die zij bevatten, wordt alleen binnen het aanroepende programma gebruikt en gewijzigd.

Het aanroepende programma kan ook expliciet een geheugenbuffer als werkgeheugen beschikbaar stellen. Het beginadres van deze buffer moet dan aan de reentrant routine worden meegedeeld via parameteroverdracht (en wel via een register of via de stack). Dit impliceert dat elk programma dat de reentrant routine aanroept, een andere pointerwaarde aan de reentrant routine moet doorgeven. Er moet door elk aanroepend programma een eigen vrije geheugenruimte als werkgeheugen worden gedefinieerd. Een waarschuwing is hier op zijn plaats. Als het aanroepende programma zelf reentrant is, treedt een probleem op. Denk bijvoorbeeld aan een IRQ-interrupt service routine die door een IRQ-interrupt kan worden onderbroken en die derhalve reentrant moet zijn. Dit is het geval, indien binnen de IRQ-interrupt service routine de I-bit wordt gereset. Nu moet hetzelfde aanroepende programma telkens een andere pointerwaarde doorgeven aan de reentrant routine. Het aanroepende programma moet hiertoe bijhouden hoeveel keren het achtereenvolgens de reentrant routine heeft aangeroepen. Deze noodzaak vervalt als de stack als werkgeheugen wordt gebruikt. Nu wordt automatisch bij elke aanroep vrije geheugenruimte gereserveerd.

Het toepassen van de stack als werkgeheugen is een tamelijk eenvoudige manier om de reentrant routine positie-onafhankelijk te maken.

Een aparte pointer overdragen is hierbij niet nodig, omdat de stack pointer het werkgeheugen aanwijst.

Wordt de stack als werkgeheugen toegepast, dan kan de reentrant routine zelf (aan het begin) hiervoor ruimte op de stack creëren. Ook dit geheugen komt alleen aan het aanroepende programma toe. Telkens als de routine opnieuw wordt aangeroepen, wordt er immers opnieuw een werkgeheugen gecreëerd. Het werkgeheugen wordt opgeheven, zodra het niet meer nodig is (bijvoorbeeld aan het einde van de routine).

Voorbeeld 12.13 (Reentrant subroutine ADDBA)

De volgende versie van routine ADDAB is reentrant.

```

300 * Reentrant versie van ADDAB:
310 *
320 ADDAB    PSHS    A                red (A)                t2, t5
330          CLRA
340 SUBR     ADDB    ,S+
350          ADCA    #0              zet de carry in A
360          STD     2,S              zet de som op gereserveerde
365 *                               stacklocatie
370          RTS

```

Aan de hand van de opbouw van de stack op de verschillende tijdstippen (zie fig. 12.10) kunt u zien dat de laatste routine inderdaad reentrant is. Er gaan geen gegevens voor het hoofdprogramma verloren.

De hieronder vermelde regelnummers hebben betrekking op de programma's HOOFD, ISR en de reentrant versie van de routine ADDAB.

Op $t=t_1$ heeft HOOFD ruimte op de stack gecreëerd ten behoeve van de overdracht van de som (msom vanaf ADDAB naar HOOFD).

Tevens is het terugkeeradres MAIN op de stack opgeborgen.

Op $t=t_2$ is de inhoud van A (= 25) op de stack gezet.

Op $t=t_3$ zijn (ten gevolge van de interrupt) de registerinhouden op de stack gered.

Op $t=t_4$ heeft ISR ruimte op de stack gecreëerd voor de som (isom) en is het terugkeeradres INTR op de stack gezet.

Op $t=t_5$ wordt nogmaals de inhoud van A op de stack opgeborgen. Nu is (A)=76.

Daarna wordt in regel 340 de inhoud van de bovenste stacklocatie opgeteld bij (B), waarbij de stack pointer met 1 wordt verhoogd. In regel 360 wordt de som ($11+76 = 87$) op de stack opgeborgen (isom) en in regel 370 wordt het terugkeeradres van de stack gehaald. Tenslotte wordt in regel 260 isom van de stack gehaald.

Als $t=t_6$ is de interrupt afgehandeld en worden de registerinhouden hersteld, waarbij tevens wordt teruggesprongen naar het onderbroken programma. Dit is de reentrant routine ADDAB, die wordt voortgezet met de instructie op het adres SUBR. Hier wordt (B) verhoogd met de bovenste locatie van de stack, het getal 25. De som ($43+25 = 68$) wordt in regel 360 op de stack gezet (msom). In regel 160 wordt deze som van de stack gehaald, waarbij de stack pointer weer op de beginpositie terugkeert ($t=t_7$).

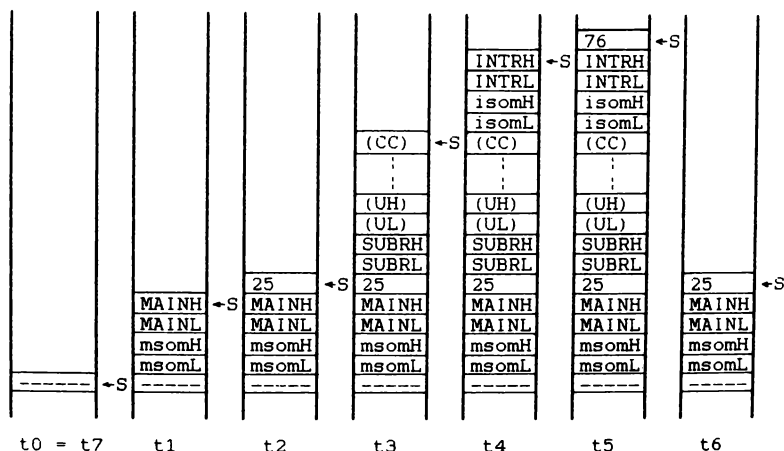


Fig. 12.10 Stackopbouw op diverse tijdstippen bij de reentrant routine ADDAB

Onthoud dat een routine reentrant moet zijn, indien deze routine door van elkaar onafhankelijke en onderbreekbare programma's kan worden aangeroepen!

12.6 Recursieve routines

Een recursieve routine is een routine die zichzelf aanroept. Een recursieve routine moet derhalve zeker reentrant zijn. Dit echter niet voldoende!

Omdat de routine zichzelf aanroept, dus zonder eerst onderbroken te worden door een interrupt, worden de registerinhouden niet automatisch op de stack gered. Dit moet dus altijd aan het begin van de recursieve routine zelf gebeuren.

Voorwaarden voor recursie

Samenvattend gelden de volgende voorwaarden, waaraan achtereenvolgens moet zijn voldaan, opdat een routine recursief is:

1. Red de registerinhouden aan het begin van de routine in een vrije geheugenruimte.
2. Creëer aan het begin van de routine een werkgeheugen in een vrije geheugenruimte.

Voor het werkgeheugen geldt hetzelfde als is opgemerkt bij reentrant routines.

Indien het werkgeheugen niet te groot is, kan worden volstaan met het gebruik van de registers van de microprocessor.

Bij een grotere werkruimte is het gebruik van de stack aan te bevelen, omdat hierbij op eenvoudige wijze bereikt wordt, dat de recursieve routine tevens positie-onafhankelijk is.

Recursie komt vrij veel voor in de wiskunde en in de informatica.

Een recursieve functie is een functie die in termen van zichzelf is gedefinieerd.

Een klassiek voorbeeld van een functie die als een recursieve functie kan worden beschreven, is de faculteit.

De faculteit van een natuurlijk getal N (N is positief en geheel) is gedefinieerd als

$$N! = N \cdot (N-1) \cdot (N-2) \cdot (N-3) \cdot \dots \cdot 3 \cdot 2 \cdot 1$$

waarbij moet gelden $N > 1$

terwijl verder is gedefinieerd dat $0! = 1! = 1$.

De recursieve notatie van deze functie is als volgt:

$$N! = 1 \quad \text{voor } N = 0 \text{ en}$$

$$N! = N \cdot (N-1)! \quad \text{voor } N > 0$$

De recursieve schrijfwijze is erg compact en erg

overzichtelijk. Een recursieve routine is eveneens erg eenvoudig van opbouw. Voor beginners kan een recursieve routine echter tamelijk verwarrend lijken.

Een recursieve routine wordt doorgaans meerdere keren volledig uitgevoerd bij elke externe aanroep.

Voorbeeld 12.14 (Recursieve subroutine voor het omkeren van een karakterstring)

Een rij karakters wordt ontvangen via een ACIA en moet in omgekeerde volgorde via de ACIA worden verzonden. Het einde van de karakterstring wordt aangegeven met het karakter ETX, End of Tekst (\$03).

Fig. 12.11 geeft het structogram van de recursieve subroutine KEEROM.

Op het eerste gezicht lijkt zo'n recursieve routine niet bijzonder. Echter u ziet dat het eigenaardige erin bestaat, dat de subroutine zichzelf aanroept.

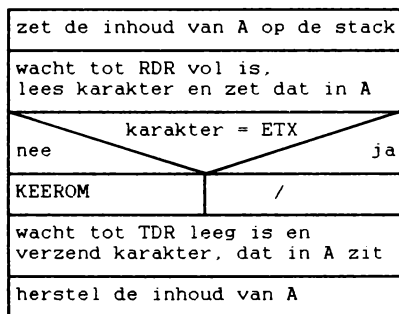


Fig. 12.11 Structogram van de recursieve subroutine KEEROM

Eerst worden de registerinhouden, voor zover deze niet verloren mogen gaan, op de stack opgeborgen. In dit voorbeeld is dat alleen de inhoud van A. Hierin wordt telkens het gelezen karakter geplaatst. Dit mag natuurlijk niet verloren gaan. Vervolgens wordt het eerste karakter gelezen en in A gezet. Is dit karakter ongelijk aan ETX, dan wordt de routine KEEROM opnieuw aangeroepen, dit keer vanuit zichzelf. De routine is nog niet afgehandeld, immers het gelezen karakter is nog niet verzonden. Er wordt teruggesprongen naar het begin van de routine, waar de inhoud van A, dus het gelezen karakter, op de stack wordt opgeborgen. Hierna wordt er een tweede karakter binnengehaald. Is ook dit tweede karakter ongelijk aan ETX, dan wordt opnieuw de subroutine KEEROM onderbroken en wordt deze weer van voren af aan uitgevoerd, enzovoorts.

Wordt uiteindelijk het karakter ETX binnengehaald, dan wordt de routine, waarin dit is gebeurd (de laatst aangeroepen routine), voortgezet. Het karakter dat tijdens de uitvoering van deze routine werd gelezen, wordt verzonden. Dit betreft dus het karakter ETX. Hiermee wordt de subroutine die het laatst is aangeroepen, het eerste afgemaakt. Aan het einde van die routine wordt de inhoud van A weer hersteld. Dit betekent dat het voorlaatste karakter in A wordt teruggezet (vanaf de stack).

Vervolgens wordt de vorige routine KEEROM afgemaakt door het verzenden van het karakter dat in A staat. Dit is het voorlaatste karakter, dus het karakter dat tijdens het uitvoeren van die routine werd binnengehaald. Aan het einde van die routine wordt wederom A vanaf de stack gevuld, nu met het op twee na laatst ontvangen karakter.

Dit gaat zo door, totdat ook de routine die het eerste, dus door het hoofdprogramma, werd aangeroepen, helemaal is afgemaakt. Dan is tenslotte ook het eerste karakter verzonden. Op deze wijze worden de ontvangen karakters in omgekeerde volgorde verzonden.

Elke routine wordt als het ware van binnen uit voltooid. Het gebruik van de stack als opslagmedium is hierbij uitstekend geschikt. In de stack wordt immers het laatst opgeborgen karakter het eerst teruggehaald. Dit is juist de volgorde die bij een recursieve routine nodig is. Een recursieve routine roept zichzelf zolang aan, totdat hij kan worden beëindigd. Als de laatst aangeroepen routine beëindigd is, wordt de voorlaatste afgemaakt, enzovoorts, totdat uiteindelijk ook de eerst aangeroepen routine voltooid is. Hiermee is de door het hoofdprogramma aangeroepen routine volledig uitgevoerd.

Door de stack als werkgeheugen te gebruiken, wordt bovendien bereikt dat de recursieve routine ook door een andere programma (met behulp van een interrupt) kan worden onderbroken, zonder dat er gegevens van het eerste programma verloren gaan.

De recursieve subroutine KEEROM (die tevens positie-onafhankelijk is) ziet er als volgt uit:

```
100 *****
110 * KEEROM *
120 *****
130 *
140 *Progr.: J.G. Rouland
150 *Revisie 0      Datum: 27-12-1986
160 *
170 *De recursieve subroutine KEEROM ontvangt een aantal
180 *karakters via een ACIA en zendt deze in omgekeerde
190 *volgorde via dezelfde ACIA weg. De adressen van de ACIA-
200 *registers moeten in een tabel staan, die wordt aangewezen
210 *door de stack mark U.
```

```

220 *Er wordt verondersteld dat er geen fouten optreden bij de
230 *transmissie.
240 *De reeks ontvangen karakters moet zijn afgesloten met ETX.
250 *
260 *KEEROM verandert de inhoud van de registers CC en B.
270 *
280 ETX      EQU      $03
290 *
300 KEEROM   PSHS      A
310 FULL     LDB        [SR,U]      wacht tot het Receive Data
320          BITB       #1          Register van de ACIA
330          BEQ        FULL        vol is
340          LDA        [RDR,U]     lees de ontvangen data
350          CMPA       #ETX
360          BEQ        EMPTY       spring als gelezen karakter=ETX
370          BSR        KEEROM      ga naar subroutine KEEROM
380 EMPTY    LDB        [SR,U]      wacht tot het Transmit Data
390          BITB       #2          Register van de ACIA
400          BEQ        EMPTY       leeg is
410          STA        [TDR,U]     verzend het gelezen karakter
420          PULS       A
430          RTS

```

Vanuit een programma kan deze recursieve routine eenvoudigweg als volgt worden aangeroepen:

```

        BSR      KEEROM
TERUG   ---

```

Bekijkt u de opbouw van de stack als functie van de tijd (zie fig. 12.12), dan ziet u dat de routine op de juiste wijze wordt uitgevoerd. Als voorbeeld is hier genomen, dat achtereenvolgens de karakters k, l en ETX worden ontvangen.

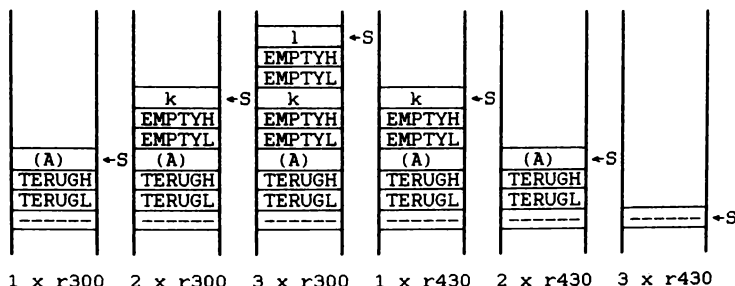


Fig. 12.12 Opbouw van de stack bij het ontvangen en verzenden van de karakters k, l en ETX met de recursieve routine KEEROM

Wordt vanuit een programma de subroutine KEEROM aangeroepen, dan wordt hierbij automatisch het terugkeeradres TERUG op de stack gezet.

De eerste keer dat de push-instructie op regel 300 wordt uitgevoerd, wordt de inhoud van A op de stack opgeborgen. Vervolgens wordt het eerste karakter, k, binnengehaald via het Receive Data Register. Daar dit ongelijk is aan ETX, wordt opnieuw de subroutine KEEROM aangeroepen (regel 370) en wordt het terugkeeradres EMPTY op de stack gezet.

De volgende instructie, die wordt uitgevoerd, is de push-instructie op regel 300. De inhoud van A, nu het karakter k, wordt op de stack gezet. Regel 300 is nu twee keer uitgevoerd. Hierna wordt weer een karakter gelezen. Dit is deze keer het karakter l. Aangezien nog steeds geen ETX is ontvangen, wordt er (in regel 370) wederom naar het begin van de subroutine KEEROM gesprongen en wordt het terugkeeradres EMPTY op de stack bewaard.

Voor de derde keer wordt de instructie PSHS A (regel 300) uitgevoerd, zodat nu het karakter l op de stack wordt gezet. Hierna wordt ETX binnengehaald, zodat er niet meer opnieuw wordt begonnen met het uitvoeren van KEEROM. De subroutine KEEROM wordt deze keer voortgezet met regel 380, zodat het karakter ETX via het Transmit Data Register wordt verzonden. Daarna wordt in regel 420 voor het eerst de instructie PULS A uitgevoerd, zodat A wordt geladen met l, het voorlaatste ontvangen karakter. Bij de RTS-instructie (regel 430) wordt het terugkeeradres EMPTY van de stack gehaald, zodat opnieuw de inhoud van A, nu het karakter l wordt verzonden.

Voor de tweede keer wordt nu de pull-instructie van regel 420 uitgevoerd, waardoor A wordt geladen met het karakter k. Dit was als eerste karakter ontvangen en wordt nu via TDR verzonden.

Nu wordt de instructie PULS A (regel 420) voor de derde maal uitgevoerd. De inhoud die A in het aanroepende programma had, wordt nu hersteld. Vervolgens wordt in regel 430 bij de RTS-instructie het terugkeeradres TERUG van de stack gehaald, zodat er wordt teruggekeerd naar het aanroepende programma. De recursieve subroutine KEEROM is nu volledig uitgevoerd (in dit voorbeeld drie keer).

Hierbij is de inhoud van accumulator A niet blijvend veranderd.

Door aan het begin van de recursieve subroutine KEEROM ook de inhouden van de registers B en CC op de stack te zetten, blijven ook deze uiteindelijk onaangetast.

Dit kost echter in totaal $3 \cdot 2 = 6$ stacklocaties, omdat KEEROM hier drie keer wordt uitgevoerd. Het is duidelijk dat dit niet erg zinvol is, omdat de inhoud van B en CC voor wat betreft KEEROM niet bewaard hoeft te blijven.

Recursieve routines zijn niet noodzakelijk. Elk probleem kan met behulp van iteratie (herhaling) worden opgelost. Bij KEEROM kunnen de karakters in een buffer worden gezet om ze daarna in omgekeerde volgorde uit te lezen en te verzenden. Hierbij moeten twee herhalingslussen worden doorlopen, een voor het binnenhalen van de gegevens en een voor het verzenden.

Bij de recursieve oplossing wordt slechts één lus doorlopen.
 Recursie is echter compact.

Om niet onnodig veel stackruimte te gebruiken, is het aan te bevelen de registerinhouden die voor het aanroepende programma niet verloren mogen gaan, in het aanroepende programma zelf te redden.

Dit kan dan als volgt geschieden:

	PSHS	B,CC
	BSR	KEEROM
TERUG	PULS	B,CC

Register A wordt al in KEEROM zelf gered!

Voorbeeld 12.15 (Recursieve subroutine van de berekening van N-faculteit)

Het getal N wordt ingevoerd via de A-poort van een PIA. De adressen van de PIA staan als globals op de stack. Register U functioneert als stack mark.

Het geldig zijn van N op PA kan bijvoorbeeld met een interrupt aan het hoofdprogramma worden meegedeeld. Hierop wordt de subroutine FACULT aangeroepen. Deze routine verzorgt een aantal acties ten behoeve van de recursieve subroutine FAC, die het eigenlijke rekenwerk uitvoert.

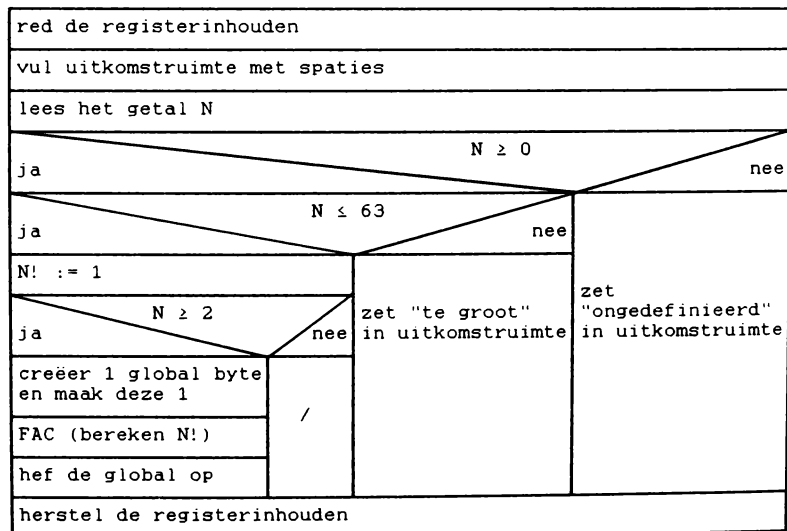


Fig. 12.13 Structogram van de subroutine FACULT

De acties die worden uitgevoerd door FACULT, zijn vermeld in het structogram van fig. 12.13.

FACULT ziet er als volgt uit:

```

5 *****
10 * Subroutine FACULT *
15 *****
20 *
25 *Progr.: J.G. Rouland
30 *Revisie 0      Datum: 29-12-1986
35 *
40 *FACULT leest een getal N (minimaal 0 en maximaal 63) van
45 *PRA van een PIA. De PIA-adressen staan in de global
50 *storage, aangewezen door stack mark U.
55 *In de uitkomstruimte $00 t/m $2F van de geheugenpagina,
60 *aangewezen door DP, komt een foutmelding in ASCII te staan
65 *of de hexadecimale waarde van N-faculteit (NFACUL).
70 *De minstwaardige byte komt op $00 te staan.
75 *
80 *FACULT is positie-onafhankelijk en verandert de
85 *registerinhouden niet.
90 *
95 *
100 ALLREG EQU    $FF
105 MAX     EQU    63                maximaal toegestane waarde van N
110 *
115 FACULT   PSHS   #ALLREG          red alle registers
120         BSR    BLANK            vul uitkomstruimte met spaties
125 *
130         LDA    [PRA,U]          lees N
135         BMI    UNDEF            spring als N<0
140         CMPA   #MAX
145         BHI    TOOBIG           spring als N>63
150         BSR    FACEEN           NFACUL := 1
155         CMPA   #2
160         BLO    EINDE            spring als N=0 of als N=1
165         TFR    S,U              definieer nieuwe stack mark
170         LDA    #1               maak LOD=1 en zet deze als
175         PSHS   A                global op de stack
180         BSR    FAC              bereken NFACUL
185 BACK     LEAS   1,S             hef global byte op
190         BRA    EINDE
195 TOOBIG   BSR    BIGFAC          geef melding "te groot"
200         BRA    EINDE
205 UNDEF    BSR    NOTDEF          geef melding "ongedefinieerd"
210 EINDE    PULS   #ALLREG        herstel alle registerinhouden
215         RTS
230 *
235 *
240 *****
245 * Subroutine BLANK *
250 *****
255 *
```

```

260 *BLANK vult de uitkomstruimte met spaties.
265 *X moet wijzen naar adres 0 van de uitkomst.
270 *
275 *Aan het einde van de subroutine bevat X het adres van de
280 *eerste locatie ($00) van de uitkomstruimte.
285 *Verder veranderen de inhouden van A, B en CC.
290 *
295 *
300 SPACE EQU $20
305 HIBYTE EQU $2F          hoogste adres van uitkomstruimte
310 *
315 BLANK CLRB              laad X met het adres van
320 TFR DP,A                locatie 0
325 TFR D,X                 van de uitkomstruimte
330 LDA #HIBYTE            A wijst naar de hoogste byte
335 LDB #SPACE
340 NEXT STB A,X            vul uitkomstruimte met spaties.
345 DECA                   te beginnen vanaf locatie $2F
350 BPL NEXT
355 RTS
370 *
375 *
380 *****
385 * Subroutine BIGFAC *
390 *****
395 *
400 *BIGFAC zet de melding "N>63. faculteit wordt te groot" in
405 *de uitkomstruimte.
410 *X moet wijzen naar adres 0 van de uitkomst.
415 *
420 *De registers A, B, CC en Y veranderen van inhoud.
425 *
430 *
435 GROOT FCC /N>63. faculteit wordt te groot/
440 *
445 BIGFAC LEAY GROOT,PC     Y wijst naar de melding
450 LDA #29
455 GADOOR LDB A,Y           zet de 30 karakters van de
460 STB A,X                  melding in de uitkomstruimte
465 DECA
470 BPL GADOOR
475 RTS
490 *
495 *
500 *****
505 * Subroutine NOTDEF *
510 *****
515 *
520 *NOTDEF zet de melding "N<0. faculteit is ongedefinieerd" in
525 *de uitkomstruimte.
530 *X moet wijzen naar adres 0 van de uitkomst.
535 *

```

```

540 *De registers A, B, CC en Y veranderen van inhoud.
545 *
550 *
555 KLEIN   FCC      /N<0. faculteit is ongedefinieerd/
560 *
565 BIGFAC  LEAY     KLEIN.PC      Y wijst naar de melding
570                               LDA      #31
575 VERDER  LDB      A,Y           zet de 32 karakters van de
580                               STB      A,X           melding in de uitkomstruimte
585                               DECA
590                               BPL      VERDER
595                               RTS
610 *
615 *
620 *****
625 * Subroutine FACEEN *
630 *****
635 *
640 *FACEEN maakt NFACUL gelijk aan 1.
645 *X moet wijzen naar adres 0 van de uitkomst.
650 *
655 *De registers B en CC veranderen van inhoud.
670 *
675 *
680 FACEEN  LDB      #HIBYTE       B wijst naar hoogste NFACUL-byte
685 NULLEN  CLR      B,X          NFACUL := 0
690                               DECB
695                               BPL      NULLEN
700                               INC      .X           maak laagste uitkomstbyte 1
705                               RTS

```

Merk op dat de stack mark van het hoogste niveau weer wordt hersteld bij de pull-instructie van regel 210. Hierbij worden de inhouden van alle registers hersteld, dus ook die van de stack mark U.

De uitkomstruimte omvat 48 locaties. Hierin past de grootste uitkomst, zijnde $63! = 1.982608 \cdot 10^{47}$.

Hiervoor zijn in feite slechts 37 bytes nodig, immers

$2^{24} \cdot 3^6 = 5,0 \cdot 10^{14}$ en $2^{24} \cdot 3^7 = 1,3 \cdot 10^{15}$.

Er is een veelvoud van 16 locaties gekozen, omdat in het LABBUS-ontwikkelingssysteem (bij een memory dump) 16 geheugenlocaties per regel worden getoond op het video display.

In regel 170 en 175 wordt het getal 1 als global byte op de stack gezet. Deze global wordt gebruikt als variabele LOLD. Deze geeft het aantal bytes aan, dat het laatst berekende produkt NFACUL bevat. De beginwaarde hiervan moet 1 zijn, immers in de subroutine FACEEN (regel 150) is NFACUL gelijk aan 1 gemaakt. Hiervoor is slechts 1 byte nodig.

In regel 165 is de nieuwe stack mark U, de zogenaamde lower level stack mark, gedefinieerd, teneinde LOLD op elk niveau

binnen de subroutine FACULT te kunnen bereiken. Dit moet geschieden met de offset -1 (zie ook regel 875). Bedenk dat de oorspronkelijke stack mark U, de highest level stack mark, naar de global storage wijst, waar onder andere de I/O-adressen zijn opgeslagen.

Fig. 12.14 laat het structogram zien van de recursieve subroutine FAC.

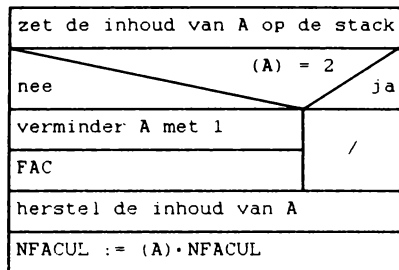


Fig. 12.14 Structogram van de recursieve subroutine FAC

De inhoud van A (= N) wordt op de stack opgeborgen. Als N gelijk is aan 2, wordt (A) weer van de stack gehaald en wordt de uitkomst N! berekent uit $2 \cdot \text{NFACUL} = 2 \cdot 1! = 2$. Hierna wordt er teruggekeerd naar het aanroepende programma FACULT. De berekening is voltooid.

Is (A) echter groter dan 2, dan wordt (A) verlaagd en wordt opnieuw de subroutine FAC aangeroepen. De berekening kan immers nog niet worden uitgevoerd, omdat $(N-1)!$ nog niet bekend is. Aan het begin van de subroutine wordt (A), dit is nu N-1, op de stack gezet.

Is (A) wederom groter dan 2, dan wordt (A) weer verlaagd. FAC wordt opnieuw onderbroken en vanaf het begin gestart.

Dit gaat zo door totdat (A) = 2. Nu wordt A geladen vanaf de stack en wordt NFACUL berekend uit $2 \cdot \text{NFACUL} = 2 \cdot 1! = 2$. Hiermee de laatst aangeroepen subroutine FAC is voltooid.

Nu wordt de voorlaatst aangeroepen subroutine FAC afgemaakt met het van de stack halen van de vorige waarde van A (= 3), waarmee wordt berekend $\text{NFACUL} = 3 \cdot 2 = 6$.

Uiteindelijk zal A geladen worden met de oorspronkelijke waarde N, waarna de einduitkomst $N \cdot \text{NFACUL} = N \cdot (N-1)!$ kan worden berekend. Nu is de subroutine FAC helemaal uitgevoerd en wordt er teruggekeerd naar het aanroepende programma FACULT.

De recursieve subroutine FAC ziet er als volgt uit:

```

720 *****
725 * Subroutine FAC *
730 *****
735 *
```



```

740 *FAC is de recursieve subroutine voor het bepalen van N-
745 *faculteit. N moet groter zijn dan 1 en kleiner dan 64.
750 *
755 *De registers A, B, CC, X en Y veranderen van inhoud.
760 *
765 *
770 FAC      PSHS    A
775          CMPA   #2
780          BEQ    REKEN      spring als (A) = 2
785          DECA
790          BSR    FAC
795 REKEN     PULS    A
800          BSR    MULTIP     NFACUL := (A)·NFACUL
805          RTS

```

Aan de hand van de stackopbouw op de diverse tijdstippen (fig. 12.15) kunt u zien dat de recursieve subroutine correct functioneert.

Hierbij is uitgegaan van de berekening van 4!

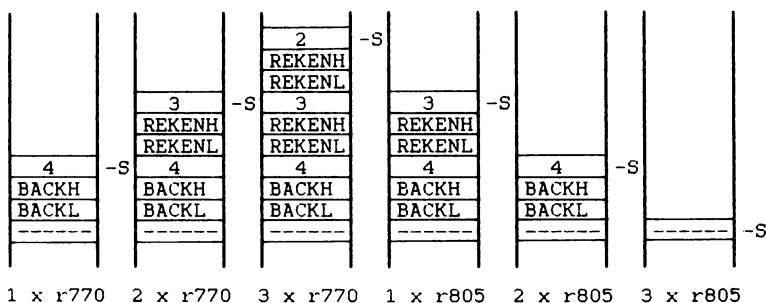


Fig. 12.15 Stackopbouw bij het berekenen van 4! met behulp van de recursieve subroutine FAC

Wordt FAC voor de eerste keer aangeroepen (dus vanuit FACULT), dan wordt het terugkeeradres BACK op de stack opgeborgen. Vervolgens wordt de inhoud van A (= 4) op de stack gezet (regel 770). Daar (A) > 2 wordt (A) verlaagd, waarna opnieuw FAC wordt aangeroepen, ditmaal vanuit FAC zelf (regel 790). Nu wordt het terugkeeradres REKEN op de stack gezet. Vervolgens wordt (A), nu 3, op de stack geplaatst. Regel 770 is nu twee keer uitgevoerd. Weer wordt (A) verlaagd en wordt FAC aangeroepen, waardoor nogmaals REKEN op de stack wordt gezet. Bij het voor de derde maal uitvoeren van de push-instructie van regel 770 komt het getal 2 op de stack te staan. Hierna wordt de routine afgemaakt door A vanaf de stack te laden, dus met 2, en vervolgens $2! = 2 \cdot 1! = 2$ te berekenen en terug te springen naar het aanroepende programma (FAC zelf). Zie regel 805 (RTS), waarbij het terugkeeradres REKEN van de stack wordt gehaald. Opnieuw wordt A geladen vanaf de stack (nu met de waarde 3),

waarna $3! = 3 \cdot 2! = 3 \cdot 2 = 6$ wordt gevonden. Voor de tweede keer wordt daarna de instructie RTS (regel 805) uitgevoerd, waardoor nogmaals wordt teruggekeerd naar REKEN (regel 795). Weer wordt A geladen vanaf de stack, zodat (A) ditmaal gelijk wordt aan 4 en $4! = 4 \cdot 3! = 4 \cdot 6 = 24$ wordt uitgerekend met behulp van de subroutine MULTIP. Voor de derde (en laatste) keer wordt in regel 805 het terugkeeradres van de stack gehaald. Deze keer is dit BACK, zodat er wordt teruggesprongen naar FACULT. De recursieve subroutine FAC is hiermee geheel uitgevoerd.

Tenslotte moet de subroutine MULTIP nog worden geschreven ten behoeve van de vermenigvuldiging van (A) met NFACUL. NFACUL kan uit meerdere bytes bestaan en het nieuwe produkt kan 1 byte meer bevatten.

Stel dat NFACUL uit drie bytes bestaat: OP3, OP2 en OP1. Er geldt dan $LOLD = 3$.

Fig. 12.16 laat zien hoe de vermenigvuldiging van het oude produkt met (A) in zijn werk gaat. (A) wordt met elke byte afzonderlijk vermenigvuldigd. De deelprodukten worden, telkens 1 byte verschoven, bij elkaar opgeteld. De som levert het nieuwe produkt, bestaande uit de bytes NP4, NP3, NP2 en NP1.

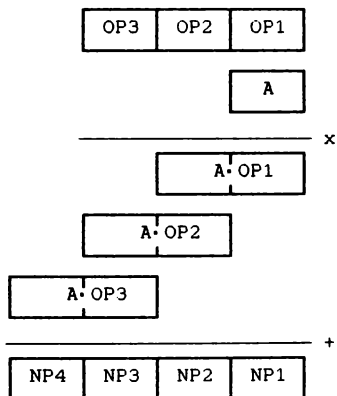


Fig. 12.16 Werkwijze bij vermenigvuldigen

Vermenigvuldigen geschiedt met behulp van de instructie MUL, waarbij $(D) := (A) \cdot (B)$. Hierbij gaat de inhoud van A verloren. Daar (A) niet alleen vermenigvuldigd moet worden met OP1, maar ook met OP2, enzovoorts, moet (A) tijdelijk worden bewaard (in de local storage, als variabele AVALUE).

B wordt beurtelings geladen met OP1, OP2, enzovoorts. De high-byte (A) van het produkt $(A) \cdot OP(i)$ moet worden opgeteld bij de low-byte van het volgende produkt $(A) \cdot OP(i+1)$. De carry die hierbij ontstaat, moet worden gesommeerd bij de high-byte van $(A) \cdot OP(i+1)$.

De high-byte (PRODH) van een produkt moet derhalve worden bewaard (in de local storage) tot het volgende produkt is berekend.

Bovendien wordt er een teller LNEW verhoogd, telkens na een vermenigvuldiging. LNEW houdt bij uit hoeveel bytes het nieuwe produkt bestaat. Na drie vermenigvuldigingen kan het nieuwe produkt uit drie of uit vier bytes bestaan. Is NP4 gelijk aan 0, dan bestaat het nieuwe produkt, evenals het oude produkt, uit drie bytes. Is NP4 echter ongelijk aan 0, dan bevat het nieuwe produkt vier bytes en moet LNEW een keer extra worden verhoogd.

Ten behoeve van de volgende berekening van NFACUL moet LOLD tenslotte de waarde van LNEW overnemen.

LNEW wordt opgeslagen in de local storage en moet de beginwaarde 0 krijgen.

Fig. 12.17 laat het structogram zien van de subroutine MULTIP. FACUL bevindt zich in de uitkomstruimte.

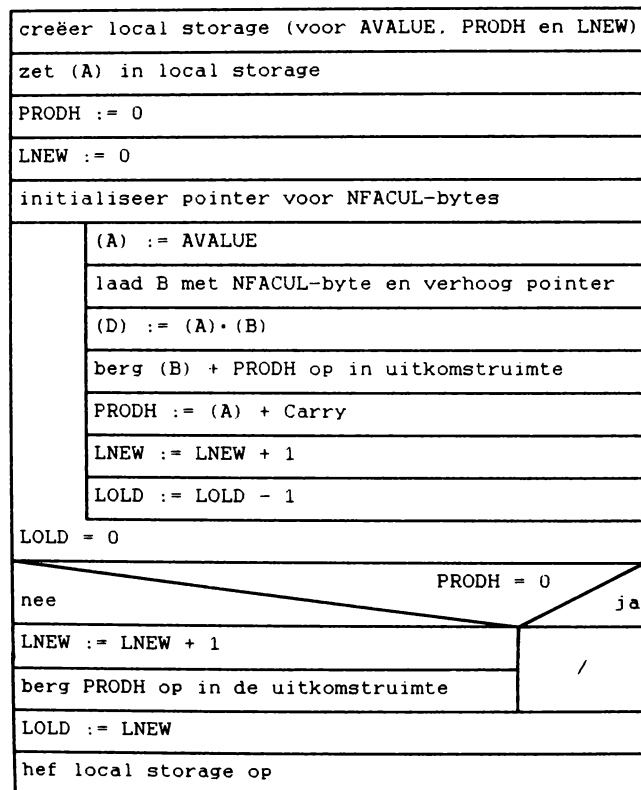


Fig. 12.17 Structogram van de subroutine MULTIP

```

820 *****
825 * Subroutine MULTIP *
830 *****
835 *
840 *MULTIP berekent het produkt (A)•NFACUL.
845 *De lengte (in bytes) van NFACUL bevindt zich als lower
850 *level global LOLD op de stack (offset is -1).
855 *
860 *De registers A, B, CC, X en Y veranderen van inhoud.
865 *
870 *
875 LOLD EQU -1 global: oude lengte van NFACUL
880 AVALUE EQU 0 local voor redden van (A)
885 PRODH EQU 1 local: high-byte van (A)•(B)
890 LNEW EQU 2 local: nieuwe lengte van NFACUL
895 *
900 MULTIP LEAS -3,S creëer local storage
905 STA AVALUE,S
910 CLR PRODH,S
915 CLR LNEW,S
920 TFR X,Y Y wijst naar low-byte NFACUL
925 VERMEN LDA AVALUE
930 LDB ,Y+ (B):=NFACUL-byte; verhoog pointer
935 MUL
940 ADDB PRODH,S
945 STB -1,Y berg nieuwe NFACUL-byte op
950 ADCA #0
955 STA PRODH,S PRODH := high-byte + C
960 INC LNEW,S
965 DEC LOLD,U
970 BNE VERMEN
975 TST PRODH,S
980 BEQ KLAAR spring als PRODH=0
982 INC LNEW,S
984 LDA PRODH,S
986 STA ,Y berg highest NFACUL-byte op
988 KLAAR LDA LNEW,S
990 STA LOLD,U LOLD:=LNEW
992 LEAS 3,S hef local storage op
994 RTS

```

Regel 900 kan vervallen, als de regels 905, 910 en 915 worden vervangen door:

```

905 MULTIP CLR ,S LNEW := 0
910 CLR ,S PRODH := 0
915 STA ,S AVALUE := (A)

```

Hierbij wordt in totaal S met 3 verlaagd en hebben AVALUE, PRODH en LNEW achtereenvolgens de offsets 0, 1 en 2 ten opzichte van S. Vanwege de leesbaarheid is deze kortere oplossing niet gekozen.

Door het kiezen van labels in plaats van getallen, kan het

programma op eenvoudige wijze worden uitgebreid voor grotere getallen. Hiertoe hoeft slechts de waarde van MAX in regel 105 te worden gewijzigd en moet er voldoende ruimte worden gedefinieerd ten behoeve van de uitkomst. Hiertoe hoeft slechts de waarde van HIBYTE in regel 305 te worden aangepast. Overigens kan deze waarde niet groter zijn dan \$FF, omdat anders de grens van de geheugenpagina wordt overschreden. Moet de uitkomst meer dan 1024 bytes omvatten, dan zijn er enkele extra wijzigingen van het programma nodig.

Voorbeeld 12.15 laat zien dat een vrij complex probleem kan worden verdeeld in een aantal eenvoudige deelproblemen. Bovendien is een ingewikkeld programma goed te begrijpen, indien dit programma wordt onderverdeeld in een aantal kleine routines, die elk een afgeronde taak uitvoeren. Per routine moet beschreven worden wat de functie van die routine is. Op deze manier ontstaat een goed gestructureerd en gedocumenteerd programma. Men dient zich wel strict aan de structogrammen te houden. Dat het programma iets minder efficiënt is voor wat betreft het gebruik van geheugenruimte en executietijd, is van ondergeschikt belang. Voorop staat de overdraagbaarheid en de wijzigbaarheid van het programma, met andere woorden, het aantal manuren.

12.7 Samenvatting

Lokale of tijdelijke variabelen zijn variabelen

die gebruikt worden tijdens het uitvoeren van een bepaalde module, en die buiten die module niet van belang zijn. Ze worden dus tijdelijk gebruikt.

Globale of blijvende variabelen zijn variabelen

die gebruikt worden door meerdere modules. Ze hebben derhalve een duurzamer karakter.

Overigens kunnen deze globals van een verschillend niveau zijn. Globals van het hoogste niveau kunnen door elke module worden gebruikt. Globals van lagere niveaus bestaan alleen voor alle modules die binnen en onder de module liggen, waarin ze zijn gedefinieerd.

Geheugengebruik voor locals en globals:

registers	vaste geheugenlocaties	stacklocaties
snel	traag	traag
positie-onafhankelijk	positie-afhankelijk	positie-onafhankelijk
klein aantal	groot aantal	groot aantal
kost geen geheugen	blijvend gereserveerd	kan worden opgeheven
eenvoudig	eenvoudig	complex
voor locals	voor locals en globals	voor locals; voor globals meestal stack mark nodig

Parameteroverdracht kan geschieden via

1. registers
2. vaste geheugenlocaties
3. de stack
4. een pointer
 - a. een 16-bits register
 - b. het direct page register
 - c. 2 vaste geheugenlocaties
 - d. 2 stacklocaties

Een positie-onafhankelijk programma is een programma dat correct functioneert, ongeacht de plaats in het geheugen, waar het is geladen.

Een programma moet positie-onafhankelijk zijn, indien

1. het programma in ROM staat en het basisadres van de ROM afhankelijk is van de hardware-configuratie.
2. het programma niet telkens wordt geladen in hetzelfde gebied, maar in een gebied dat op een bepaald moment vrij is.

Voorwaarde voor positie-onafhankelijkheid:

Gebruik geen absolute adressen.

MC6809-programma's kunnen positie-onafhankelijk zijn, dankzij

1. branch- en long branch-instructies voor het springen
2. program counter relative addressing instructies voor het ophalen en opbergen van data
3. load effective address instructies voor het initialiseren van pointers

Een reentrant routine is een routine

waarbij geen gegevens van het aanroepende programma verloren gaan, indien deze routine door een ander programma wordt aangeroepen, voordat de routine beëindigd is.

Een routine moet reentrant zijn, indien

deze routine door van elkaar onafhankelijke en onderbreekbare programma's kan worden aangeroepen.

Voorwaarde voor reentrancy:

Het werkgeheugen, dat door een reentrant routine wordt gebruikt, moet uitsluitend zijn toegewezen aan het aanroepende programma.

Een routine is reentrant, indien

1. het aanroepende programma een werkruimte creëert, of indien
2. aan het begin van de routine een werkruimte wordt gecreëerd, die zodoende aan het aanroepende programma wordt toegewezen. Dit gaat het eenvoudigste op de stack. Het werkgeheugen moet zich natuurlijk in een vrij geheugengebied bevinden.

Een recursieve routine is een routine die zichzelf aanroept.

Recursieve routines zijn niet noodzakelijk.

Elk probleem kan worden opgelost met behulp van iteratie (herhaling).

Het werken met recursieve routines maakt het programma compacter. Een recursieve functie is eenvoudig te doorzien.

Voorwaarden voor recursie:

1. Registerinhouden die voor de routine zelf van belang zijn, moeten telkens aan het begin van de routine in een vrije geheugenruimte worden opgeborgen.
2. Aan het begin van de routine moet een werkgeheugen worden gecreëerd in een vrije geheugenruimte.
Als vrije geheugenruimte kan het eenvoudigst de stack worden gebruikt.

Fig. 12.18 toont de structuur van een recursieve routine RECROU.

Vaak wordt de initiële berekening in het aanroepende programma uitgevoerd en kan dan in de recursieve routine achterwege blijven.

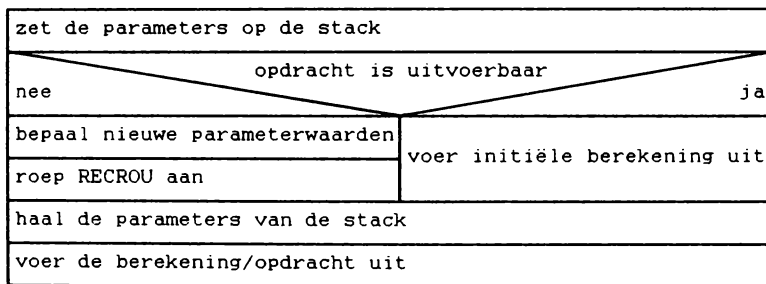


Fig. 12.18 Structuur van de recursieve routine RECROU

12.8 Opgaven

- 1 Definieer een local storage van 20 bytes, die zich direct achter achter de module MODU in het geheugen bevindt. Initialiseer de pointer voor die local storage op een positie-onafhankelijke wijze.
- 2 Vervang de instructie (L)BRA VERDER door een positie-onafhankelijke JMP-instructie. Geef de voordelen en nadelen van een positie-onafhankelijke JMP-instructie.

- 3 Geef met behulp van instructies nauwkeurig aan hoe een parameter via register X kan worden overgedragen van een IRQ-interrupt service routine naar het onderbroken programma.

- 4 In een programma PRGR wordt de subroutine CALC aangeroepen. CALC berekent de absolute waarde (ABS) en het kwadraat (KWAD) van een two's complement getal (NUM). In PRGR moeten de nodige acties worden uitgevoerd ten behoeve van de parameteroverdracht van de gegevens naar en vanuit CALC.

De geheugenindeling is als volgt:

NUM	RMB	1
ABS	RMB	1
KWAD	RMB	2

Schrijf het gedeelte van PRGR dat nodig is voor de parametertransfer, alsmede de volledige subroutine CALC.

Doe dit op de volgende twee manieren:

- parametertransfer via de stack
- parametertransfer via een pointer op de stack

- 5 In een programma MAIN wordt de subroutine REKEN aangeroepen. REKEN bepaalt de uitkomst (VALUE) van $25 \cdot (G) + (H)$. (G) en (H) zijn 1-byte unsigned getallen. In MAIN moeten de nodige acties worden uitgevoerd ten behoeve van de parameteroverdracht van de gegevens naar en vanuit REKEN.

De geheugenindeling is als volgt:

G	RMB	1
H	RMB	1
VALUE	RMB	2

Schrijf het gedeelte van MAIN dat nodig is voor de parametertransfer, alsmede de volledige subroutine REKEN.

Doe dit voor de volgende manieren van parametertransfer:

- via registers
- via vaste geheugenlocaties
- via een pointer op 2 vaste geheugenlocaties

- 6 Gegeven:

	LDX	#ACTTAB
	LDA	SWITCH
	ASLA	
	JSR	[A,X]
	SWI	
ACTTAB	FDB	\$F100,\$F140,\$F180,\$F180,\$F200,\$F240,\$F2C0
	FDB	\$F300,\$F340,\$F350,\$F370,\$F390,\$F3A0,\$F3D0
	FDB	\$F400,\$F450,\$F460,\$F480,\$F4B0,\$F4F0,\$F510
	FDB	\$F530,\$F550,\$F570,\$F5A0,\$F5D0,\$F5F0
SWITCH	FCB	\$0C

Op welk adres begint de aangeroepen subroutine?

7 Gegeven: ---

```
      LDA    VAR
      STA    4,S
      ---
      BSR    WERK
TERUG  ---
      ---
      SWI
WERK   PSHS   A,X,Y
      LEAS   -50,S
      ---
      LDA    OFF,S
      ---
      LEAS   50,S
      PULS   A,X,Y
      RTS
```

- a. Teken nauwkeurig de opbouw van de stack op het moment dat de instructie LDA OFF,S wordt uitgevoerd.
- b. Bereken tevens de waarde van OFF, opdat de accumulator wordt geladen met de waarde van VAR.

8 Gegeven:

```
STRNG1 FCB    $AA,$A0,$A9,$AC,--,--,.....
STRNG2 FCB    $BB,$B5,$B0,$BF,--,--,.....
BEGIN  LDX    #STRNG1
      LEAY   STRNG2,PC
      PSHS   X,Y
      JSR    COMPAR
WEER    ---
      ---
      SWI
COMPAR  LDA    [CON1,S]
      LDB    [CON2,S]
      ---
      RTS
```

- a. Geef de opbouw van de stack op het moment dat de LDB-instructie wordt uitgevoerd.
- b. Bepaal de waarden van CON1 en CON2, opdat A en B worden geladen met de waarden \$AA, respectievelijk \$BB.

9 Schrijf een positie-onafhankelijke subroutine MOVE.

MOVE plaatst de inhoud van een geheugenblok met lengte LEN (> 0) een aantal locaties DISPL verderop in het geheugen. Ga uit van de volgende geheugenindeling:

```
MOVE    ---
      ---
      RTS
DISPL    RMB    1
LEN      RMB    1
BLOK     RMB    1                begin van geheugenblok
```

- 10 Schrijf een positie-onafhankelijke subroutine ZOEK.
 ZOEK gaat na of een tabel van 16-bits getallen de waarde (KENTAL) bevat en zet het aantal keren dat (KENTAL) voorkomt in het geheugenblok in A en het laagste adres van het gevonden kental in X.
 Het geheugenblok bevat tenminste één 16-bits woord.
 De geheugenindeling is als volgt:

```
ZOEK      ---
          ---
          RTS
KENTAL    RMB      2
AANTAL    RMB      2
TABEL     RMB      1                begin van de tabel
```

- 11 Schrijf een reentrant subroutine MULT2.
 MULT2 berekent het produkt van twee two's complement getallen, die zich bevinden in A en in B.
 De uitkomst moet komen te staan in D.
 Volg de onderstaande procedure:
- ```
 reset FLAG
 als (A) negatief, dan (A):=-(A) en FLAG:=FLAG+1
 als (B) negatief, dan (B):=-(B) en FLAG:=FLAG+1
 (D) := (A)·(B)
 als FLAG oneven, dan (D):=-(D)
```
- 12 Schrijf een reentrant subroutine VOLG, die de two's complement getallen in X, Y en U in volgorde zet, zodanig dat X het kleinste, Y het middelste en U het grootste getal bevat bij het verlaten van de subroutine.
- 13 Schrijf een recursieve routine MACHT voor het berekenen van  $Q^M$ . Q en M zijn gehele positieve 1-byte getallen ( $M > 0$ ).  
 Neem aan dat de uitkomst in 1 byte past.  
 Gebruik de recursieve formule
- $$\begin{aligned} \text{exp}(M) &= Q \cdot \text{exp}(M-1) && \text{voor } M > 0 \\ \text{exp}(M) &= 1 && \text{voor } M = 0 \end{aligned}$$
- De parameteroverdracht geschiedt via vaste geheugenlocaties:
- ```
M      RMB      1
Q      RMB      1
EXP     RMB      1
```
- 14 Schrijf een recursieve routine TELOP voor het berekenen van de som $1 + 2 + 3 + \dots + N$ volgens de recursieve formule
- $$\begin{aligned} \text{som}(N) &= N + \text{som}(N-1) && \text{voor } N > 0 \\ \text{som}(N) &= 0 && \text{voor } N = 0 \end{aligned}$$
- Gegeven is dat N een geheel positief 1-byte getal is.
 N wordt ingevoerd via B en de som wordt uitgevoerd via X.

APPENDIX A

ASCII-TABEL

MSB	0	1	2	3	4	5	6	7
LSB								
0	NUL	DLE	SP	0	@	P	~	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	:
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

NUL: NULl

SOH: Start Of Heading

STX: Start of TeXt

ETX: End of TeXt

EOT: End Of Transmission

ENQ: ENQuiry

ACK: ACKnowledge

BEL: BELl or alarm

BS: Back Space

HT: Horizontal Tabulation

LF: Line Feed

VT: Vertical Tabulation

FF: Form Feed

CR: Carriage Return

SO: Shift Out

SI: Shift In

DLE: Data Link Escape

DC1: Device Control 1

DC2: Device Control 2

DC3: Device Control 3

DC4: Device Control 4

NAK: Negative Acknowledge

SYN: SYNchronous idle

ETB: End of Transmission Block

CAN: CANcel

EM: End of Medium

SUB: SUBstitute

ESC: ESCape

FS: File Separator

GS: Group Separator

RS: Record Separator

US: Unit Separator

SP: SSpace

DEL: DELe

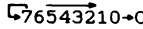
APPENDIX B

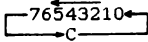
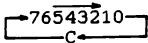
INSTRUCTIESET

Indeling en gebruikte symbolen:

- Kolom "instr" vermeldt de mnemonics van de instructies.
- Kolom "op" geeft de hexadecimale notatie van de opcodes.
- Onder "~" vindt u de duur van de instructies, uitgedrukt in het aantal klokcycli.
- Bij "+" moeten nog een aantal klokcycli worden bijgeteld om de executietijd van de instructies te vinden. Dit bij te tellen aantal is vermeld in kolom "+" van de postbyte-tabel van de indexed en indirect addressing instructies.
- In kolom "*" is aangegeven uit hoeveel bytes de instructies bestaan.
- De indexed en indirect addressing instructies bevatten het aantal bytes dat vermeld is in kolom "+*", vermeerderd met het aantal van de postbyte-kolom "+*".
- Onder "beschrijving" staat een korte uitleg van de instructies.
- In de kolommen "H", "N", "Z", "V" en "C" wordt vermeld wat er met de betreffende bits van het condition code register gebeurt. Hierbij zijn de volgende symbolen gebruikt:
 - "-": bit verandert niet
 - "*": bit wordt aangepast aan het resultaat van de instructie
 - "0": bit wordt altijd gereset
 - "1": bit wordt altijd geset
 - "?": de waarde van de bit is niet bekend
- Een kleine letter verwijst naar een opmerking.
- Met (M) (M+1) wordt een 2-bytes getal aangeduid, waarbij de high-byte op adres M staat en de low-byte op adres M+1.

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ *	direct op ~ *	indexed. indirect op ~+ **	extend op ~ *	inher op ~ *		
ABX					3A 3 1	(X) := (X) + (B)	a -----
ADCA	89 2 2	99 4 2	A9 4+ 2+	B9 5 3		(A) := (A) + (M) + C	*****
ADCB	C9 2 2	D9 4 2	E9 4+ 2+	F9 5 3		(B) := (B) + (M) + C	*****
ADDA	8B 2 2	9B 4 2	AB 4+ 2+	BB 5 3		(A) := (A) + (M)	*****
ADDB	CB 2 2	DB 4 2	EB 4+ 2+	FB 5 3		(B) := (B) + (M)	*****
ADDD	C3 4 3	D3 6 2	E3 6+ 2+	F3 7 3		(D) := (D) + (M) (M+1)	-----
ANDA	84 2 2	94 4 2	A4 4+ 2+	B4 5 3		(A) := (A) AND (M)	-**0-
ANDB	C4 2 2	D4 4 2	E4 4+ 2+	F4 5 3		(B) := (B) AND (M)	-**0-
ANDCC	1C 3 2					(CC) := (CC) AND M	bbbbbb
ASLA					48 2 1	A	?****
ASLB					58 2 1	B	?****
ASL		08 6 2	68 6+ 2+	78 7 3		M C-76543210-0	?****

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
ASRA ASRB ASR		07 6 2	67 6+ 2+	77 7 3	47 2 1 57 2 1	A B  C M	***- ***- ***-
BITA BITB	85 2 2 C5 2 2	95 4 2 D5 4 2	A5 4+ 2+ E5 4+ 2+	B5 5 3 F5 5 3		(A) AND (M) (B) AND (M)	***0- ***0-
CLRA CLRB CLR		0F 6 2	6F 6+ 2+	7F 7 3	4F 2 1 5F 2 1	(A) := 0 (B) := 0 (M) := 0	*0100 *0100 *0100
CMPPA CMPB CMPD	81 2 2 C1 2 2 10 5 4	91 4 2 D1 4 2 10 7 3	A1 4+ 2+ E1 4+ 2+ 10 7+ 3+	B1 5 3 F1 5 3 10 8 4		(A) - (M) (B) - (M) (D) - (M) (M+1)	***** ***** *****
CMPS	83 11 5 4	93 11 7 3	A3 11 7+ 3+	B3 11 8 4		(S) - (M) (M+1)	*****
CMPU	8C 11 5 4	9C 11 7 3	AC 11 7+ 3+	BC 11 8 4		(U) - (M) (M+1)	*****
CMPX CMPY	8C 8C 4 3 10 5 4 8C	9C 9C 6 2 10 7 3 9C	AC AC 6+ 2+ 10 7+ 3+ AC	BC BC 7 3 10 8 4 BC		(X) - (M) (M+1) (Y) - (M) (M+1)	***** *****
COMA COMB COM		03 6 2	63 6+ 2+	73 7 3	43 2 1 53 2 1	(A) := NOT (A) (B) := NOT (B) (M) := NOT (M)	***01 ***01 ***01
CWAI	3C c 2					(CC) := (CC) AND (M) and wait for int.	bbbbbb
DAA					19 2 1	decimal adjust A	***0*
DECA DECB DEC		0A 6 2	6A 6+ 2+	7A 7 3	4A 2 1 5A 2 1	(A) := (A) - 1 (B) := (B) - 1 (M) := (M) - 1	****- ****- ****-
EORA EORB	88 2 2 C8 2 2	98 4 2 D8 4 2	A8 4+ 2+ E8 4+ 2+	B8 5 3 F8 5 3		(A) := (A) EOR (M) (B) := (B) EOR (M)	***0- ***0-
EXG	1E 8 2					wissel (R1), (R2)	-----
INCA INCB INC		0C 6 2	6C 6+ 2+	7C 7 3	4C 2 1 5C 2 1	(A) := (A) + 1 (B) := (B) + 1 (M) := (M) + 1	****- ****- ****-
JMP JSR		0E 3 2 9D 7 2	6E 3+ 2+ AD 7+ 2+	7E 4 3 BD 8 3		jump jump to subr.	----- -----
LDA LDB LDD LDS	86 2 2 C6 2 2 CC 3 3 10 4 4 CE	96 4 2 D6 4 2 DC 5 2 10 6 3 DE	A6 4+ 2+ E6 4+ 2+ EC 5+ 2+ 10 6+ 3+ EE	B6 5 3 F6 5 3 FC 6 3 10 7 4 FE		(A) := (M) (B) := (M) (D) := (M) (M+1) (S) := (M) (M+1)	***0- ***0- ***0- ***0-

instr	addressing modes					beschrijving	(CC) bit
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ #+	extend op ~ #	inher op ~ #		
LDU	CE 3 3	DE 5 2	EE 5+ 2+	FE 6 3		(U) := (M) (M+1)	---*0-
LDX	8E 3 3	9E 5 2	AE 5+ 2+	BE 6 3		(X) := (M) (M+1)	---*0-
LDY	10 4 4	10 6 3	10 6+ 3+	10 7 4		(Y) := (M) (M+1)	---*0-
	8E	9E	AE	BE			
LEAS			32 4+ 2+			(S) := eff. adres	-----
LEAU			33 4+ 2+			(U) := eff. adres	-----
LEAX			30 4+ 2+			(X) := eff. adres	---*-
LEAY			31 4+ 2+			(Y) := eff. adres	---*-
LSLA					48 2 1	A	?****
LSLB					58 2 1	B C ← 76543210 → 0	?****
LSL		08 6 2	68 6+ 2+	78 7 3		M	?****
LSRA					44 2 1	A	-0*-*
LSRB					54 2 1	B 0 ← 76543210 → C	-0*-*
LSR		04 6 2	64 6+ 2+	74 7 3		M	-0*-*
MUL					3D d 1	(D) := (A) * (B)	a ---*-e
NEGA					40 2 1	(A) := -(A)	?****
NEGB					50 2 1	(B) := -(B)	?****
NEG		00 6 2	60 6+ 2+	70 7 3		(M) := -(M)	?****
NOP					12 2 1	no operation	-----
ORA	8A 2 2	9A 4 2	AA 4+ 2+	BA 5 3		(A) := (A) OR (M)	---*0-
ORB	CA 2 2	DA 4 2	EA 4+ 2+	FA 5 3		(B) := (B) OR (M)	---*0-
ORCC	1A 3 2					(CC) := (CC) OR M	fffff
PSHS	34 g 2					reg. naar S-stack	-----
PSHU	36 g 2					reg. naar U-stack	-----
PULS	35 g 2					reg. van S-stack	-----
PULU	37 g 2					reg. van U-stack	-----
ROLA					49 2 1	A	-----
ROLB					59 2 1	B	-----
ROL		09 6 2	69 6+ 2+	79 7 3		M 	-----
RORA					46 2 1	A	---*-*
RORB					56 2 1	B	---*-*
ROR		06 6 2	66 6+ 2+	76 7 3		M 	---*-*
RTI					3B h 1	return from int.	iiiiii
RTS					39 5 1	return from subr.	-----
SBCA	82 2 2	92 4 2	A2 4+ 2+	B2 5 3		(A) := (A) - (M) - C	?****
SBCB	C2 2 2	D2 4 2	E2 4+ 2+	F2 5 3		(B) := (B) - (M) - C	?****
SEX					1D 2 1	Ai := B7, i=0 t/m 7	---*0-

instr	addressing modes					beschrijving	(CC) bit 53210 HNZVC
	immed op ~ #	direct op ~ #	indexed, indirect op ~+ ##	extend op ~ #	inher op ~ #		
STA		97 4 2	A7 4+ 2+	B7 5 3		(M) := (A)	---*0-
STB		D7 4 2	E7 4+ 2+	F7 5 3		(M) := (B)	---*0-
STD		DD 5 2	ED 5+ 2+	FD 6 3		(M) (M+1) := (D)	---*0-
STS		10 6 3	10 6+ 3+	10 7 4		(M) (M+1) := (S)	---*0-
		DF	EF	FF			
STU		DF 5 2	EF 5+ 2+	FF 6 3		(M) (M+1) := (U)	---*0-
STX		9F 5 2	AF 5+ 2+	BF 6 3		(M) (M+1) := (X)	---*0-
STY		10 6 3	10 6+ 3+	10 7 4		(M) (M+1) := (Y)	---*0-
		9F	AF	BF			
SUBA	80 2 2	90 4 2	A0 4+ 2+	B0 5 3		(A) := (A) - (M)	?****
SUBB	C0 2 2	D0 4 2	E0 4+ 2+	F0 5 3		(B) := (B) - (M)	?****
SUBD	83 4 3	93 6 2	A3 6+ 2+	B3 7 3		(D) := (D) - (M) (M+1)	-----
SWI					3F j 1	softw. int. 1	1-----
SWI2					10 k 2	softw. int. 2	-----
					3F		
SWI3					11 k 2	softw. int. 3	-----
					3F		
SYNC					13 m 1	synchr. met int.	-----
TFR	1F 6 2					(R2) := (R1)	-----
TSTA					4D 2 1	(A) - 0	---*0-
TSTB					5D 2 1	(B) - 0	---*0-
TST		0D 6 2	6D 6+ 2+	7D 7 3		(M) - 0	---*0-

instr	addr. mode		spring als	(CC) bit 53210 HNZVC
	relat op ~ #			
BCC	24 3 2	C=0		-----
LBCC	10 n 4	C=0		-----
	24			
BCS	25 3 2	C=1		-----
LBCCS	10 n 4	C=1		-----
	25			
BEQ	27 3 2	Z=1		-----
LBEQ	10 n 4	Z=1		-----
	27			
BGE	2C 3 2	(N EOR V) = 0; (R) ≥ (M)		o-----
LBGE	10 n 4	(N EOR V) = 0; (R) ≥ (M)		o-----
	2C			
BGT	2E 3 2	(Z OR (N EOR V)) = 0; (R) > (M)		o-----
LBGT	10 n 4	(Z OR (N EOR V)) = 0; (R) > (M)		o-----
	2E			
BHI	22 3 2	(C OR Z) = 0; (R) > (M)		a-----
LBHI	10 n 4	(C OR Z) = 0; (R) > (M)		a-----
	22			

instr	addr. mode	spring als	(CC) bit 53210 HNZVC
		relat op ~ #	
BHS	24 3 2	C = 0; (R) $\geq$ (M)	a -----
LBHS	10 n 4	C = 0; (R) $\geq$ (M)	a -----
	24		
BLE	2F 3 2	(Z OR (N EOR V)) = 1; (R) $\leq$ (M)	o -----
LBLE	10 n 4	(Z OR (N EOR V)) = 1; (R) $\leq$ (M)	o -----
	2F		
BLO	25 3 2	C=1; (R) < (M)	a -----
LBLO	10 n 4	C=1; (R) < (M)	a -----
	25		
BLS	23 3 2	(C OR Z) = 1; (R) $\leq$ (M)	a -----
LBLS	10 n 4	(C OR Z) = 1; (R) $\leq$ (M)	a -----
	23		
BLT	2D 3 2	(N EOR V) = 1; (R) < (M)	o -----
LBLT	10 n 4	(N EOR V) = 1; (R) < (M)	o -----
	2D		
BMI	2B 3 2	N=1	-----
LBMI	10 n 4	N=1	-----
	2B		
BNE	26 3 2	Z=0	-----
LBNE	10 n 4	Z=0	-----
	26		
BPL	2A 3 2	N=0	-----
LBPL	10 n 4	N=0	-----
	2A		
BRA	20 3 2	altijd	-----
LBRA	16 5 3	altijd	-----
BRN	21 3 2	nooit	-----
LBRN	10 5 4	nooit	-----
	21		
BSR	8D 7 2	altijd naar subroutine	-----
LBSR	17 9 3	altijd naar subroutine	-----
	8D		
BVC	28 3 2	V=0	-----
LBVC	10 n 4	V=0	-----
	28		
BVS	29 3 2	V=1	-----
LBVS	10 n 4	V=1	-----
	29		

Opmerkingen:

- Unsigned.
- Flags worden gereset of veranderen niet, afhankelijk van de waarde van het masker.
- Tenminste 20 klokcycli.
- 11 klokcycli.
- C := bit 7 van accumulator B.
- Flags worden geset of veranderen niet, afhankelijk van de waarde van het masker.
- 5 klokcycli + 1 cyclus voor elke byte die wordt gepushed of gepulled.
- 6 klokcycli na FI, 15 klokcycli na andere interrupt.
- (CC) wordt van de stack gehaald.
- 19 klokcycli.
- 20 klokcycli.

- l. SWI set de condition code bits I en F.
- m. Tenminste 4 klokcycli.
- n. 5 cycli als er niet wordt gesprongen, 6 als er wel wordt gesprongen.
- o. Two's complement.

Overzicht van de voorwaardelijke sprongen

instr.	voorwaarde
(L)BMI	N=1
(L)BEQ	Z=1
(L)BVS	V=1
(L)BCS	C=1

instr.	voorwaarde
(L)BPL	N=0
(L)BNE	Z=0
(L)BVC	V=0
(L)BCC	C=0

unsigned		beschrijving	two's complement	
instr.	voorwaarde		instr.	voorwaarde
(L)BLO	C=1	kleiner dan	(L)BLT	(N EOR V)=1
(L)BLS	(C OR Z)=1	kleiner of gelijk	(L)BLE	(Z OR (N EOR V))=1
(L)BEQ	Z=1	gelijk aan	(L)BEQ	Z=1
(L)BHS	C=0	groter of gelijk	(L)BGE	(N EOR V)=0
(L)BHI	(C OR Z)=0	groter dan	(L)BGT	(Z OR (N EOR V))=0
(L)BNE	Z=0	ongelijk aan	(L)BNE	Z=0

Postbyte bij indexed addressing en indirect addressing

mode	operand	indexed postbyte	+ + ~ #	operand	indirect + + postbyte	~ #
0-bits offset	,R	1RR0 0100	0 0	[.R]	1RR1 0100	3 0
5-bits offset	n,R	0RRn nnnn	1 0		a	
8-bits offset	n,R	1RR0 1000	1 1	[n,R]	1RR1 1000	4 1
16-bits offset	n,R	1RR0 1001	4 2	[n,R]	1RR1 1001	7 2
acc. A offset	A,R	1RR0 0110	1 0	[A,R]	1RR1 0110	4 0
acc. B offset	B,R	1RR0 0101	1 0	[B,R]	1RR1 0101	4 0
acc. D offset	D,R	1RR0 1011	4 0	[D,R]	1RR1 1011	7 0
increment met 1	,R+	1RR0 0000	2 0		b	
increment met 2	,R++	1RR0 0001	3 0	[,R++]	1RR1 0001	6 0
decrement met 1	,-R	1RR0 0010	2 0		b	
decrement met 2	,--R	RRR0 0011	3 0	[,--R]	1RR1 0011	6 0
8-bits offset van PC	n,PC	1XX0 1100	1 1	[n,PC]	1XX1 1100	4 1
16-bits offset van PC	n,PC	1XX0 1101	5 2	[n,PC]	1XX1 1101	8 2
extended indirect	-	-		[n]	1001 1111	5 2

X = don't care

n = constante

nnnnn = binaire waarde van 5-bits offset

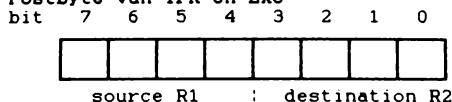
RR = indexregister: 00=X 01=Y
10=U 11=S

Opmerkingen:

a. De 5-bits offset wordt een 8-bits offset.

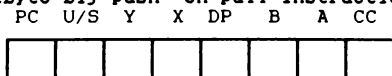
b. Niet toegestaan.

Postbyte van TFR en EXG



0000-D 0100-S 1000-A
 0001-X 0101-PC 1001-B
 0010-Y 1010-CC
 0011-U 1011-DP

Postbyte bij push- en pull-instructies



→ push-volgorde

← pull-volgorde

Numeriek oplopende opcodes

op	mnem.	addr.	~ #	op	mnem.	addr.	~ #	op	mnem.	addr.	~ #
00	NEG	direc	6 2	03	COM	direc	6 2	04	LSR	direc	6 2
06	ROR	direc	6 2	07	ASR	direc	6 2	08	A/L SL	direc	6 2
09	ROL	direc	6 2	0A	DEC	direc	6 2	0C	INC	direc	6 2
0D	TST	direc	6 2	0E	JMP	direc	3 2	0F	CLR	direc	6 2
12	NOP	inher	2 1	13	SYNC	inher	4 1	16	LBRA	relat	5 3
17	LBSR	relat	9 3	19	DAA	inher	2 1	1A	ORCC	immed	3 2
1C	ANDCC	immed	3 2	1D	SEX	inher	2 1	1E	EXG	immed	8 2
1F	TFR	immed	6 2	20	BRA	relat	3 2	21	BRN	relat	3 2
22	BHI	relat	3 2	23	BLS	relat	3 2	24	BHS/BCC	relat	3 2
25	BLO/BCS	relat	3 2	26	BNE	relat	3 2	27	BEQ	relat	3 2
28	BVC	relat	3 2	29	BVS	relat	3 2	2A	BPL	relat	3 2
2B	BMI	relat	3 2	2C	BGE	relat	3 2	2D	BLT	relat	3 2
2E	BGT	relat	3 2	2F	BLE	relat	3 2	30	LEAX	index	4+2+
31	LEAY	index	4+2+	32	LEAS	index	4+2+	33	LEAU	index	4+2+
34	PSHS	immed	5 2	35	PULS	immed	5 2	36	PSHU	immed	5 2
37	PULU	immed	5 2	39	RTS	inher	5 1	3A	ABX	inher	3 1
3B	RTI	inher	h 1	3C	CWAI	immed	k 2	3D	MUL	inher	d 1
3F	SWI	inher	j 1	40	NEGA	inher	2 1	43	COMA	inher	2 1
44	LSRA	inher	2 1	46	RORA	inher	2 1	47	ASRA	inher	2 1
48	A/L SLA	inher	2 1	49	ROLA	inher	2 1	4A	DECA	inher	2 1
4C	INCA	inher	2 1	4D	TSTA	inher	2 1	4F	CLRA	inher	2 1
50	NEGB	inher	2 1	53	COMB	inher	2 1	54	LSRB	inher	2 1
56	RORB	inher	2 1	57	ASRB	inher	2 1	58	A/L SLB	inher	2 1
59	ROLB	inher	2 1	5A	DECB	inher	2 1	5C	INCB	inher	2 1
5D	TSTB	inher	2 1	5F	CLRB	inher	2 1	60	NEG	index	6+2+
63	COM	index	6+2+	64	LSR	index	6+2+	66	ROR	index	6+2+
67	ASR	index	6+2+	68	A/L SL	index	6+2+	69	ROL	index	6+2+
6A	DEC	index	6+2+	6C	INC	index	6+2+	6D	TST	index	6+2+
6E	JMP	index	3+2+	6F	CLR	index	6+2+	70	NEG	exten	7 3
73	COM	exten	7 3	74	LSR	exten	7 3	76	ROR	exten	7 3
77	ASR	exten	7 3	78	A/L SL	exten	7 3	79	ROL	exten	7 3
7A	DEC	exten	7 3	7C	INC	exten	7 3	7D	TST	exten	7 3
7E	JMP	exten	4 3	7F	CLR	exten	7 3	80	SUBA	immed	2 2
81	CMPA	immed	2 2	82	SBCA	immed	2 2	83	SUBD	immed	4 3
84	ANDA	immed	2 2	85	BITA	immed	2 2	86	LDA	immed	2 2
88	EORA	immed	2 2	89	ADCA	immed	2 2	8A	ORA	immed	2 2
8B	ADDA	immed	2 2	8C	CMPX	immed	4 3	8D	BSR	relat	7 2
8E	LDX	immed	3 3	90	SUBA	direc	4 2	91	CMPA	direc	4 2
92	SBCA	direc	4 2	93	SUBD	direc	6 2	94	ANDA	direc	4 2
95	BITA	direc	4 2	96	LDA	direc	4 2	97	STA	direc	4 2
98	EORA	direc	4 2	99	ADCA	direc	4 2	9A	ORA	direc	4 2
9B	ADDA	direc	4 2	9C	CMPX	direc	6 2	9D	JSR	direc	7 2
9E	LDX	direc	5 2	9F	STX	direc	5 2	A0	SUBA	index	4+2+
A1	CMPA	index	4+2+	A2	SBCA	index	4+2+	A3	SUBD	index	6+2+

op	mnem.	addr.	~ #	op	mnem.	addr.	~ #	op	mnem.	addr.	~ #
A4	ANDA	index	4+2+	A5	BITA	index	4+2+	A6	LDA	index	4+2+
A7	STA	index	4+2+	A8	EORA	index	4+2+	A9	ADCA	index	4+2+
AA	ORA	index	4+2+	AB	ADDA	index	4+2+	AC	CMPX	index	6+2+
AD	JSR	index	7+2+	AE	LDX	index	5+2+	AF	STX	index	5+2+
B0	SUBA	exten	5 3	B1	CMPA	exten	5 3	B2	SBCA	exten	5 3
B3	SUBD	exten	7 3	B4	NDA	exten	5 3	B5	BITA	exten	5 3
B6	LDA	exten	5 3	B7	STA	exten	5 3	B8	EORA	exten	5 3
B9	DCA	exten	5 3	BA	ORA	exten	5 3	BB	ADDA	exten	5 3
BC	CMPX	exten	7 3	BD	JSR	exten	8 3	BE	LDX	exten	6 3
BF	STX	exten	6 3	C0	SUBB	immed	2 2	C1	CMPB	immed	2 2
C2	SBCB	immed	2 2	C3	ADDD	immed	4 3	C4	ANDB	immed	2 2
C5	BITB	immed	2 2	C6	LDB	immed	2 2	C8	EORB	immed	2 2
C9	ADCB	immed	2 2	CA	ORB	immed	2 2	CB	ADDB	immed	2 2
CC	LDD	immed	3 3	CE	LDU	immed	3 3	D0	SUBB	direc	4 2
D1	MPB	direc	4 2	D2	SBCB	direc	4 2	D3	ADDD	direc	6 2
D4	ANDB	direc	4 2	D5	BITB	direc	4 2	D6	LDB	direc	4 2
D7	STB	direc	4 2	D8	EORB	direc	4 2	D9	ADCB	direc	4 2
DA	ORB	direc	4 2	DB	ADDB	direc	4 2	DC	LDD	direc	5 2
DD	STD	direc	5 2	DE	LDU	direc	5 2	DF	STU	direc	5 2
E0	SUBB	index	4+2+	E1	CMPB	index	4+2+	E2	SBCB	index	4+2+
E3	ADDD	index	6+2+	E4	ANDB	index	4+2+	E5	BITB	index	4+2+
E6	LDB	index	4+2+	E7	STB	index	4+2+	E8	EORB	index	4+2+
E9	ADCB	index	4+2+	EA	ORB	index	4+2+	EB	ADDB	index	4+2+
EC	LDD	index	5+2+	ED	STD	index	5+2+	EE	LDU	index	5+2+
EF	STU	index	5+2+	F0	SUBB	exten	5 3	F1	CMPB	exten	5 3
F2	SBCB	exten	5 3	F3	ADDD	exten	7 3	F4	ANDB	exten	5 3
F5	BITB	exten	5 3	F6	LDB	exten	5 3	F7	STB	exten	5 3
F8	EORB	exten	5 3	F9	ADCB	exten	5 3	FA	ORB	exten	5 3
FB	ADDB	exten	5 3	FC	LDD	exten	6 3	FD	STD	exten	6 3
FE	LDU	exten	6 3	FF	STU	exten	6 3				

op	mnem.	addr.	~ #	op	mnem.	addr.	~ #
1021	LBRN	relat	5 4	1022	LBHI	relat	n 4
1023	LBLS	relat	n 4	1024	LBHS/LBCC	relat	n 4
1025	LBCS/LBLO	relat	n 4	1026	LBNE	relat	n 4
1027	LBEQ	relat	n 4	1028	LBVC	relat	n 4
1029	LBVS	relat	n 4	102A	LBPL	relat	n 4
102B	LBMI	relat	n 4	102C	LBGE	relat	n 4
102D	LBLT	relat	n 4	102E	LBGT	relat	n 4
102F	LBLE	relat	n 4	103F	SWI2	inher	k 2
1083	CMPD	immed	5 4	108C	CMPY	immed	5 4
108E	LDY	immed	4 4	1093	CMPD	direc	7 3
109C	CMPY	direc	7 3	109E	LDY	direc	6 3
109F	STY	direc	6 3	10A3	CMPD	index	7+3+
10AC	CMPY	index	7+3+	10AE	LDY	index	6+3+
10AF	STY	index	6+3+	10B3	CMPD	exten	8 4
10BC	CMPY	exten	8 4	10BE	LDY	exten	7 4
10BF	STY	exten	7 4	10CE	LDS	immed	4 4
10DE	LDS	direc	6 3	10DF	STS	direc	6 3
10EE	LDS	index	6+3+	10EF	STS	index	6+3+
10FE	LDS	exten	7 4	10FF	STS	exten	7 4
113F	SWI3	inher	k 2	1183	CMPU	immed	5 4
118C	CMPs	immed	5 4	1193	CMPU	direc	7 3
119C	CMPs	direc	7 3	11A3	CMPU	index	7+3+
11AC	CMPs	index	7+3+	11B3	CMPU	exten	8 4
11BC	CMPs	exten	8 4				

APPENDIX C

PIA (MC6821)

Registerselectie

inputs RS1 RS0		CR-bits CRA2 CRB2		geselecteerd register
0	0	0	X	Data Direction Register DDRA
0	0	1	X	Peripheral Register PRA
0	1	X	X	Control Register CRA
1	0	X	0	Data Direction Register DDRB
1	0	X	1	Peripheral Register PRB
1	1	X	X	Control Register CRB

X = don't care

Voor * dient A, respectievelijk B te worden ingevuld.

RESET

Als de ingang RESET gelijk aan 0 is, worden alle bits van alle PIA-registers gereset.

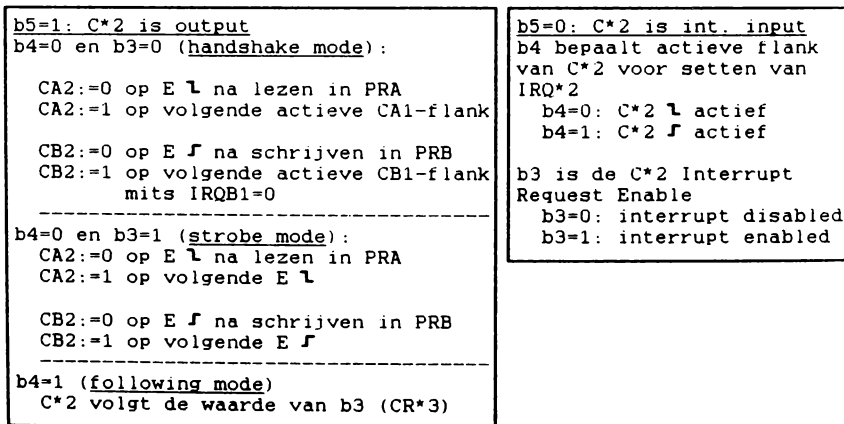
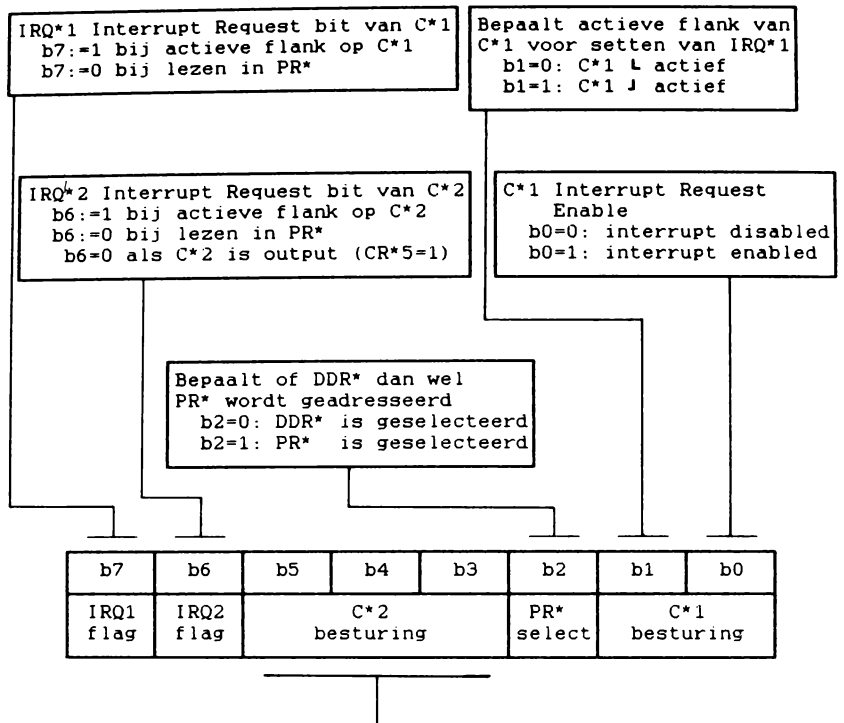
Data Direction Register

b7	b6	b5	b4	b3	b2	b1	b0
----	----	----	----	----	----	----	----

Bit $DDR*i=0$ betekent signaal $P*i$ is input

Bit $DDR*i=1$ betekent signaal $P*i$ is output

Control Register



APPENDIX D

ACIA (MC6850)

Registersselectie

RS	R/W	geselecteerde register
0	0	Control Register CR
0	1	Status Register SR
1	0	Transmit Data Register TDR
1	1	Receive Data Register RDR

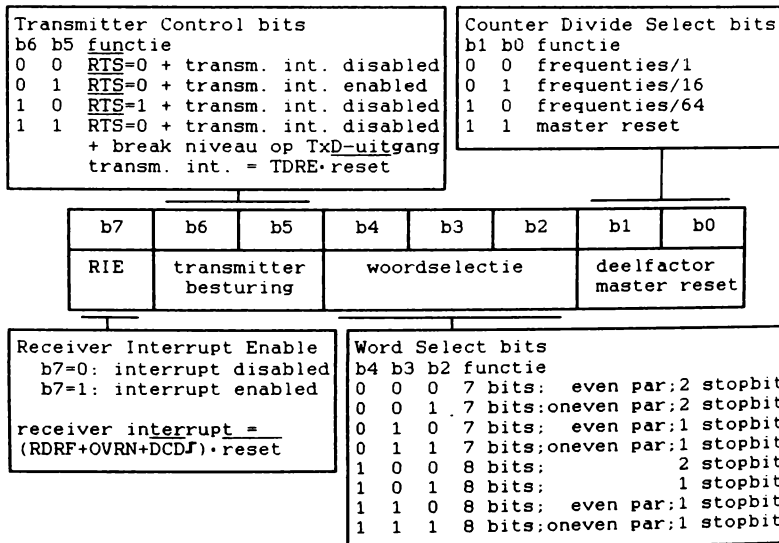
Master reset

De master reset initialiseert het zender- en het ontvangerdeel van de ACIA, reset het Status Register (behalve de bits CTS en DCD) en maakt de IRQ-uitgang hoog. De overige bits van het Control Register worden niet gewijzigd.

Gedurende de eerste master reset na het inschakelen van de voedingsspanning wordt ook de uitgang RTS hoog gemaakt. Bij de overige master resets blijft de RTS-uitgang zoals geprogrammeerd (met behulp van CR5 en CR6).

Bij het inschakelen van de voedingsspanning wordt de ACIA in een power-on reset toestand gehouden. Deze toestand wordt pas verlaten na een master reset.

Control Register



Status Register

Interrupt Request bit (= inverse van IRQ-uitgang)
 $b7 = (TIE \cdot TDRE + \overline{RIE} \cdot (RDRF + OVRN + DCD)) \cdot (\text{reset})$
 Hierin is $TIE = CR6 \cdot CR5$ en $RIE = CR7$

Transmit Data Register Empty Receive
 $b1:=1$ als TDR naar Transmit Shift
 Reg. wordt overgebracht
 $b1:=0$ bij schrijven in TDR
 $b1=0$ als $CTS=1$

Data Register Full
 $b0:=1$ als Receive Shift Reg.
 in RDR wordt gezet:
 let op PE, OVRN en FE
 $b0:=0$ bij lezen in RDR
 $b0=0$ als $DCD=1$

Clear to Send
 $b3 = CTS$ -ingang
 (onafh. van reset)

Data Carrier Detect
 $b2:=1$ bij ✓ op DCD-ingang
 $b2:=0$ na lezen in SR en RDR, mits
 DCD-ingang 0 is, anders
 volgt $b2$ deze ingang

b7	b6	b5	b4	b3	b2	b1	b0
IRQ	PE	OVRN	FE	$\overline{CTS}$	$\overline{DCD}$	TDRE	RDRF

Parity Error
 $b6:=0$ bij correcte pariteit
 $b6:=1$ bij verkeerde pariteit
 tijdens het laden van RDR
 $b6=0$ bij transport zonder par

Framing Error
 $b4:=0$ als dataframe goed is
 $b4:=1$ als dataframe fout is
 bij het laden van RDR
 (fout bij ontbreken 1ste stopbit)

Overrun
 $b5:=1$ na lezen van geldige (RDR) tijdens overrun conditie
 $b5:=0$ na lezen in RDR

APPENDIX E

PTM (MC6840)

Registersselectie

Register Select inputs RS2 RS1 RS0			geselecteerde register	
			R/W=0	R/W=1
0	0	0	Control Register 3 als CR20=0	geen enkel register
			Control Register 1 als CR20=1	geen enkel register
0	0	1	Control Register 2	Status Register
0	1	0	MSB Buffer Register	High-byte Counter 1
0	1	1	Low-byte tellerlatch 1	LSB Buffer Register
1	0	0	MSB Buffer Register	High-byte Counter 2
1	0	1	Low-byte tellerlatch 2	LSB Buffer Register
1	1	0	MSB Buffer Register	High-byte Counter 3
1	1	1	Low-byte tellerlatch 3	LSB Buffer Register

Voor * dient 1, 2, respectievelijk 3 te worden ingevuld.

Status Register

Interrupt Request bit (= inverse van $\overline{\text{IRQ}}$ -uitgang)
 $b7 = I1 \cdot CR16 + I2 \cdot CR26 + I3 \cdot CR36$

bij lezen is
 $b6=b5=b4=b3=0$

b7	b6	b5	b4	b3	b2	b1	b0
IRQ	not used				I3	I2	I1

Individuele interrupt bits I*:

I*:=1 bij de klokpuls na het 0 worden van teller *
 in de 16-bits mode, dan wel na het 0 worden van de
 high-byte van teller * in de 2x8-bits mode
 I*:=0 bij een reset, bij het schrijven in latches *, bij
 de initialisatie van teller * en bij het lezen van
 de stand van teller *, mits tevoren het Status
 Register is gelezen terwijl I* geset was

Control Register

Timer Output Enable bit:
b7=0: output 0* disabled
b7=1: output 0* enabled

Interrupt Enable bit:
b6=0: interrupt 1* disabled
b6=1: interrupt 1* enabled

Operating Mode Selection bits:
CR5 CR4 CR3 operatiemode

0	0	0	continuous mode; tellerinitialisatie bij $\overline{G^*}$ 1 of schrijven in latches * of reset
0	0	1	frequency comparison mode; interrupt als (periodesdij $\overline{G^*}$) < (time-out *)
0	1	0	continuous mode; tellerinitialisatie bij $\overline{G^*}$ 1 of reset
0	1	1	pulse width comparison mode; interrupt als (0-puls op $\overline{G^*}$) < (time-out *)
1	0	0	single shot mode; tellerinitialisatie bij $\overline{G^*}$ 1 of schrijven in latches * of reset
1	0	1	frequency comparison mode; interrupt als (periodesdij $\overline{G^*}$) > (time-out *)
1	1	0	single shot mode; tellerinitialisatie bij $\overline{G^*}$ 1 of reset
1	1	1	pulse width comparison mode; interrupt als (0-puls op $\overline{G^*}$) > (time-out *)

b7	b6	b5	b4	b3	b2	b1	b0
OE	IE	operating mode select			count select	klok select	

Clock Selection bit:

b*1=0: timer * gebruikt de externe klok $\overline{C^*}$
b*1=1: timer * gebruikt de interne klokpuls E

Counting Mode Control bit:

CR*2=0: latches * bevatten een 16-bits getal

Teller * wordt 0 na (N+1) klokperiodes vanaf $\overline{G^*}$ 1.

N is het 16-bits getal in latches *.

CR*2=1: latches * bevatten twee 8-bits getallen

Teller * wordt 0 na (L+1) * (M+1) klokperiodes vanaf $\overline{G^*}$ 1.

L is het 8-bits getal in de low-byte van latches *.

M is het 8-bits getal in de high-byte.

De minstwaardige bits van de Control Registers hebben een unieke functie:

CR10 is de interne resetbit:

CR10=0: de drie timers kunnen hun ingestelde functie uitvoeren

CR10=1: er treedt een interne reset op

CR20 is de interne adresbit voor CR1 en CR3:

CR20=0: CR3 is adresseerbaar

CR20=1: CR1 is adresseerbaar

CR30 is klokbesturingsbit van timer 3:

CR30=0: de klokfrequentie voor timer 3 wordt niet gedeeld

CR30=1: de klokfrequentie voor timer 3 wordt gedeeld door 8

De frequentiedeling kan zowel worden toegepast op de interne klokpuls E als op de externe klokpuls C3.

De deling is alleen van invloed op timer 3!

Externe reset

Na het detecteren van een 0 op de RESET-ingang wordt de PTM gereset:

- Alle latches worden met hun maximale waarde (\$FFFF = 65.535) geladen.
- Alle Control Register bits worden gereset, uitgezonderd bit CR10 (de interne resetbit). CR10 wordt geset. CR10 is bit 0 van Control Register 1, het control register van timer 1.
- Alle tellers worden geladen met de waarden van de latches.
- De uitgangen O1, O2 en O3 worden gereset.
- Alle tellerklokken worden gedisabled.
- Alle bits van het Status Register worden gereset.

Interne reset

Tijdens een interne reset (CR10=1) gebeurt het volgende:

- Alle tellers worden geladen met de waarden van de bijbehorende latches.
- De uitgangen O1, O2 en O3 worden gereset.
- Alle tellerklokken worden gedisabled.
- Alle bits van het Status Register worden gereset.

De latches en de Control Registers worden niet beïnvloed door een interne reset. Zij kunnen tijdens een interne reset zelfs worden geladen met informatie.

APPENDIX F

DMAC (MC6844)

Registerselectie

adres A4A3A2A1A0	geselecteerde register
0 0 0 0 0	Address Register 0 high
0 0 0 0 1	Address Register 0 low
0 0 0 1 0	Byte Count Register 0 high
0 0 0 1 1	Byte Count Register 0 low
0 0 1 0 0	Address Register 1 high
0 0 1 0 1	Address Register 1 low
0 0 1 1 0	Byte Count Register 1 high
0 0 1 1 1	Byte Count Register 1 low
0 1 0 0 0	Address Register 2 high
0 1 0 0 1	Address Register 2 low
0 1 0 1 0	Byte Count Register 2 high
0 1 0 1 1	Byte Count Register 2 low
0 1 1 0 0	Address Register 3 high
0 1 1 0 1	Address Register 3 low
0 1 1 1 0	Byte Count Register 3 high
0 1 1 1 1	Byte Count Register 3 low
1 0 0 0 0	Channel Control Register 0
1 0 0 0 1	Channel Control Register 1
1 0 0 1 0	Channel Control Register 2
1 0 0 1 1	Channel Control Register 3
1 0 1 0 0	Priority Control Register
1 0 1 0 1	Interrupt Control Register
1 0 1 1 0	Data Chain Register

RESET

Als de RESET-ingang gelijk is aan 0, worden alle registers van de DMAC gereset, uitgezonderd de vier Address Registers en de vier Byte Count Registers.

Voor * dient 0, 1, 2, respectievelijk 3 te worden ingevuld.

Channel Control Register

DMA END flag

b7:=1 tijdens transport van laatste byte van blok gegevens
b7:=0 na het lezen van het bijbehorende Channel Control Register

Busy/ready flag

b6:=1 als TxRQ*=1, mits (Byte Count Register *) ongelijk 0 is
b6:=0 tijdens transport van laatste byte van blok gegevens

Address up/down bit

b3=0: Address Register * wordt verhoogd
b3=1: Address Register * wordt verlaagd

b7	b6	b5	b4	b3	b2	b1	b0
DEND	busy	not used		up	MCA	MCB	R/W

Mode Control A bit

b2=0: DMA Request via DRQ2
b2=1: DMA Request via DRQ1

Mode Control B bit

b1=0: single byte mode
b1=1: burst mode

Let op: MCA=MCB=1 is ongedefinieerd!

Read/Write bit

b0=0: schrijven in het geheugen
b0=1: lezen in het geheugen

Priority Control Register

Rotate Control bit

b7=0: fixed priority
b7=1: rotating priority

Request Enable bits

RE*=0: transfer request disabled
RE*=1: transfer request enabled

b7	b6	b5	b4	b3	b2	b1	b0
rotate	not used			RE3	RE2	RE1	RE0

Interrupt Control Register

DMA END/IRQ flag

$$b7 = DEND0 \cdot DIE0 + DEND1 \cdot DIE1 + DEND2 \cdot DIE2 + DEND3 \cdot DIE3$$

DMA END/IRQ Enable bits

DIE*=0: DMA END interrupt disabled

DIE*=1: DMA END interrupt enabled

b7	b6	b5	b4	b3	b2	b1	b0
DEND/IRQ	not used			DIE3	DIE2	DIE1	DIE0

Data Chain Register

2/4 Channel Select bit

b3=0: 2-channel mode

b3=1: 4-channel mode

Data Chain Enable bit

b0=0: transfer request disabled

b0=1: transfer request enabled

b7	b6	b5	b4	b3	b2	b1	b0
not used				2/4	DCB	DCA	DCE

Data Chain Select bits:

MCB MCA selected channel

b2 b1 for data chaining

0	0	0
0	1	1
1	0	2
1	1	undefined

APPENDIX G

IC-PINNING

MC6809

microprocessors

MC6809E

Vss	1*	40	HALT
NMI	2	39	XTAL
IRQ	3	38	EXTAL
FIRQ	4	37	RESET
BS	5	36	MRDY
BA	6	35	Q
Vcc	7	34	E
A0	8	33	DMA/BREQ
A1	9	32	R/W
A2	10	31	D0
A3	11	30	D1
A4	12	29	D2
A5	13	28	D3
A6	14	27	D4
A7	15	26	D5
A8	16	25	D6
A9	17	24	D7
A10	18	23	A15
A11	19	22	A14
A12	20	21	A13

Vss	1*	40	HALT
NMI	2	39	TSC
IRQ	3	38	LIC
FIRQ	4	37	RESET
BS	5	36	AVMA
BA	6	35	Qin
Vcc	7	34	Ein
A0	8	33	BUSY
A1	9	32	R/W
A2	10	31	D0
A3	11	30	D1
A4	12	29	D2
A5	13	28	D3
A6	14	27	D4
A7	15	26	D5
A8	16	25	D6
A9	17	24	D7
A10	18	23	A15
A11	19	22	A14
A12	20	21	A13

MC6821 PIA

Vss	1*	40	CA1
PA0	2	39	CA2
PA1	3	38	IRQA
PA2	4	37	IRQB
PA3	5	36	RS0
PA4	6	35	RS1
PA5	7	34	RESET
PA6	8	33	D0
PA7	9	32	D1
PB0	10	31	D2
PB1	11	30	D3
PB2	12	29	D4
PB3	13	28	D5
PB4	14	27	D6
PB5	15	26	D7
PB6	16	25	E
PB7	17	24	CS1
CB1	18	23	CS2
CB2	19	22	CS0
Vcc	20	21	R/W

MC6844 DMAC

Vss	1*	40	E
CS/TxAKB	2	39	RESET
R/W	3	38	DGRNT
A0	4	37	DRQ1
A1	5	36	DRQ2
A2	6	35	TxAKA
A3	7	34	TxSTB
A4	8	33	IRQ/DEND
A5	9	32	TxRQ0
A6	10	31	TxRQ1
A7	11	30	TxRQ2
A8	12	29	TxRQ3
A9	13	28	D0
A10	14	27	D1
A11	15	26	D2
A12	16	25	D3
A13	17	24	D4
A14	18	23	D5
A15	19	22	D6
Vcc	20	21	D7

MC6840 PTM

Vss	1*	28	C1
G2	2	27	O1
O2	3	26	G1
C2	4	25	D0
G3	5	24	D1
O3	6	23	D2
C3	7	22	D3
RESET	8	21	D4
IRQ	9	20	D5
RS0	10	19	D6
RS1	11	18	D7
RS2	12	17	E
R/W	13	16	CS1
Vcc	14	15	CS0

AM27256 EPROM

Vpp	1*	28	Vcc
A12	2	27	A14
A7	3	26	A13
A6	4	25	A8
A5	5	24	A9
A4	6	23	A11
A3	7	22	OE
A2	8	21	A10
A1	9	20	CE/PGM
A0	10	19	O7
O0	11	18	O6
O1	12	17	O5
O2	13	16	O4
GND	14	15	O3

MC6850 ACIA

Vss	1*	24	CTS
RxDATA	2	23	DCD
RxCLK	3	22	D0
TxCLK	4	21	D1
RTS	5	20	D2
TxDATA	6	19	D3
IRQ	7	18	D4
CS0	8	17	D5
CS2	9	16	D6
CS1	10	15	D7
RS	11	14	E
Vcc	12	13	R/W

MC6828 PIC

CS1	1*	24	Vcc
STRETCH	2	23	INT
CS0	3	22	Z4
IN0	4	21	Z3
IN1	5	20	Z2
IN2	6	19	Z1
IN3	7	18	E
IN4	8	17	R/W
IN5	9	16	A1
IN6	10	15	A2
IN7	11	14	A3
GND	12	13	A4

IMS1420 SRAM

A4	1*	20	Vcc
A5	2	19	A3
A6	3	18	A2
A7	4	17	A1
A8	5	16	A0
A9	6	15	I/O1
A10	7	14	I/O2
A11	8	13	I/O3
E	9	12	I/O4
Vss	10	11	W

INDEX

A

absolute addressing, 69
accestijd, 437
accumulator, 50, 58
achterbord, 336
achtergrondgeheugen, 43
ACIA, 362, 614
actuele parameter, 219
add-instructie, 154
adres, 63
adresbus, 8, 47
adresdecoder, 125
adrespointer, 84
adresregister, 50
adresselectie, 334
adresseren, 64
advanced valid memory
 address, 453
afrekinstructie, 155
afronden, 159
ALU, 49
and-instructie, 135
arithmetic shift, 157, 160
ASCII, 26, 603
assembler, 18, 96, 111, 517
assembler directive, 106
assembly, 18, 97
asynchrone transmissie, 308,
 317, 364
auto-decrement, 77, 198
auto-increment, 77, 198

B

back panel, 336
backward reference, 112
basisadres, 336
baud (rate), 365
BCD, 25
BCD-optelling, 163, 168
belastingscapaciteit, 432
benchmarkprogramma, 435
besteltraject, 250
besturingsbus, 47
bidirectioneel, 47, 317
binair, 22
biphase modulatie, 366
bit, 22
bitgeorganiseerd geheugen, 463
bit rate, 365
bitsynchronisatie, 320
bittest-instructie, 145
bittijd, 320
block transfer DMA, 408

block transfer routine, 131
borrow, 29, 59
bottom-up, 16
bouncing, 429
branch, 89
branch-instructie, 172, 174,
 176, 519
bubble-sort, 187
buffer, 48
buffer gate, 473
burst mode DMA, 408, 489
bus, 7, 47
busarbitration, 447
bus board, 336
bus grant, 440
busmaster, 404, 438
bus request, 440
bus timing, 476
busy, 447
byte, 25

C

calculated jump, 531
carry, 28, 59
channel control register, 416
chip, 6
clear-instructie, 134
commentaar, 104
communicatieprotocol, 324
compare-instructie, 162
compiler, 18
computer, 10
computerarchitectuur, 45
concurrent proces, 419
conditional assembler, 112
condition code, 50, 59
continuous operating mode, 383
controlbus, 8, 47
controller/sequencer, 51
control register, 333, 342,
 367, 369, 380
coprocessor, 435
cross-assembler, 113
CWA1, 299
cycle-stealing DMA, 408

D

daisy-chain, 275
databus, 8, 47
data chaining, 419
data chain register, 419
data direction register, 333
dataframe, 323

datageheugen, 195
 datapointer, 57
 datatransport, 121
 debuggen, 127, 251
 decimaal, 21
 decimal adjust, 163
 decrement-instructie, 155
 deel-instructie, 159
 deelroutine, 242
 denderen, 429
 digit, 21
 direct memory access, 126,
 direct (page) addressing, 70
 direct page register, 58, 524
 308, 401, 441
 disassembler, 113
 diskette, 421
 disk unit, 43
 displacement, 74, 102
 DMA, 126, 401
 DMAC, 403, 619
 DMA-Controller, 403, 411
 DMA-mode, 406
 DMA request, 440
 documentatie, 232, 251
 dode cyclus, 406, 438
 do-while, 12, 15, 128
 DRAM, 467
 drempelspanning, 471
 dummy cycle, 424, 490
 dummy instructie, 153
 dummy read, 350
 duty-cycle, 385
 dynamische geheugentoewijzing,
 513
 dynamische RAM, 467
 dynamisch geheugen, 435

E

EAROM, 45, 463
 echo, 359
 editor, 100
 EEPROM, 45, 463
 effectief adres, 69
 END, 109
 entire-bit, 282
 EPROM, 45, 463
 EQU, 106
 erasable, 45
 erasable PROM, 463
 exchange-instructie, 124
 exclusive-or-instructie, 139
 executiecyclus, 56
 expressie, 102
 extended addressing, 69
 extended indirect addressing, 84

F

fabricage, 251

faculteit, 542
 fast Fourier processor, 435
 fast interrupt, 284
 FCB, 107
 FCC, 108
 FDB, 107
 FDC, 421
 fetch cyclus, 56
 file, 116
 fixed point, 36
 fixed priority, 417
 flag, 50, 59
 flexibele schijf, 421
 floating point, 37
 floating point processor, 436
 floppy disk, 421
 floppy disk controller, 421
 flowchart, 3
 flowdiagram, 3
 following mode, 359
 formatteren, 421
 formele parameter, 219
 forward reference, 112
 framing error, 368
 frequency comparison mode, 387
 full duplex, 358
 functioneel ontwerp, 250
 fysisch adres, 80

G

geheugen, 44
 geheugen indirect addressing, 84
 gemiddelde, 474
 globaal geheugen, 494
 globale variabelen, 494
 global storage, 494

H

half carry, 60, 164
 half duplex, 358
 halt burst mode, 417, 489
 halt steal mode, 417, 488
 halt-toestand, 441
 handshaking, 324, 352, 404, 420
 hardware, 6
 hardware-generatie, 10
 hardware interrupt, 258, 443
 hardware polling, 275
 hardware stack, 198
 hardware vectoring, 260, 270,
 455
 herhaling, 12, 15
 herstarten, 280
 hexadecimaal, 23
 hidden DMA, 409
 high-byte, 195
 hogere programmeertaal, 18
 hold time, 436
 hysteresis, 432

I

IC, 6
 idle state, 363
 if-then-else, 127
 immediate addressing, 66
 implied addressing, 65
 increment-instructie, 153
 indexed addressing, 72
 indexed indirect addressing, 87
 indexregister, 57
 indirect addressing, 82
 inherent addressing, 65
 input, 507
 input/output, 46, 306
 instructie, 6, 44, 62, 116
 instructiecode, 56
 instructiecyclus, 56
 instructiedecoder, 51
 instructieduur, 118
 instructiepointer, 58
 instructieregister, 50
 instructieteller, 50
 integrated circuit, 6
 interface, 209, 507
 interlocked handshake, 326, 360
 interpreter, 18
 interrupt, 198, 255, 341, 382
 interrupt acknowledge, 274, 439
 interrupt controller, 453
 interrupt control register, 418
 interrupt hardware, 270
 interrupt I/O, 307, 313, 339, 402
 interrupt level, 276
 interrupt mask bit, 278
 interrupt masker, 456
 interrupt request, 288, 369
 interrupt service routine, 260
 interruptvector, 260, 272, 295
 inverteerinstructie, 139
 invoer, 46
 invoerparameter, 507
 I/O, 7
 I/O-adres, 525
 I/O-mapped I/O, 126, 307
 I/O-timing, 476-491
 ISR, 260
 iteratie, 12

J

jump-instructie, 172
 jump-table, 530

K

karactersynchronisatie, 322
 keuzestructuur, 12, 127
 keyboard, 41
 klokpuls, 434

L

label, 97, 99
 last-in/first-out, 196, 212
 last instruction cycle, 446
 latch, 377
 lichtkrant, 399
 LIFO, 196
 linkage editor, 209
 load effective address, 165, 523
 loader, 17, 174, 517
 load-instructie, 122
 local storage, 493
 lokaal geheugen, 493
 lokale variabele, 493
 logic shift instructie, 142, 160
 logische instructie, 133
 long branch, 174, 519
 looptijd, 433
 low-byte, 195
 LSB, 22
 LSI, 11
 lus, 12

M

machinecode, 17
 macro, 112, 217
 macro-assembler, 112, 218, 222
 macro body, 218
 macro call, 217
 macrodefinitie, 218
 macro expansion, 219
 macro header, 218
 maintenance, 252
 masker, 135-145
 mask-programmed ROM, 463
 master reset, 369
 memory, 44
 memory map, 457
 memory-mapped I/O, 125, 306
 memory ready, 442
 meta-assembler, 113
 microcomputer, 10
 microprocessor, 6
 missing interrupt, 346
 mnemonic, 18, 97
 modem, 366
 modulair programmeren, 16, 207
 modulair systeem, 336
 modulariteit, 250
 module, 207, 493
 monitor, 41
 monitorprogramma, 46
 MPU, 52
 MPU-mode, 406
 MSB, 22
 multiply-instructie, 158
 multiprocessorsysteem, 445
 multitasking, 257, 532
 multi-user, 257

N

nacalculatie, 251
NAM, 109
Nassi-Shneidermann, 15
negate-instructie, 156
negative flag, 60
nesten, 212, 265
Neumann, Von, 56
nibble, 25
niet-procedurele taal, 19
non-maskable interrupt, 281
non-return-to-zero, 328
non-volatile, 46, 467
normale kansverdeling, 474
NOVRAM, 46
NRZ, 328, 467

O

object program, 18
offset, 74
onderhoud, 252
one-pass assembler, 111
one's complement, 139
one-shot, 376, 459
ongestructureerd, 14
ontwerp, 250
onvoorwaardelijke sprong, 172
opcode, 17, 62, 100, 117
open-collector, 267
open-drain, 267
operand, 56, 62, 101, 117
operatiecode, 17, 56
operating system, 43, 115
opstarten, 280
ORG, 106
or-instructie, 137
orthogonale instructieset, 119
oscillator, 434
output, 46, 507
overflow, 34, 59, 186
overhead, 402, 405
overlappende handshake, 326
overrun, 368

P

packed BCD, 147
PAGE, 110
parallelle computer, 11
parallelle processen, 419
paralleltransport, 7, 308
parameter, 219
parameteroverdracht, 223, 302, 507
pariteit, 26
parity error, 368
periferie, 306, 312
peripheral, 44
peripheral interface adapter, 330

peripheral register, 333
permanent geheugen, 494
permanent storage, 494
PIA, 330, 612
PIC, 453
pinning, 622
planning, 248
pointer, 57, 513, 521
pointerregister, 57
polling, 268
portable, 19
positie-afhankelijk, 175
positie-onafhankelijk, 80, 92, 175, 517
postbyte, 68-88, 117, 125, 200
post DMA routine, 404
post-increment, 77
pre-decrement, 77
printer, 43
prioriteit, 264
priority control register, 417
priority encoder, 276
probleemanalyse, 247
probleemdefinitie, 246
procedurele taal, 18
processor, 6, 44
processorarchitectuur, 49
program counter, 50, 58
program counter relative addressing, 79, 520
program generator, 19
programma, 45, 115
programmable ROM, 463
programmable timer module, 376
programmaflow, 264
programmeergeugen, 195
programmastructuur-diagram, 15
programma-uitvoer, 116
programmed I/O, 307, 309, 402
programmeerfout, 119
programmeermodel, 57
programmeren, 6
projectaanpak, 246
PROM, 45, 463
PSD, 15
pseudo-instructie, 106
PTM, 376, 616
pull, 196, 202
pull-up weerstand, 267, 437
pulsbreedtemeting, 389
pulsed handshake, 324, 349
pulse width comparison mode, 388
push, 197, 200

R

RAN, 45, 466
RAMtestroutine, 232
randapparaat, 44
random access, 45, 466
read-modify-write, 380, 448

read mostly memory. 463, 467
 read only. 45, 367
 read only memory. 463
 read/write. 45, 466
 real time clock. 286
 receive data register. 364
 receive shift register. 364
 recursief. 537
 reentrant. 532
 refresh. 435, 467
 register. 48, 494
 register addressing. 66
 register indirect addressing. 84
 registerselectie. 333, 367, 379, 415
 rekenkundige instructie. 153
 rekenorgaan. 48
 relative addressing. 89, 102
 relocatable. 174, 517
 relocatable assembler. 517
 relocatable loader. 517
 repeat-until. 13, 15, 128
 reset. 332, 369, 379
 reset-instructie. 134
 resident assembler. 113
 restart interrupt. 278
 return code. 227
 return from interrupt. 174, 261, 293
 return from subroutine. 173
 return-to-zero. 328
 RMB. 108
 RMM. 463, 467
 ROM. 45, 463
 rotate-instructie. 142
 rotating priority. 418
 roteerinstructie. 142
 RS232-C. 367
 RWM. 45
 RZ. 328

S

SAM. 45
 schijveneenheid. 43
 schuifinstructie. 141
 sector. 421
 selectie. 12, 15
 self-assembler. 113
 self-refresh. 435
 semafoor. 284, 420, 449
 serial access. 45
 serie-parallelomzetting. 148
 serietransport. 7, 308, 371
 set-instructie. 137
 setup time. 436
 seven-segment display. 395
 shift-instructie. 142
 short branch. 174
 signaalcodering. 320
 sign-and-magnitude. 29
 sign extension. 165
 simplex. 358
 simultaneous DMA. 408
 single byte mode DMA. 408
 single-shot operating mode. 386
 software. 6
 software-generaties. 17
 software interrupt. 174, 258, 289
 software polling. 268
 sorteerroutine. 186
 source program. 18
 spagettistructuur. 14
 SPC. 110
 spoor. 421
 spreiding. 474
 spronginstructie. 172, 176
 SRAM. 467
 stack. 58, 195, 210, 497, 510
 stack mark. 500
 stack overflow. 278, 286
 stack pointer. 58, 196, 523
 stapelgeheugen. 196
 startbit. 322, 363
 statische RAM. 467
 statistische aanpak. 473
 status. 59
 statusregister. 50, 59, 363, 367, 382
 step-by-step mode. 442
 sterverbinding. 7
 stopbit. 322, 363
 store-instructie. 121
 strap. 335
 string. 171
 strobing. 317, 347
 stroomschema. 3
 structogram. 15, 126
 subroutine. 198, 210
 subroutine call. 173, 210
 subtract-instructie. 155
 symbol table. 112, 222
 synchrone transmissie. 308, 315, 364
 synchronisatie. 308, 315, 319, 444, 478
 synchronisatiekarakter. 322
 synchronization acknowledge. 440
 synchronize-instructie. 300
 syntaxfout. 119
 system call. 258, 289
 system stack. 198, 216

T

technisch ontwerp. 250
 tellerinitialisatie. 382
 temporary storage. 494
 terminal. 41
 test-instructie. 161, 176
 three-state. 407

- three-state control stealing, 416, 481
- throughput error, 408
- tijdelijk geheugen, 494
- tijdintervalmeting, 388
- tijdintervalvergelijking, 387
- tijdmeting, 392
- time-multiplexed, 487
- time-out, 329, 353, 375, 384, 390
- timer, 376
- time-sharing, 257, 496, 532
- time slice, 496
- timing, 431
- timingberekening, 465, 470
- toetsenbord, 41
- top-down, 16
- track, 421
- transfer-instructie, 124
- transistor-transistor-logic, 432
- transmit data register, 362
- transmit shift register, 363
- trap, 258
- tri-state, 407
- TTL, 432
- two-pass assembler, 112
- two's complement, 30

U

- UART, 312, 362
- uitvoer, 46
- uitvoerparameter, 507
- unidirectioneel, 47

- unsigned, 29
- USART, 312
- user stack, 198

V

- V24, 366
- valid memory address, 485
- vectoradresvertaling, 272, 454
- vectorinformatie, 260
- vectoring, 270, 455
- vergelijkinstructie, 161
- vermenigvuldiginstructie, 157
- video display, 41
- VLSI, 11
- volgorde, 12, 15
- voorwaardelijke sprong, 90, 176

W

- wachttijdroutine, 152
- watchdog, 330
- wave measurement mode, 376
- wave synthesis mode, 376
- werkgeheugen, 44
- wired-AND, 266, 460
- woordgeorganiseerd geheugen, 463
- worst case, 472
- write-only, 367

Z

- zero flag, 60
- zoekroutine, 171, 237

ANTWOORDEN

§ 1.9

1

doof groene licht
ontsteek gele licht
wacht 3 seconden
doof gele licht
ontsteek rode licht

2 goed:

	/
er is verkeer	
doof rode licht	
ontsteek groene licht	

fout:

er is verkeer	
ja	nee
doof rode licht	/
ontsteek groene licht	

Het licht staat op rood. Er moet gewacht worden, totdat er verkeer is. In de herhalingslus hoeft niets gedaan te worden. Let wel dat er een herhaling nodig is, waarin getest wordt of er verkeer is.

Indien u een keuzestructuur toepast, blijft het licht rood als er op het moment van de test geen verkeer is. Dan wordt het programma vervolgd, zonder dat het licht groen wordt. Indien er iets later wel verkeer is, wordt de betreffende test niet meer uitgevoerd, want we zijn dan al verder in het programma. Het licht blijft dan rood.

3

herhaal telkens	
laat de bel stil zijn	
	/
er nadert een trein	
	luidt de bel
de trein is voorbij en er nadert geen tweede	

\$ 2.14

- 1 Let op de volgorde van de bewerkingen!
 Van decimaal naar hexadecimaal omrekenen en omgekeerd gaat vrij eenvoudig en snel.
 Van decimaal naar BCD en omgekeerd, evenals van hexadecimaal naar binair en omgekeerd is uiterst eenvoudig, omdat hier elk cijfer apart kan worden vertaald.
 - a. $725 = 2 \cdot 16^2 + 13 \cdot 16^1 + 5 \cdot 16^0 =$
 $= \$2D5 = \%0010\ 1101\ 0101 = \text{BCD } 0111\ 0010\ 0101$
 - b. $\$1AF = 1 \cdot 256 + 10 \cdot 16 + 15 = 431 = \text{BCD } 0100\ 0011\ 0001 =$
 $= \%0001\ 1010\ 1111$
 - c. $\%1001\ 1101\ 0010 = \$9D2 = 9 \cdot 256 + 13 \cdot 16 + 2 = 2514 =$
 $= \text{BCD } 0010\ 0101\ 0001\ 0100$
 - d. $\text{BCD } 1000\ 0111\ 0101 = 875 = 3 \cdot 256 + 6 \cdot 16 + 11 = \$36B =$
 $= \%0011\ 0110\ 1011$
- 2 De boodschap "Zend 3 bytes." luidt in ASCII:
 $\$5A\ \$65\ \$6E\ \$64\ \$20\ \$33\ \$20\ \$62\ \$79\ \$74\ \$65\ \$73\ \$2E$
- 3 ($\%01001\ 101$ 1-onthouden)
 $\%0110\ 1101$
 $\%0100\ 1101$
 $\%1011\ 1010$ +
- 4 ($\$010$ 1-onthouden)
 $\$2C9$
 $\$791$
 $\$A5A$ +
- 5 $\$9A$
 $\$3C$ +
 $\$D6$
 De carry is 0, aangezien de som past in 8 bits.
- 6 ($\%1110\ 100$ 1-lenen)
 $\%1100\ 1011$
 $\%0111\ 0101$ -
 $\%0101\ 0110$ -
- 7 ($\$1$ 1-lenen)
 $\$73$
 $\$37$ -
 $\$3C$
- 8 Bij het berekenen van $351 - 718$ zal de waarde van de borrow 1 zijn, immers het aftrekken gaat niet zonder te lenen van het niet-bestaande naastliggende hogere cijfer (een duizendtal).
- 9 $+94 = +(5 \cdot 16 + 14) = +(\$5E) = +(\%0101\ 1110) =$
 $= \%0101\ 1110$ (sign-and-magnitude)
 $-57 = -(3 \cdot 16 + 9) = -(\$39) = -(\%0011\ 1001) =$
 $= \%1011\ 1001$ (sign-and-magnitude)
- 10 $+27 = +(1 \cdot 16 + 11) = +(\$1B) = +(\%0001\ 1011) =$
 $= \$0001\ 1011$ (two's complement)
 $-111 = -(6 \cdot 16 + 15) = -(\$6F) = -(\%0110\ 1111)$
 We vinden de two's complement notatie van een negatief

- getal, door de absolute waarde van dat getal af te trekken van 0, ofwel door de absolute waarde te inverteren en er vervolgens 1 bij op te tellen.
Inverteren van %0110 1111 levert %1001 0000. Tellen we hierbij 1 op, dan vinden we %1001 0001.
Dit is dus de two's complement notatie van -11.
- 11 Het kleinste unsigned getal van 4 bits is %0000 = 0; het grootste getal is %1111 = 15.
Het kleinste sign-and-magnitude getal van 4 bits is %1111 = -(%111) = -7; het grootste getal is %0111 = +(%111) = +7.
Het kleinste two's complement getal van 4 bits is %1000 = -8 (immers de inverse van %1000 is %0111 en dit vermeerderd met 1 levert %1000 = 8); het grootste getal is %0111 = +7.
- 12 %1101.111 = 8 + 4 + 0 + 1 + 0.5 + 0.25 + 0.125 = 13.875
\$50.A = 5.16 + 0 + 10. (1/16) = 80.625
- 13 fixed point notatie: %xxxx xxx.x xxxx = %xxx xxxx,xxxx x +36.75 = %010 0100.1100 0 = %0100 1001 1000
-11.0625 = -(%000 1011.0001 0)
We vinden de two's complement notatie door dit getal te inverteren en er vervolgens 1 bij op te tellen. Merk hierbij op dat er een minstwaardige bit wordt opgeteld, niet een echte eenheid!
Inverteren geeft %111 0100.1110 1. Tellen we hierbij 1 op dan vinden we %111 0100.1111 0 en dus is
-11.75 = %111 0100.1111 0 = %1110 1001 1110
- 14 Floating point getal in sign-and-magnitude notatie:
%EEEE Et.fff ffff
De exponent bestaat uit 5 bits. Deze wordt gevolgd door de fractie, die uit 7 bits bestaat.
A = %1011 0011 0100, dus exponent = %10110 = -6 en fractie = %0.110100 = +0.8125
Dit betekent dat A = $2^{-6} \cdot 0.8125 = +0.0126953125$
B = %0100 1101 1001, dus exponent = %01001 = +9 en fractie = %1.011001 = -0.390625
Dit betekent dat B = $2^9 \cdot (-0.390625) = -200$
- 15 Floating point getal in two's complement notatie:
%EEEE Et.fff ffff
De exponent bestaat uit 5 bits. Deze wordt gevolgd door de fractie, die uit 7 bits bestaat.
X = %0001 1100 0101, dus exponent = %00011 = +3 en fractie = %1.000101
Deze fractie is negatief. De absolute waarde vinden we door te inverteren en 1 erbij te tellen: %0.111010 + 1 = %0.111011.
Dus de fractie is -%0.111011 = -0.921875
Dit betekent dat X = $2^3 \cdot (-0.921875) = -7.375$
Y = %1110 1111 0000, dus exponent = %11101 en fractie = %1.110000
De exponent is negatief. We vinden de absolute waarde door te inverteren en 1 erbij te tellen: %00010 + 1 = %00011.
Dus de exponent is -%00011 = -3
De fractie is ook negatief. We vinden de absolute waarde door te inverteren en 1 erbij te tellen: %0.001111 + 1 = %0.010000.

Dus de fractie bedraagt $-0,010000 = -0,25$
 Dit betekent dat $Y = 2^2 \cdot (-0,25) = -0,03125$

\$ 3.8

- 1 De MC68000 is een microprocessor met een 16-bits externe databus, die echter inwendig kan rekenen met 32-bits getallen. Hiertoe bezit hij 32-bits rekenregisters en een 32-bits interne databus.
- 2 (CC) = EFHINZVC = %0000 0000
 - a.

%1011 1100	(negatief getal)
<u>%1110 0100</u>	+ (negatief getal)
%(1)1010 0000	+ (negatief getal)

Er is een carry en een half-carry, dus C=H=1.
 De uitkomst is negatief (dus N=1) en ongelijk 0 (dus Z=0).
 De som van twee negatieve getallen levert hier inderdaad een negatief getal op, zodat er geen overflow optreedt (V=0).
 Conclusie: (CC) = %0010 1001
 - b.

%0110 1100	(positief getal)
<u>%1010 1000</u>	+ (negatief getal)
%(1)0001 0100	+ (positief getal)

Wederom treden er een carry en een half-carry op, dus C=H=1.
 Er is geen overflow (dus V=0), immers de som van een positief getal en een negatief getal past altijd in het register. Er treedt nooit een tekenfout op in de uitkomst, immers de uitkomst kan zowel positief als negatief zijn.
 Daar de uitkomst ongelijk 0 en positief is, geldt Z=N=0.
 Conclusie: (CC) = %0010 0001
 - c.

%0101 0101	(positief getal)
<u>%0110 1001</u>	+ (positief getal)
%(0)1011 1110	+ (negatief getal)

Er treedt hier geen carry noch een half-carry op, zodat C=H=0.
 Er treedt een tekenfout op, immers de som van twee positieve getallen kan niet negatief zijn, dus hebben we te maken met overflow (V=1).
 Daar de uitkomst negatief is en ongelijk 0, geldt N=1 en Z=0.
 Conclusie: (CC) = %0000 1010
 - d.

%0001 1110	(positief getal)
<u>%0101 1011</u>	+ (positief getal)
%(0)0111 1001	+ (positief getal)

We hebben hier geen carry, maar wel een half-carry, dus C=0 en H=1.
 Er treedt geen tekenfout op, zodat V=0.
 De uitkomst is positief en ongelijk 0, zodat N=Z=0.
 Conclusie: (CC) = %0010 0000
 - e.

%1101 0001	(negatief getal)
<u>%1001 0111</u>	+ (negatief getal)
%(1)0110 1000	+ (positief getal)

Hier geldt C=1 en H=0.
 Er treedt een tekenfout op, zodat V=1. De som van twee negatieve getallen kan immers niet positief zijn.
 De uitkomst is positief en ongelijk 0, dus N=Z=0.

- Conclusie: (CC) = %0000 0011
- f. %0111 0010 (positief getal)
 %1000 1001 - (negatief getal)
 %(1)1110 1001 (negatief getal)
- We hebben 1 moeten lenen van de (niet-bestaande) bit 8, zodat er een borrow is, ofwel C=1.
 De half-carry bit verandert bij aftrekken niet! Derhalve blijft H=0.
 Trekken we een negatief getal af van een positief getal, dan is de uitkomst altijd positief. Dit is hier niet het geval, zodat er een overflow is ontstaan (V=1).
 De uitkomst is negatief en ongelijk 0, zodat N=1 en Z=0.
 Conclusie: (CC) = %0000 1011
- g. %1010 1100 (negatief getal)
 %0100 0011 - (positief getal)
 %(0)0110 1001 (positief getal)
- Hier is C=0 en H verandert niet bij aftrekken, dus H=0.
 Trekken we een positief getal af van een negatief getal, dan moet de uitkomst negatief zijn. Dit is hier niet het geval, hetgeen betekent dat er overflow is opgetreden, dus V=1.
 Een positieve uitkomst ongelijk 0 betekent N=Z=0.
 Conclusie: (CC) = %0000 0010
- h. \$DB (negatief getal)
 \$C9 + (negatief getal)
 \$(1)A4 + (negatief getal)
- Hier zijn de carry en de half-carry beiden 1, dus C=H=1.
 Er treedt geen overflow op, immers er is geen tekenfout, zodat V=0.
 De uitkomst is negatief en ongelijk 0, dus N=1 en Z=0.
 Conclusie: (CC) = %0010 1001
- i. \$17 (positief getal)
 \$AF - (negatief getal)
 \$(1)68 (positief getal)
- Hier moet er 1 geleend worden, zodat C=1. Bij aftrekken verandert de half-carry niet, zodat H=0 blijft.
 Trekken we een negatief getal af van een positief getal, dan moet de uitkomst positief zijn. Dit is hier inderdaad het geval, zodat we geen overflow hebben (V=0).
 Een positieve uitkomst ongelijk 0 betekent N=Z=0.
 Conclusie: (CC) = %0000 0001

\$ 4.11

- | | | | |
|------|------|----------|---|
| 1 a. | STA | <\$F2 | direct page addressing (vanwege <) |
| b. | LDB | #\$44 | immediate addressing (vanwege #) |
| c. | DECA | | inherent addressing (immers geen operand nodig) |
| d. | ADDD | \$59F4 | extended (absolute) addressing (2-bytes operand) |
| e. | SUBD | #\$AB67 | immediate addressing (vanwege #) |
| f. | TFR | CC.A | immediate addressing (hier wordt de postbyte gebruikt om aan te duiden om welke registers het gaat) |
| g. | STD | [\$19DC] | extended indirect addressing (vanwege de rechte haken) |
| h. | JMP | <\$C7 | direct page addressing (vanwege <) |

- i. LDA [\$20FF] extended indirect addressing
(vanwege de rechte haken)
- 2 De opcodes voor de volgende instructies zijn:
- a. LDA = \$86 immediate
= \$96 direct
= \$B6 extended (absolute)
= \$A6 extended indirect
- b. STD = \$DD direct
= \$FD extended (absolute)
= \$ED indexed
- c. DECB = \$5A inherent
- d. TFR = \$1F immediate
- 3 instructie effectieve adres
- a. LDA ,X (X) = \$A48C
- b. LDA -9,X $-9+(X) = -9+\$A48C = \$A483$
- c. LDA 20,X $20+(X) = \$0014+\$A48C = \$A4A0$
- d. LDA \$7A,X $\$7A+(X) = \$007A+\$A48C = \$A506$
- e. LDA <\$90,X $\$90+(X) = \$FF90+\$A48C = \$A41C$
- f. LDA \$7104,X $\$7104+(X) = \$7104+\$A48C = \1590
- g. LDA \$F300,X $\$F300+(X) = \$F300+\$A48C = \$978C$
- 4 postbyte indexed addressing mode indexregister offset
- a. 1110 1000 8-bits constant offset S
- b. 1000 0100 0-bits constant offset X 0
- c. 0011 0111 5-bits constant offset Y %10111
- d. 1110 1001 16-bits constant offset S
- e. 0100 0110 5-bits constant offset U %00110
- f. 1000 1001 16-bits constant offset X
- 5 instructie adresseringsmethode effectieve adres
- a. LDA #\$F4 immediate \$3A43 (op dit adres bevindt zich immers de data, \$F4)
- b. STD \$EC09 extended (absolute) \$EC09
- c. CMPX <\$A6.Y 8-bits constant offset $\$A6+(Y) = \$FFA6+\$2064 = \$200A$
- d. LDB ,X+ auto-increment indexed (X) = \$2F58 (voordat X wordt verhoogd)
- e. ADDD ,--S auto-decrement indexed (S) = \$7FFF, immers (S) is met 2 verminderd voordat de instructie wordt uitgevoerd
- f. BRA \$49 relative $\$49+(PC) = \$0049+\$3A44 = \$3A8D$:
bedenk dat (PC) = \$3A44 na het ophalen van de 2-bytes instructie
- g. JMP \$282D,PC program counter relative $\$282D+(PC) = \$282D+\$3A46 = \6273 :
bedenk dat (PC) = \$3A46 na het ophalen van de 4-bytes instructie
- h. SUBA B,U acc. B offset indexed (B)+(U) = \$0020+\$5C47 = \$5C67; bedenk dat (D) = (A)(B), zodat (B)=\$20

- i. LBCC \$8FA2 (long branch) relative (PC) = \$3A46, indien de sprong niet wordt uitgevoerd (dus als C=1)
 $\$8FA2 + (PC) = \$8FA2 + 3A46 = \$C9E8$
indien de sprong wel wordt uitgevoerd (dus als C=0)
- 6 a. LDA [,-Y] (Y) = \$AA48-1 = \$AA47 = adrepointer, dus het effectieve adres <ea> is <eaH> = (\$AA47) = \$47 en <eaL> = (\$AA48) = \$48, zodat <ea> = \$4748
- b. STD [Y,++] (Y) = \$AA48 = adrepointer, zodat <eaH> = (\$AA48) = \$48 en <eaL> = (\$AA49) = \$49, dus <ea> = \$4849
- c. ADDB [\$25.Y] $\$25 + (Y) = \$0025 + \$AA48 = \$AA6D$ = adrepointer, zodat <eaH> = (\$AA6D) = \$6D en <eaL> = (\$AA6E) = \$6E, zodat <ea> = \$6D6E
- d. SUBA [-22.Y] $-22 + (Y) = -\$0016 + \$AA48 = \$AA32$ = adrepointer, zodat <eaH> = (\$AA32) = \$32 en <eaL> = (\$AA33) = \$33, zodat <ea> = \$3233

\$ 5.7

- 1 Duidt de opcode op indexed (of indirect) addressing, dan is de volgende byte een postbyte, die meedeelt om welk type indexed of indirect addressing het gaat.
- | | |
|---------------------|-----------------|
| a. \$4F | CLRA |
| b. \$B6 00 A8 | LDA >\$A8 |
| c. \$86 33 | LDA #\$33 |
| d. \$D6 E4 | LDB <\$E4 |
| e. \$E6 9F 57 BC | LDB [\$57BC] |
| f. \$A7 89 37 21 | STA \$3721.X |
| g. \$10 AF FD E3 09 | STY [\$E309.PC] |
| h. \$AF A3 | STX ,--Y |
- 2 FCB zorgt ervoor dat elk getal als dubbel byte in het geheugen wordt opgeborgen. FCC bergt de ASCII-waarden van karakters op. RMB reserveert een aantal bytes in het geheugen. Er worden hierbij een aantal locaties overgeslagen. EQU kent een adres toe aan een label. Dit legt verder geen beslag op de geheugenruimte. ORG definieert de geheugenlocatie, waar moet worden begonnen.

adres	inhoud	adres	inhoud
\$2000	\$A5	\$2017	\$19
\$2001	\$6E	\$2018	\$EC
\$2002	\$00	\$2019	\$11
\$2003	\$33	\$201A	\$5A
\$2004	\$8D	\$201B	\$06
\$2005	\$E5	\$3500	\$00
\$2006	\$34	\$3501	\$0C
\$2007	\$80	\$3502	\$03
\$2008	\$65	\$3503	\$C4
\$2009	\$72	\$3504	\$ED

\$200A	\$72	\$3505	\$00
\$200B	\$6F	\$3506	\$40
\$200C	\$72	\$3507	\$10

§ 6.8

1	instructie	opcode	postbyte	ruimte in bytes	duur in μ s
a.	CLRA	\$4F		1	2
b.	CMPPX #5A3D	\$8C		3	4
c.	STD <\$DE	\$DD		2	5
d.	LDA \$6089	\$B6		3	5
e.	CMPPY .X++	\$108C	\$81	3	10
f.	ADDD -400,U	\$E3	\$C9	4	10
g.	LDB [\$6C]	\$E6	\$9F	4	9
h.	LBCC \$A7	\$1024		4	5 of 6
i.	LEAX \$22,PC	\$30	\$8C	3	5
	of:	\$30	\$8D 4 9		
j.	LBRA BEGIN	\$16		3	5

2	instructie	postbyte
a.	TFR A,DP	\$8B
b.	TFR DP,A	\$B8
c.	TFR X,U	\$13
d.	TFR U,PC	\$35
e.	EXG Y,S	\$24
f.	EXG A,B	\$89
g.	EXG S,Y	\$42
h.	EXG X,D	\$10

- 3 Na het wegschrijven van \$FF (in voorbeeld 6.11) wordt de pointer met 1 verhoogd. Daarna wordt gekeken of de EPROM-buffer geheel gevuld is (dus t/m adres \$17FF). Als de pointerinhoud gelijk is aan \$17FF, is deze locatie nog niet gevuld. Er moet derhalve nog één keer worden teruggesprongen. Daarom moet de pointerinhoud worden vergeleken met het volgende adres, dus met \$1800.

- 4 Object program van de eerste oplossing van voorbeeld 6.12:

adres	instructie	duur in μ s
\$3000	\$8E 10 00	3
\$3003	\$10 8E 4C 00	4
\$3007	\$A6 A0	6
\$3009	\$A7 80	6
\$300B	\$8C 52 62	4
\$300E	\$26 F7	3
\$3010	\$3F	19

Dit programma omvat 17 bytes.

Het programma kost $26 \mu\text{s} + 19 \mu\text{s}$ voor elke byte die verplaatst wordt. Voor 1000 bytes is dit dus $19.026 \mu\text{s}$.

- 5 Tweede oplossing van voorbeeld 6.12:

instructie	aantal bytes	aantal μ s
LDX #EBUF	3	3
LDA DISPL,X	4	8
STA .X+	2	6
CMPPX #LAATST+1	3	4
BNE NEXT	2	3
SWI	1	19

Dit programma bestaat uit 15 bytes.
De executietijd bedraagt echter 22 μ s + 21 μ s voor elke te verplaatsen byte. Voor 1000 bytes is dit derhalve 21.022 μ s.

6

initialiseer de sourcepointer	
initialiseer de destinationpointer	
byteteller := aantal te verplaatsen bytes	
	verplaats een byte en verhoog de pointers
	verlaag de byteteller
byteteller = 0	

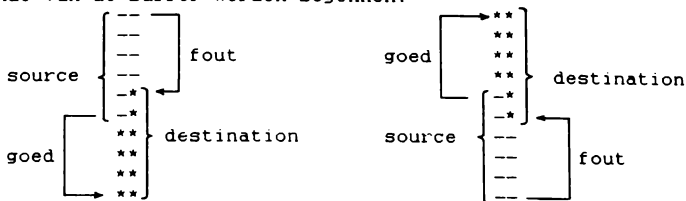
```

10          NAM      MOVE BLOCK OF BYTES FROM FIRST
20 *Progr.: J.G. Rouland
30 *Revisie 0      Datum: 25-05-1986
40 *
50 *Er worden 2500 bytes verplaatst, te beginnen met de
60 *eerste. De sourcebuffer begint op adres $5A00, de
70 *destinationbuffer op $5300.
80 *
90 *
100         ORG      $8000
110 SOURCE   EQU      $5A00      beginadres van sourcebuffer
120 DESTIN   EQU      $5300      beginadres van
125 *                               destinationbuffer
130 AANTAL   EQU      2500      aantal te verplaatsen bytes
140 *
150 MOVBEQ   LDY      #SOURCE      initialiseer sourcepointer
160          LDU      #DESTIN      initialiseer
165 *                               destinationpointer
170          LDX      #AANTAL      initialiseer de byteteller
180 NEXT     LDA      ,Y+          verplaats een byte en
190          STA      ,U+          verhoog de pointers
200          LEAX     -1,X         byteteller:=byteteller-1
210          BNE      NEXT        spring als niet 2500 bytes
215 *                               gehad
220          SWI                               stop
230          END      MOVBEQ

```

In dit geval kan niet begonnen worden met het verplaatsen van de laatste byte. De sourcebuffer en de destinationbuffer overlappen elkaar, terwijl de data naar een lager adres wordt overgebracht. Het aantal bytes bedraagt 2500=\$9C4, zodat de destinationbuffer in het geheugengebied ligt met de adressen \$5300 t/m \$5CC3. De hoogste adressen liggen dus in de sourcebuffer. Zouden we met de laatste byte beginnen, dan zouden er gegevens in de sourcebuffer vernield worden, voordat deze gegevens zijn verplaatst. Bedenk hierbij dat beide buffers zich in hetzelfde geheugen bevinden. Alleen de adressen zijn (ten dele) verschillend. Uit de figuren blijkt, dat bij overlapping van source- en destinationbuffer niet met de laatste byte mag worden begonnen, indien de transfer naar een lager adres plaatsvindt.

Vindt de overdracht naar een hoger adres plaats, dan mag niet met de eerste byte worden begonnen. Als er geen overlap is, mag zowel aan het begin als aan het einde van de buffer worden begonnen.



niet met laatste
byte beginnen

niet met eerste
byte beginnen

7

initialiseer de sourcepointer		
initialiseer de destinationpointer		
byteteller := aantal te verplaatsen bytes		
byteteller ongelijk 0		
<table> <tr> <td>verlaag pointers en verplaats een byte</td></tr> <tr> <td>verlaag de byteteller</td></tr> </table>	verlaag pointers en verplaats een byte	verlaag de byteteller
verlaag pointers en verplaats een byte		
verlaag de byteteller		

```

10      NAM      MOVE BLOCK OF BYTES FROM LAST
20 *Progr.: J.G. Rouland
30 *Revisie 0    Datum: 25-05-1986
40 *
50 *Er worden bytes verplaatst. te beginnen met de
60 *laatste. Het beginadres van de sourcebuffer staat op
70 *de adressen SOURCE en SOURCE+1. Het beginadres van de
80 *destinationbuffer staat op DESTIN en DESTIN+1. Het
90 *aantal bytes staat op LENGTH en LENGTH+1
100 *
110 *
120      ORG      $2000
130 MOVEND LDD     SOURCE      de sourcepointer wijst 1
140      ADDD     LENGTH      locatie verder dan
150      TFR      D,Y          hoogste adres van buffer
160      LDD      DESTIN      de destinationpointer wijst
170      ADDD     LENGTH      1 plaats verder dan
180      TFR      D,U          hoogste adres van buffer
190      LDX      LENGTH      initialiseer de byteteller
200 TEST   BEQ     EINDE      spring als alle bytes
205 *      verplaatst
210      LDA      ,-Y          verlaag de pointers en
220      STA      ,-U          verplaats een byte
230      LEAX     -1,X         byteteller:=byteteller-1
240      BRA      TEST
250 EINDE   SWI      stop
260      END      MOVEND

```

Er wordt hier begonnen met het testen of de byteteller 0 is, teneinde te voorkomen dat er bytes worden verplaatst, terwijl (LENGTH)(LENGTH+1) gelijk aan 0 is. Om het programma MOVEND te kunnen vertalen en uitvoeren, moeten de labels SOURCE, DESTIN en LENGTH worden gedefinieerd.

- 8 Met één bittest-instructie kan worden nagegaan of meerdere bits tegelijk 0 zijn. Moeten we testen of geldt A2=A5=A7=0, dan kan dit met de instructie

BITA #%10100010

De Z-bit van het condition code register wordt alleen dan geset, als de and-functie van (A) met het masker de waarde 0 oplevert. Dit is alleen het geval, indien de genoemde bits alle drie gelijk aan 0 zijn.

- 9 Het is niet mogelijk om met één bittest-instructie na te gaan of meerdere bits tegelijk 1 zijn. De Z-bit wordt immers niet alleen gereset als deze bits allen 1 zijn, maar ook als slechts een van die bits 1 is.

Om na te gaan of meerdere bits 1 zijn, kunnen we alle bits eerst inverteren en daarna testen of de betreffende bits allen 0 zijn. Inverteren kan eenvoudig geschieden met behulp van de instructie COMA. Na de bittest- en de spronginstructie kan de waarde van A weer worden hersteld met de instructie COMA.

Testen of geldt dat A4=A6=1 gaat op deze wijze als volgt:

COMA

BITA #%01010000

BEQ ENEN

spring als voor inverteren
geldt A4=A6=1

NIET1 COMA

--

--

ENEN COMA

herstel (A)

Indien de inhoud van A niet meer nodig is, laten we het herstellen van (A) natuurlijk achterwege.

10

initialiseer de pointers	
byteteller := aantal BCD-bytes	
byteteller ongelijk 0	
	lees BCD-byte
	vertaal linker nibble naar ASCII en berg het resultaat op
	vertaal rechter nibble naar ASCII en berg het resultaat op
	verlaag byteteller

10 NAM BCD-ASCII-CONVERSION

20 *Progr.: J.G. Rouland

30 *Revisie 0 Datum: 25-05-1986

40 *

50 *Packed BCD-getallen worden omgezet in ASCII-karakters

60 *en opgeborgen in een buffer; te beginnen op adres


```

70 *$7800. De linker nibble (MSN, most significant nibble)
80 *wordt telkens het eerst vertaald en opgeborgen.
90 *De BCD-buffer begint op adres $7000, waarbij de eerste
100 *locatie het aantal bytes bevat.
110 *
120 *
130          ORG      $3500
140 BCDBUF   EQU      $7000          beginadres van BCD-buffer
150 ASCBUF   EQU      $7800          beginadres van ASCII-buffer
160 *
170 BCDASC   LDY      #BCDBUF        initialiseer BCD-pointer
180          LDU      #ASCBUF        initialiseer ASCII-pointer
190          LDB      ,Y+            byteteller := aantal BCD-
200          CLRA                      bytes en de BCD-pointer
210          TFR      D,X            wijst naar de eerste BCD-
215 *                                     byte
220 TEST     CMPX      #0            spring als alle bytes
225          BEQ      EINDE          behandeld zijn
230          LDA      ,Y+            laad een BCD-byte
240          TFR      A,B
250          LSRA                      zet MSN om in ASCII
260          LSRA
270          LSRA
280          LSRA
290          ORA      #$30
300          STA      ,U+            berg het resultaat op
310          ANDB     #$0F          zet LSN om in ASCII
320          ORB      #$30
330          STB      ,U+            berg het resultaat op
340          LEAX     -1,X          byteteller:=byteteller-1
350          BRA      TEST
360 EINDE     SWI
370          END      BCDASC        stop

```

Omdat we de accumulatoren A en B later in het programma nodig hebben, gebruiken we X als teller. hoewel het aantal slechts een 1-byte getal is. Dit aantal kan niet rechtstreeks worden geladen in X, omdat in dat geval 2 bytes in X worden gezet, namelijk (\$7000)(\$7001). Dit is niet correct, immers aantal=(\$7000).

```

11 10          NAM      SERIETRANSPORT NAAR PRINTER
20  *Progr.: J.G. Rouland
30  *Revisie 0      Datum: 25-05-1986
40  *
50  *Een aantal 16-bits woorden worden in serie. te
60  *beginnen met de MSB, via bit 2 van OUTPUT naar een
70  *printer verzonden. Telkens nadat een bit gereed staat.
80  *wordt KLOK gedurende ongeveer 1 ms 1 en daarna
90  *gedurende ongeveer 1 ms 0. KLOK is bit 5 van OUTPUT.
100 *OUTPUT heeft adres $FF01.
110 *
115 *Het aantal te verzenden woorden bevindt zich op de
120 *geheugenplaatsen $3000 en $3001. Daarachter volgen de
125 *woorden.
130 *
140          ORG      $2000
150 SRCBUF   EQU      $3000          beginadres van sourcebuffer
160 OUTPUT   EQU      $FF01          adres van OUTPUT-register
170 DATEEN   EQU      %000000100    masker om data te zetten
180 DATNUL   EQU      %11111011     masker om data te resetten

```

initialiseer de woordpointer
KLOK:=0
woordteller := aantal te verzenden woorden
woordteller ongelijk 0
bitteller:=16
lees woord en verhoog woordpointer
schuif MSB naar printerinterface
KLOK:=1 en wacht ongeveer 1 ms
KLOK:=0 en wacht ongeveer 1 ms
verlaag bitteller met 1
16 bits verzonden (bitteller = 0)
verlaag woordteller

190	KLKEEN	EQU	%00100000	masker om KLOK te setten
200	KLKNUL	EQU	%11011111	masker om KLOK te resetten
210	REDAB	RMB	2	ruimte voor redden van (A)
215	*			en (B)
220	*			
230	SEROUT	LDU	#SRCBUF	initialiseer woordpointer
240		CLR	OUTPUT	KLOK:=0
250		LDX	,U++	woordteller:=aantal woorden
260	*			en woordpointer wijst naar
265	*			eerste woord
270	TEST	CMFX	#0	spring als alle woorden
275		BEQ	EINDE	verzonden
280		LDY	#16	bitteller:=16
290		LDD	,U++	lees woord en verhoog
295	*			woordpointer
300	NOGBIT	LSLB		schuif bits naar links en
310		ROLA		zet MSB in C-bit
320		STD	REDAB	red (A) en (B)
330		LDA	OUTPUT	maak databit gelijk
340		BCS	EEN	aan C
350	NUL	ANDA	#DATNUL	reset databit
360		BRA	ZEND	
370	EEN	ORA	#DATEEN	set data
380	ZEND	STA	OUTPUT	verzend databit
385		ORA	#KLKEEN	set KLOK
390		STA	OUTPUT	verzend data en klok
400		LDB	#200	wacht ongeveer 1 ms
405	*			(200·5=1000 µs)
410	WACHT	DECB		duurt 2 µs
420		BNE	WACHT	duurt 3 µs
430		ANDA	#KLKNUL	reset KLOK
440		STA	OUTPUT	
450		LDB	#200	wacht ongeveer 1 ms
460	WAIT	DECB		
470		BNE	WAIT	

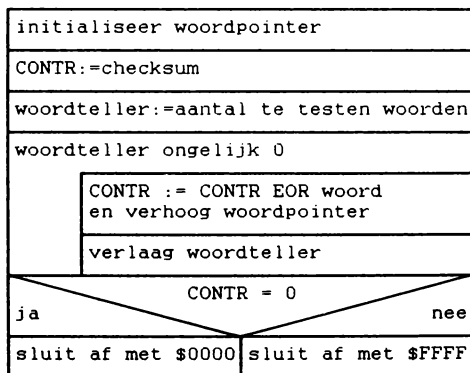
480	LDD	REDAB	herstel (A) en (B)
490	LEAY	-1.Y	verlaag bitteller
500	BNE	NOGBIT	spring als geen 16 bits
505 *			verzonden
510	LEAX	-1.X	verlaag woordteller
520	BRA	TEST	
530 EINDE	SWI		stop
540	END	SEROUT	

Merk op dat U in regel 250 met 2 wordt verhoogd. Dit moet omdat er een 2-bytes getal wordt geladen, zodat het volgende getal twee plaatsen verder staat.

Vergeet niet de instructie STA OUTPUT (regel 390 en 440) toe te passen. Pas bij deze instructie wordt de data, die naar de printer gaat, gewijzigd!

Realiseer u dat de registers U en S niet kunnen worden gebruikt als teller, omdat bij de instructies LEAU -1,U en LEAS -1,S de Z-flag niet wordt aangepast! Dit gebeurt wel bij LEAX en LEAY.

- 12 De checksum moet gelijk zijn aan het resultaat van de exclusive-or-functie van alle woorden. Nemen we de exclusive-or van de checksum en dat resultaat, dan moet dit \$0000 opleveren bij het gelijk zijn aan elkaar, immers (0 EOR 0)=0 en (1 EOR 1)=0. Dit betekent dat we voor deze test de exclusive-or-functie kunnen nemen van de checksum en alle woorden. Het uiteindelijke resultaat moet dan 0 zijn.



```

10          NAM      CONTROLEER CHECKSUM
20 *Progr.: J.G. Rouland
30 *Revisie 0      Datum 25-05-1986
40 *
50 *Een checksum over een aantal woorden (van 16 bits)
60 *bevindt zich op de locaties $5000 en $5001. Het aantal
70 *woorden staat op $5002 en $5003. Daarna volgen de
80 *woorden zelf.
90 *Indien de checksum juist is, wordt de reeks woorden
100 *afgesloten met $0000. Klopt de checksum niet, dan
110 *wordt er afgesloten met $FFFF.
120 *
130 *
```

140		ORG	\$3000	
150	BUFFER	EQU	\$5000	beginadres van de buffer
160	*			
170	CONCHE	LDU	#BUFFER	initialiseer woordpointer
180		LDD	,U++	CONTR := checksum; verhoog pointer
185	*			
190		LDX	,U++	laad aantal en verhoog woordpointer
195	*			
200	TEST	CMPX	#0	spring als alle woorden
205		BEQ	CHECK	getest
210		EORA	,U+	CONTR := CONTR EOR woord en
220		EORB	,U+	verhoog pointer
230		LEAX	-1,X	woordteller:=woordteller-1
240		BRA	TEST	
250	CHECK	CMPD	#0	spring als
260		BEQ	OK	CONTR=0
270	FOUT	LDD	#\$FFFF	sluit af met \$FFFF
280	OK	STD	,U	(foutmelding) of met \$0000 (ok-melding)
285	*			
290		SWI		stop
300		END	CONCHE	

- 13 Bij negatieve getallen wordt door de instructies

```

      ASRA
      ADCA #0
naar beneden afgerond. Dit wil zeggen dat -0,5 wordt
afgerond naar 0. Het resultaat van het schuiven wordt met
+1 verhoogd als de carry gelijk is aan 1. Een negatief
getal verhogen met 1 betekent dat het getal minder negatief
wordt. Dit impliceert dat er bij negatieve getallen naar
beneden wordt afgerond.
Zo wordt bijvoorbeeld %1111 1100 (= -4) na delen door twee
gelijk aan %1111 1110 (= -2). Hierbij wordt C=0, zodat
afronden geen verandering teweeg brengt.
Wordt echter %1111 1101 (= -3) door twee gedeeld, dan
levert dit %1111 1110 (= -2) op en de carry wordt 1. Na
afronden (carry bijtellen) krijgen we %1111 1111 (= -1),
zodat inderdaad naar beneden is afgerond.

```

- 14 Indien n groter dan 127 kan zijn (en dus uit twee bytes bestaat), dan moet er in het programma ADDBCD van voorbeeld 6.48 het volgende veranderd, respectievelijk toegevoegd worden:

125	REDA	RMB	1	ruimte voor redden van (A)
127	CARRY	RMB	1	ruimte voor carry
140	ADDBCD	LDD	DATA	
170		LEAX	D,X	
180		LEAX	2,X	n omvat twee bytes!
190		LEAY	D,X	
210		LEAU	D,Y	
230		CLR	CARRY	
235	REKEN	STA	REDA	red high-byte van
236	*			byteteller
240		LDA	, -X	
245		ADDA	CARRY	verhoog (A) met CARRY
271		BCC	CNUL	
272	CEEN	CLR	CARRY	set

273	INC	CARRY	CARRY
274	BRA	DECR	
275	CNUL	CLR	CARRY
276	DECR	LDA	REDA
277	*		reset CARRY
280	SUBD	#1	herstel high-byte van byteteller
			verlaag de byteteller
300	LDB	CARRY	

De instructie SUBD #1 (regel 280) vernielt de carry, die bij de BCD-optelling is ontstaan. Vandaar dat deze carry in een aparte geheugenlocatie CARRY wordt gered (regel 271 t/m 275).

- 15 In regel 170 van het programma ZOEKTK van voorbeeld 6.49 wordt de stringpointer met 1 verhoogd, zodat deze vast naar het volgende karakter wijst. Blijkt in regel 180 en 190 dat het gezochte teken gevonden is, dan moet de stringpointer in regel 200 met 1 worden verlaagd, teneinde naar het gezochte teken in de string te wijzen.
- 16 Tellen we maximaal 255 8-bits unsigned getallen op, dan past de som in twee bytes.

initialiseer de bytetpointer		
teller := aantal op te tellen bytes		
som:=0		
teller ongelijk 0		
<table border="1"> <tr> <td>somL:=somL+getal en wijs naar de volgende byte</td></tr> <tr> <td>verlaag teller</td></tr> </table>	somL:=somL+getal en wijs naar de volgende byte	verlaag teller
somL:=somL+getal en wijs naar de volgende byte		
verlaag teller		
berg som op		

```

10          NAM      ADD 1-BYTE UNSIGNED NUMBERS
20 *Progr.: J.G. Rouland
30 *Revisie 0      Datum: 25-05-1986
40 *
50 *Er worden n 1-byte unsigned getallen opgeteld. N staat
60 *op locatie $7700. Dan volgen de getallen. Achter deze
70 *getallen wordt de 2-bytes som geplaatst.
80 *
90 *
100         ORG      $5000
110 N       EQU      $7700          locatie van aantal bytes
120 *
130 ADDUNS  LDY      #N+1          Y wijs naar eerste op te
140 *                      tellen byte
150         LDA      N
160 *                      laad aantal op te tellen
170         LDX      #0            bytes
180 TEST    TSTA
190         BEQ      EINDE         som:=0
200 *                      spring als alle bytes zijn
                                opgeteld

```

210	LDBA	.Y+	tel byte erbij en verhoog
220	ABX		bytepointer
230	DECA		verlaag de byteteller
240	BRA	TEST	
250	EINDE	STX	SOM
260		SWI	stop
270	END	ADDUNS	

- 17 Een BCD-getal van vier nibbles kan hoogstens 9999 zijn. Om dit getal in gewogen binaire code weer te geven, hebben we 14 bits nodig. De binaire waarde past dus in twee bytes. Eerst splitsen we het BCD-getal in vier afzonderlijke cijfers N3, N2, N1 en N0. Hieruit kan de binaire code berekend worden met behulp van de formule:

$$\text{BINAIR} = 1000 \cdot N3 + 100 \cdot N2 + 10 \cdot N1 + N0 =$$

$$= ((N3 \cdot 10 + N2) \cdot 10 + N1) \cdot 10 + N0$$

$$\begin{array}{c} \underbrace{\hspace{10em}} \\ \leq 99 \\ \underbrace{\hspace{10em}} \\ \leq 990 \\ \underbrace{\hspace{10em}} \\ \leq 9999 \end{array}$$

splits het BCD-getal in 4 cijfers		
(B) := 10 · N3	≤	90
(B) := (B) + N2	≤	99
(D) := 10 · (B)	≤	990
(D) := (D) + N1	≤	999
(D) := 10 · (D)	≤	9990
(D) := (D) + N0	≤	9999

Een probleem treedt op bij het berekenen van $10 \times (D)$, omdat (D) een 2-bytes getal is. De instructie MUL laat alleen toe om 1-byte getallen te vermenigvuldigen. Dit betekent dat de vermenigvuldiging gesplitst moet worden:

$$\begin{array}{rcl} (D) = & \text{AAAAAAAABBBBBBBB} & \leq 99 \\ 10 = & \begin{array}{r} \text{00001010} \\ \hline \text{00000000} \cdot 10 \cdot (B) \cdot \\ \hline \text{00000000} \cdot 10 \cdot (A) \cdot \\ \hline \text{SSSSSSSSSSSSSSSS} \end{array} & \begin{array}{l} \times \\ \leq 90 \\ + \leq 90 \\ \leq 990 \end{array} \end{array}$$

Het bijbehorende programma ziet er nu als volgt uit:

```

5      NAM          4 CIJFERS BCD NAAR BINAIR
10 *Progr.: J.G. Rouland
20 *Revisie 0      Datum: 25-05-1985
30 *
40 *Het BCD-getal in (D) wordt vertaald naar de binaire
50 *waarde, die eveneens in (D) komt te staan. De BCD-
60 *waarde wordt hierbij overschreven.
70 *
80 *
90      ORG          $6000

```

100	REDD	RMB	2	ruimte om (D) te redden
110	N0	RMB	1	ruimte voor nibble 0
120	N1	RMB	1	ruimte voor nibble 1
130	N2	RMB	1	ruimte voor nibble 2
140	N3	RMB	1	ruimte voor nibble 3
150	*			
160	BCD4	STD	REDD	red (D)
170		TFR	A,B	neem de meestwaardige byte
180		LSRA		splits BCD-byte in 2
190		LSRA		cijfers
200		LSRA		
210		LSRA		
220		STA	N3	
230		ANDB	#\$0F	
240		STB	N2	
250		LDA	REDD+1	lees de minstwaardige byte
260		TFR	A,B	
270		LSRA		splits de BCD-byte in 2
280		LSRA		cijfers
290		LSRA		
300		LSRA		
310		STA	N1	
320		ANDB	#\$0F	
330		STB	N0	
340		LDB	N3	(B) := 10·N3
350		LDA	#10	
360		MUL		
370		ADDB	N2	(B) := (B) + N2
380		LDA	#10	(D) := 10·(B)
390		MUL		
400		ADDB	N1	(D) := (D) + N1
410		ADCA	#0	
420		STD	REDD	red (D)
430		LDA	#10	vermenigvuldig low-byte
440		MUL		met 10
450		STB	REDD+1	berg 10·<low-byte> op
460		LDA	#10	vermenigvuldig high-byte
470		LDB	REDD	met 10
480		MUL		
490		TFR	B,A	voeg low-byte en high-byte
500		LDB	REDD+1	van 10·(D) samen
510		ADDB	N0	(D) := (D) + N0
520		ADCA	#0	
530		SWI		stop
540		END	BCD4	

Let op dat (A) opnieuw 10 moet worden gemaakt in de regels 380, 430 en 460, omdat (A)=0 wordt ten gevolge van de voorafgaande MUL-instructies!

- 18 De absolute waarde van het grootste two's complement getal van 8 bits is 128 (= 2^7). Tellen we 2^{16} van deze getallen op, dan bedraagt de som ten hoogste $(2^{16}) \cdot (2^7) = 2^{23}$. Deze waarde past in 24 bits, ofwel in 3 bytes. Dit betekent dat bij het optellen van 2^{16} two's complement getallen van 8 bits, nooit overflow kan ontstaan, indien er voor de som 3 bytes worden gereserveerd.

- 19 In het programma BUBBLE-SORT van voorbeeld 6.62 mag de instructie BLE op regel 290 niet worden vervangen door Gelijke getallen mogen niet worden verwisseld, omdat programma in dat geval nooit eindigt, indien er gelijke getallen in de reeks voorkomen. De reeks zou dan immers niet zonder verwisselingen doorlopen worden, zodat RE telkens wordt gereset.
- 20 Wil BUBBLE-SORT werken voor unsigned getallen, dan moet instructie BLE van regel 290 vervangen worden door BL
- 21 Om 2-bytes two's complement getallen te sorteren, moet volgende regels van het programma BUBBLE van voorbeeld veranderd worden:

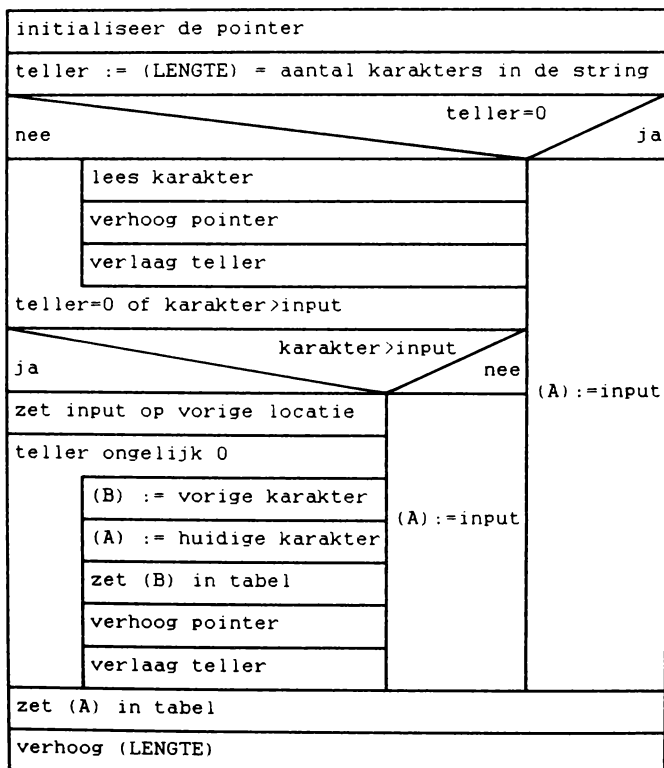
```

270 NEXT      LDU      ,X++
280           CMPU     ,X

300 WISSEL    LDY      ,X
310           STY      -2.X
320           STU      .X

```

22



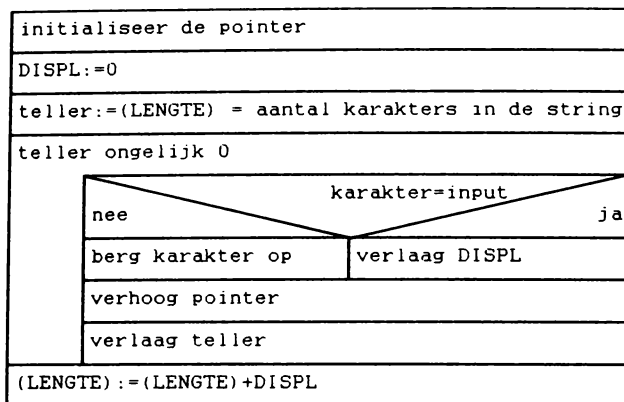

```

10      NAM      INSERT ELEMENT
20 *Progr.: J.G. Rouland
30 *Revisie 0   Datum: 25-05-1986
40 *
50 *In een (in oplopende volgorde) gesorteerde tabel wordt
60 *een ASCII-karakter tussengevoegd. De tabel begint op
70 *adres TABEL en het aantal karakters van de tabel staat
80 *op adres LENGTE. Het tussen te voegen karakter wordt
90 *geladen uit een register met als adres INPUT.
100 *
110 *
120      ORG      $2000
130 TELLER RMB    1           ruimte voor teller
140 *
150 INSERT LDU    #TABEL      initialiseer pointer
160      LDA     LENGTE      teller := lengte van string
170      STA     TELLER
180      BEQ     LEESIN      spring als lengte = 0
190 ZOEK   LDA     .U+        lees karakter en verhoog
195 *      pointer
200      DEC     TELLER      springs als alle karakters
210      BEQ     TSTKAR      gehad
220      CMPA    INPUT
230      BLS     ZOEK        spring als karakter < input
240 TSTKAR CMPA    INPUT      spring als
250      BLS     LAADIN      karakter < input
260 VOEGIN LDB     INPUT      lees input en zet deze in
270      STB     -1,U        tabel op vorige locatie
280      TST     TELLER
290 TEST   BEQ     LASTIN    spring als alle karakters
295 *      gelezen
300      TFR     A,B         red het vorige karakter
310      LDA     .U         lees huidige karakter
320      STB     .U+        zet vorige in tabel,
325 *      verhoog pointer
330      DEC     TELLER
340      BRA     TEST
350 LAADIN LDA     INPUT
360      BRA     LASTIN
370 LEESIN LDA     INPUT
380 LASTIN STA     .U        berg laatste karakter op
390      INC     LENGTE
400      SWI     stop
341      END     INSERT

```

De regels 350 en 360 kunnen vervallen als LAADIN (regel 250) wordt vervangen door LEESIN. In dat geval wordt de structuur van het programma echter verlaten.

- 23 Een bepaald karakter op elke locatie uit een tabel verwijderen, gaat eenvoudig door alle karakters vanaf het begin te lezen en ze telkens DISPL locaties naar voren in de tabel op te bergen. Begin met DISPL=0. Schrijf elk gelezen karakter dat gelijk is aan het te verwijderen karakter, niet terug en verlaag in dat geval DISPL met 1. Op deze wijze wordt een bepaald karakter overal uit de tabel verwijderd en schuiven de volgende karakters naar voren, zodanig dat de overblijvende karakters weer netjes aansluiten.
Merk op dat de tabel bij deze procedure niet gesorteerd hoeft te zijn.



DISPL levert een offset, die negatief moet zijn, omdat het volgende karakter moet worden opgeborgen op een lager adres. Bij indexed addressing wordt de offset opgeteld bij de inhoud van het indexregister.

```

10      NAM      DELETE ELEMENT
20 *Progr.: J.G. Rouland
30 *Revisie 0    Datum: 25-05-1986
40 *
50 *Alle karakters die gelijk zijn aan INPUT, worden uit
60 *de tabel verwijderd. De volgende karakters schuiven
70 *hierbij naar voren en de lengte van de tabel wordt
80 *aangepast. Deze bevindt zich op adres LENGTE. De tabel
90 *begint op adres TABEL.
100 *
110      ORG      $3000
120 TELLER RMB    1           ruimte voor teller
130 *
140 DELETE LDU    #TABEL     initialiseer de pointer
150      CLRA
160      LDB      LENGTE     DISPL:=0
170      STB      TELLER     teller:=(LENGTE)
180 TEST   TST     TELLER
185      BEQ      KLAAR      spring als alle karakters
190      LDB      ,U         gelezen
200      CMPB     INPUT
210      BEQ      WEG         spring als karakter=input
220 BERGOP  STB     A,U       berg het karakter op
230      BRA      INCPOI
240 WEG     DECA
250 INCPOI  LEAU    1,U       verlaag DISPL
260      DEC      TELLER     verhoog pointer
270      BRA      TEST
280 KLAAR   ADDA    LENGTE    (LENGTE):=(LENGTE)+DISPL
290      STA      LENGTE
300      SWI
310      END      DELETE
stop

```

§ 7.7

- 1 a. PSHS A.CC.DP.X.U heeft als postbyte %0101 1011
b. PULU X.Y.D.CC heeft als postbyte %0011 0111

- 2 a. PSHS # \$CA =
PSHS # %1100 1010 =
PSHS PC.U.DP,A

adres stackinhoud

\$4E84 \$AB↖
\$4E85 \$01 |
\$4E86 \$51 |
\$4E87 \$F0 |
\$4E88 \$20 |
\$4E89 \$80 |
\$4E8A \$..↖S

(S) := \$4E8A-6=\$4E84

- b. PSHU # \$75 =
PSHU # %0111 0101 =
PSHU S.Y.X.B,CC

adres stackinhoud

\$51E8 \$83↖
\$51E9 \$CD |
\$51EA \$55 |
\$51EB \$CC |
\$51EC \$EF |
\$51ED \$27 |
\$51EE \$4E |
\$51EF \$8A |
\$51F0 \$..↖U

(U) := \$51F0-8=\$51E8

- 3 PSHU D.PC.S.DP levert:
adres stackinhoud
(A) \$444E \$D1
(B) \$444F \$D0
(DP) \$4450 \$A8
(SH) \$4451 \$EE
(SL) \$4452 \$FF
(PCH) \$4453 \$20
(PCL) \$4454 \$79
\$4455 \$..

Dus (D) := \$D0D0 (X) := \$0011
(DP) := \$A8 (Y) := \$A8EE
(CC) := \$D1 (U) := \$4454
(PC) := \$8079 (S) := \$FF80

- PULU S.A.CC,Y levert:
(CC)
(A)
(YH)
(YL)
(SH)
(SL)

§ 8.9

- 1 Een subroutine komt in het objectprogramma slechts één keer voor. Derhalve zijn de instructies van een subroutine niet wijzigbaar. De subroutine kan meerdere keren worden aangeroepen. Hierbij verandert er in de subroutine niets. Wel kunnen bij elke aanroep de registerinhouden of de inhouden van bepaalde locaties in het datageheugen anders zijn.
Een macro daarentegen wordt ingelast in het objectprogramma op de plaats van de macro call. De macro wordt even zoveel keren ingelast als er macro calls zijn. Dit schept de mogelijkheid om kleine wijzigingen aan te brengen in de instructies van de macro, tijdens het creëren van het objectprogramma.

- 2 Met behulp van een subroutine kan geen wachttijd van exact n ms worden gemaakt, indien n een variabele is. Dit komt doordat de wachttijd is opgebouwd uit een vast deel T_v (het laden van de gewenste duur, de sprong naar de subroutine en de sprong terug) en een variabel deel T_v (het q keer doorlopen van de wachtlus).
 Als de wachtlus T_l ms bedraagt, is $T_v = q \cdot T_l$.
 Er geldt dan: wachttijd = $T_v + q \cdot T_l$
 Hierin kan q worden gevarieerd om zodoende de gewenste vertraging te krijgen. Daar de wachtlus alleen een geheel aantal malen kan worden doorlopen, is q een geheel getal. Dit betekent dat de wachttijd niet exact gelijk kan zijn aan n , voor elke willekeurige waarde van n .
 Wel kan de wachttijd buiten de subroutine om exact worden aangevuld tot de waarde van n . Hiertoe mag de wachttijd van de subroutine niet groter zijn dan n ms.

- 3 Zonder macro's en subroutines ziet de subroutine ADD4 van voorbeeld 8.7 er als volgt uit:
- ```

10 *****
20 * Subroutine ADD4 *
30 *****
40 *
50 *Progr.: J.G. Rouland
60 *Revisie 1 Datum: 28-05-1986
70 *
80 *Revisie 1 gebruikt geen macro, maar een programmalus.
90 *
100 *ADD4 berekent de unsigned en de two's complement som
110 *van de getallen op GETAL1, GETAL2, GETAL3 en GETAL4 en
120 *zet deze op de locaties SOMUNS en SOMUNS+1.
130 *respectievelijk SOMTWO en SOMTWO+1.
140 *
150 *De registers blijven ongewijzigd.
160 *
170 GETAL1 RMB 4 ruimte voor op te tellen
175 * bytes
180 SOMUNS RMB 2 ruimte voor unsigned som
190 SOMTWO RMB 2 ruimte voor two's
195 * complement som
200 *
210 ADD4 PSHS CC,D,X,U,Y red registers
220 LDU #GETAL1 pointer wijst naar GETAL1
230 LDY #4 teller:=4
240 CLRA
250 CLRB
260 STD SOMTWO reset SOMTWO
270 TFR D,X reset unsigned som
280 NEXT LDB ,U+ laad getal en wijs naar
285 * volgende getal
290 ABX verhoog unsigned som met
295 * getal
300 SEX verhoog
310 ADD SOMTWO SOMTWO met
320 STD SOMTWO getal
330 LEAY -1,Y verlaag teller en spring
340 BNE NEXT als geen 4 getallen
345 * opgeteld
350 STX SOMUNS berg de unsigned som op
360 PULS CC,D,X,U,Y herstel registers
370 RTS naar aanroepende programma

```

- 4 De test om na te gaan of de pagina deel uitmaakt van het programmeergeheugen, zal iets ingewikkelder worden. We mogen nu alleen de RAM-test uitvoeren op pagina's met een lager nummer dan PROBEG en op die met een hoger nummer dan PROEND. PROBEG en PROEND zijn de nummers van de eerste en laatste pagina van het programmeergeheugen. De volgende wijzigingen en aanvullingen zijn nu nodig in het programma RAMMAP van voorbeeld 8.8:

```

570 PROBEG RMB 1 nummer van eerste pagina
575 * van programma
580 PROEND RMB 1 nummer van laatste pagina
585 * van programma

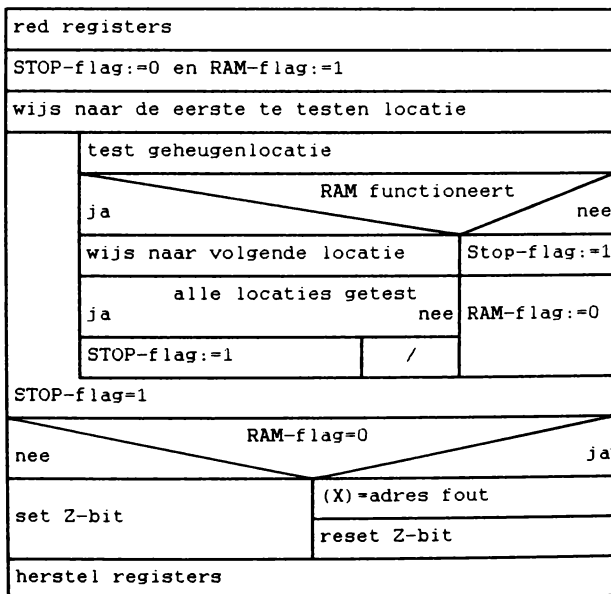
490 STA RAMBEG
491 LDD #EINDE berg nummer van laatste
492 STA PROEND pagina van programma-
493 * geheugen op in PROEND

540 NXTPAG CMPA PROBEG spring als paginanummer
550 BLO RAMTST lager is dan nummer van
551 * eerste programmapagina
552 CMPA PROEND spring als paginanummer <
553 BLS INCRDP nummer van laatste
554 * programmapagina
560 RAMTST TEST $AA:$00

800 EINDE SWI

```

5



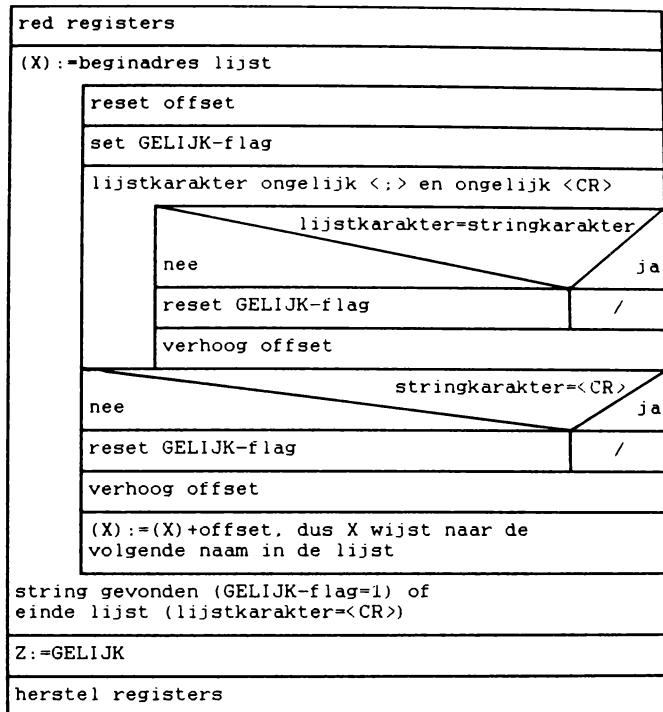
Om te testen of een bepaalde pagina van de RAM in orde is, schrijven we op elke locatie beurtelings \$AA en \$55. We controleren of deze informatie daadwerkelijk in de RAM wordt opgeslagen.

Gebruiken we een offset om de locatie binnen de pagina aan te wijzen, dan hebben we de gehele pagina getest als de offset 0 is geworden. Bedenk dat na 255=\$FF de waarde 0 volgt.

```

10 *****
20 * Subroutine RAMPAG *
30 *****
40 *
50 *Progr.: J.G. Rouland
60 *Revisie 0 Datum: 28-05-1986
70 *
80 *Er wordt getest of elke locatie van de geheugenpagina
90 *(DP) RAM bevat.
100 *Is dit zo, dan wordt Z=1 gemaakt.
110 *Zo niet, dan wordt Z=0 en (X) wordt gelijk aan het
120 *adres van de eerste locatie die geen RAM bevat.
130 *
140 *Alleen (CC) en (X) veranderen.
150 *
160 ZEEN EQU %00000100 masker om Z te setten
170 ZNUL EQU %11111011 masker om Z te resetten
180 STOP RMB 1 ruimte voor STOP-flag
190 RAMBIT RMB 1 ruimte voor RAM-flag
200 *
210 RAMPAG PSHS A,B red registers
220 CLR STOP
230 LDA #1
240 STA RAMBIT
250 CLRB
260 TFR DP,A wijs naar eerste
270 TFR D,X locatie
280 NEXT LDA #$AA test de geheugenlocatie
290 STA B,X
300 CMPA B,X
310 BNE FOUT
320 LDA #$55
330 STA B,X
340 CMPA B,X
350 BNE FOUT
360 INCB
370 BNE TSTEND wijs naar volgende locatie
380 * spring als niet alle
390 INC STOP locaties getest
400 BRA TSTEND STOP-flag:=1
410 FOUT INC STOP
420 CLR RAMBIT
430 TSTEND TST STOP
440 BEQ NEXT
450 TST RAMBIT
460 BEQ NORAM spring bij leesfout
470 OK ORCC #ZEEN geen fout gevonden, set Z
480 BRA HERST
490 NORAM ABX
500 * (X):=adres eerste
510 ANDCC #ZNUL foutlocatie
520 HERST PULS A,B reset Z
530 RTS herstel registers
 naar aanroepende programma

```



Het einde van een lijstnaam moet samenvallen met het einde van de gezochte string. Anders zijn lijstnaam en gezochte string niet gelijk aan elkaar en moet de GELIJK-flag worden gereset (regel 410).

```

10 *****
20 * Subroutine SEARCH *
30 *****
40 *
50 *Progr.: J.G. Rouland
60 *Revisie 0 Datum: 28-05-1986
70 *
80 *Er wordt gezocht naar het voorkomen van een string in
90 *een lijst. Komt de string voor, dan wordt Z=1 en (X)
100 *krijgt het adres van de gezochte string in de lijst.
110 *Zo niet, dan wordt Z=0.
120 *
130 *Y bevat het beginadres van de te zoeken string, U
140 *bevat het beginadres van de lijst
150 *
160 *Alleen (CC) en (X) veranderen.
170 *
180 *
190 GELIJK RMB 1

```





```

100 *en de rest in B.
110 *Indien (B) gelijk aan 0 is. wordt (A)=0 gemaakt.
120 *
130 *Alleen (A) en (B) veranderen.
140 *
150 QUOT RMB 1 ruimte voor quotiënt
160 DELER RMB 1 ruimte voor deler
170 *
180 DEELUN PSHS CC red CC
190 CLR QUOT quotiënt:=0
200 STB DELER
210 BEQ FOUT spring als deler=0
220 TEST CMPA DELER
230 BLO GEREED spring als (A)<deler
240 SUBA DELER
250 INC QUOT
260 BRA TEST
270 GEREED TFR A,B (B):=rest
280 LDA QUOT (A):=quotiënt
290 BRA EINDE
300 FOUT CLRA reset quotiënt
310 EINDE PULS CC herstel CC
320 RTS naar aanroepende programma

```

- 8 Om bij de subroutine FDIVU van voorbeeld 8.11 een 16-bits quotiënt te krijgen met de binaire komma in het midden, moet er nog 8 keer extra worden gedeeld. Deze laatste 8 keer moet er evenwel telkens een 0 worden ingeschoven. Dit is vrij eenvoudig te realiseren door de instructies 380 t/m 460 in een aparte subroutine DEEL8 op te nemen. De wijzigingen en aanvullingen zien er dan als volgt uit:

```

305 QUOT RMB 2 ruimte voor quotiënt

380 BSR DEEL8
390 STA QUOT berg hele deel van quotiënt
395 *
400 CLRA op
405 * zorg voor het
 binnenschuiven van 0
410 BSR DEEL8
420 STA QUOT+1 berg fractie van quotiënt
425 * op

```

Het quotiënt komt nu te staan op de geheugenplaatsen QUOT en QUOT+1. De regels 430 t/m 460 vervallen. De subroutine DEEL8 ziet er als volgt uit:

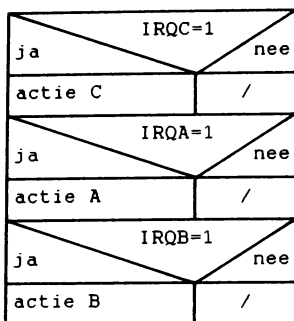
```

500 DEEL8 LDX #8
510 NEXT LSLA
520 ROLB
530 CMPB DELER
540 BLO DECR
550 SUBB DELER
560 INCA
570 DECR LEAX -1,X
580 BNE NEXT
590 RTS

```

## § 9.10

1



|   |     |        |     |          |                            |
|---|-----|--------|-----|----------|----------------------------|
| 2 | 10  | RESISR | LDX | \$\$4000 | initialiseer de            |
|   | 20  |        | STX | \$\$FFC  | NMI-interruptvector        |
|   | 30  |        | LDX | \$\$4A00 | initialiseer de            |
|   | 40  |        | STX | \$\$FFFA | SWI-interruptvector        |
|   | 50  |        | LDX | \$\$4DF0 | initialiseer de            |
|   | 60  |        | STX | \$\$FF8  | IRQ-interruptvector        |
|   | 70  |        | LDA | \$\$40   | blokkeer FIRQ en           |
|   | 80  |        | TFR | A.CC     | enable IRQ                 |
|   | 90  |        | LDS | \$\$A7A8 | initialiseer stack pointer |
|   | 100 |        | JMP | \$2000   | start procesbesturings-    |
|   | 110 | *      |     |          | programma                  |

## § 10.16

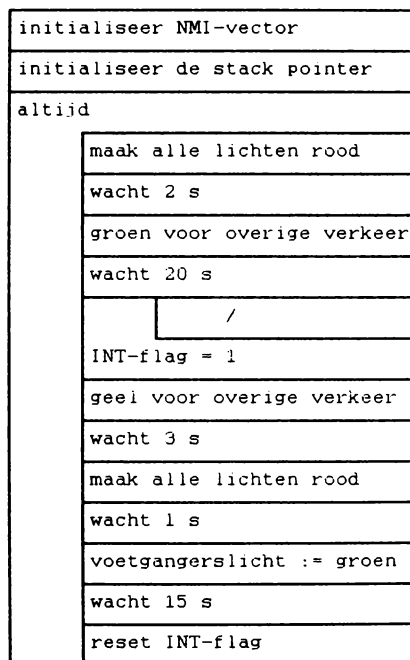
|   |     |                                                    |      |        |                                 |
|---|-----|----------------------------------------------------|------|--------|---------------------------------|
| 1 | 100 | *Subroutine STUUR                                  |      |        |                                 |
|   | 110 | *Deze subroutine verzorgt de volgende invoer-,     |      |        |                                 |
|   | 120 | *verwerkings- en uitvoeracties:                    |      |        |                                 |
|   | 130 | *(PA(i+1)) := (PA(i)) voor i=0,2,4,6 en            |      |        |                                 |
|   | 140 | *PB7 t/m PB4 krijgt de two's complement waarde van |      |        |                                 |
|   | 150 | *het sign-and-magnitude getal in PB3 t/m PB0.      |      |        |                                 |
|   | 160 | *                                                  |      |        |                                 |
|   | 170 | STUUR                                              | LDA  | PRA    | lees PAi                        |
|   | 180 |                                                    | COMA |        | inverteer alle bits             |
|   | 190 |                                                    | LSLA |        | oneven bits := even bits        |
|   | 200 |                                                    | STA  | PRA    | zet oneven bits op PAi; de even |
|   | 210 | *                                                  |      |        | PAi-lijnen veranderen niet;     |
|   | 220 | *                                                  |      |        | ze zijn als input ingesteld     |
|   | 230 |                                                    | LDA  | PRB    | lees sign-and-magnitude getal   |
|   | 240 |                                                    | LSRA |        | zet dit (4-bits) getal          |
|   | 250 |                                                    | LSRA |        | in de linker nibble             |
|   | 260 |                                                    | LSRA |        |                                 |
|   | 270 |                                                    | LSRA |        |                                 |
|   | 280 |                                                    | BMI  | POS    | spring als getal positief       |
|   | 290 | NEG                                                | AND  | \$\$7F | maak getal positief en bepaal   |
|   | 300 |                                                    | NEGA |        | de two's complement waarde      |
|   | 310 | POS                                                | STA  | PRB    | zet resultaat op PB7 t/m PB4    |
|   | 320 |                                                    | RTS  |        |                                 |

```

350 *Subroutine INIPIA
360 *Deze subroutine zorgt voor de juiste instelling van de
370 *PIA ten behoeve van de subroutine STUUR.
380 *
390 INIPIA CLR CRA adresseer DDRA
400 CLR CRB en DDRB
410 LDA 01010101 oneven lijnen output, even
420 STA DDRA lijnen van FA als input
430 LDA 11110000 PB7 t/m PB4 als output,
440 STA DDRB PB3 t/m PB0 als input
450 LDA 4 adresseer
460 STA CRA PRA en
470 STA CRB PRB
480 RTS

```

- 2 In de NMI-interrupt service routine wordt de flag INT geset om aan te geven dat er voetgangers willen oversteken.



```

10 NAM OVER
20 *
30 *Regeling voor voetgangersoversteekplaats.
40 *Normaal zijn de voetgangerslichten rood.
50 *Door op de knop te drukken worden ze 15 s groen.
60 *Daarna worden ze weer rood en krijgt het overige
70 *verkeer tenminste 20 s groen licht.
80 *
90 ORG $A000

```

|     |        |     |            |                            |
|-----|--------|-----|------------|----------------------------|
| 100 | NMIVEC | EQU | \$FFFC     |                            |
| 110 | LICHT  | EQU | \$FF01     | adres van lichtbesturing   |
| 120 | ALROOD | EQU | %10000010  |                            |
| 130 | RIJDEN | EQU | %00100010  | groen voor overige verkeer |
| 140 | GEEL   | EQU | %01000010  | geel voor overige verkeer  |
| 150 | LOPEN  | EQU | %10000001  | groen voor voetgangers     |
| 160 | INT    | RMB | 1          | ruimte voor INT-flag       |
| 170 | *      |     |            |                            |
| 180 | OVER   | LDX | #VOET      | initialiseer de            |
| 190 |        | STX | NMIVEC     | NMI-vector                 |
| 200 |        | LDS | #STACK+100 | initialiseer stack pointer |
| 210 | *      |     |            | enable de NMI-interrupts   |
| 220 | ALSTOP | LDA | #ALROOD    |                            |
| 230 |        | STA | LICHT      | alle lichten rood          |
| 240 |        | LDB | #2         |                            |
| 250 |        | BSR | WACHT      | wacht 2 s                  |
| 260 |        | LDA | #RIJDEN    |                            |
| 270 |        | STA | LICHT      | groen voor overige verkeer |
| 280 |        | LDB | #20        |                            |
| 290 |        | BSR | WACHT      | wacht 20 s                 |
| 300 | TEST   | TST | INT        | wacht tot INT=1 (tot een   |
| 310 |        | BEQ | TEST       | voetganger wil             |
| 315 | *      |     |            | oversteken                 |
| 320 |        | LDA | #GEEL      |                            |
| 330 |        | STA | LICHT      | geel voor overige verkeer  |
| 340 |        | LDB | #3         |                            |
| 350 |        | BSR | WACHT      | wacht 3 s                  |
| 360 |        | LDA | #ALROOD    |                            |
| 370 |        | STA | LICHT      | alle lichten rood          |
| 380 |        | LDB | #1         |                            |
| 390 |        | BSR | WACHT      | wacht 1 s                  |
| 400 |        | LDA | #LOPEN     |                            |
| 410 |        | STA | LICHT      | groen voor voetgangers     |
| 420 |        | LDB | #15        |                            |
| 430 |        | BSR | WACHT      | wacht 15 s                 |
| 440 |        | CLR | INT        | reset INT-flag             |
| 450 |        | BRA | ALSTOP     |                            |
| 460 | *      |     |            |                            |
| 470 | STACK  | RMB | 100        | ruimte voor de stack       |
| 480 | *      |     |            |                            |
| 490 | VOET   | LDA | #1         |                            |
| 500 |        | STA | INT        | set INT-flag               |
| 510 |        | RTI |            |                            |
| 520 |        | END | OVER       |                            |

- 3 Om een veranderde schakelaarstand zo snel mogelijk in de LED's te zetten, moet het lezen van de schakelaars in de wachtlus plaatsvinden. De kleinste wachttijd moet ongeveer 50 ms bedragen. Vandaar dat we een basiswachttijd van 50 ms kiezen, die een aantal malen (TIJD) kan worden doorlopen. Door (TIJD) te verdubbelen, wordt de wachttijd verdubbeld en de loopsnelheid gehalveerd. (TIJD)=1 levert dus een wachttijd op van 50 ms. Zeven keer verdubbelen levert een wachttijd van  $128 \cdot 50 \text{ ms} = 6.4 \text{ s}$  op.

Het hoofdprogramma verzorgt slechts het roteren van de LED's. De subroutine X50ms telt het gewenste veelvoud (TIJD) van 50 ms af en zorgt tevens voor het zichtbaar maken van de stand van de schakelaars op de LED's. De interrupt service routine CHANGE zorgt voor het veranderen van de loopsnelheid.

|                                                          |
|----------------------------------------------------------|
| disable de interrupts                                    |
| initialiseer de IRQ-vector<br>en de PIA: (TIJD) := 0.8 s |
| LED0 := 1; de overige LED's<br>worden gedooft            |
| enable IRQ-interrupts                                    |
| altijd                                                   |
| schuif LED's naar links                                  |
| alle LED's uit                                           |
| ja nee                                                   |
| LED0 := 1 /                                              |
| wacht (TIJD) maal 50 ms                                  |

|                     |        |
|---------------------|--------|
| CA1-interrupt       | ja nee |
| reset CA1-interrupt | /      |
| halveer (TIJD)      |        |
| (TIJD) < 50ms       |        |
| ja nee              | /      |
| (TIJD) := 50 ms     |        |
| CB1-interrupt       | ja nee |
| reset CB1-interrupt | /      |
| (TIJD) < 6.4s       |        |
| ja nee              |        |
| verdubbel (TIJD)    | /      |

```

10 NAM ROTEEER LED'S
20 *Er wordt een looplicht geroteerd van PA0 tot PA7 en dan
30 *weer vanaf PA0. Tegelijkertijd wordt de stand van PB0
40 *t/m PB3 getoond op PB4 t/m PB7.
50 *
60 *Een interrupt op CA1 (opgaande flank) verdubbelt de loop-
70 *snelheid, een interrupt op CB1 (neergaande flank)
80 *halveert de loopsnelheid.
90 *Een LED moet minstens 50 ms oplichten, maar zal niet
100 *langer dan 6.4 s zijn.
110 *
120 ORG $4000
130 TIJD RMB 1 ruimte voor wachttijd
140 IRQVEC EQU $DFF8
150 *
160 ROTEEER ORCC #$50 disable de interrupts
170 LDX #CHANGE
180 STX IRQVEC initialiseer IRQ-vector
190 BSR PIAINT initialiseer de PIA
200 LDB #$10
210 STB TIJD (TIJD) := 16.50 ms = 0.8 s
220 LDA #1
230 STA PRA initialiseer de LED's
240 ANDCC #$EF enable de IRQ-interrupt
250 SCHUIF LSL PRA spring als niet alle LED's
260 BNE WACHT uit zijn
270 LDA #1
280 STA PRA initialiseer de LED's
290 WACHT BSR X50ms
300 BRA SCHUIF
310 *
320 *Subroutine PIAINT
330 *Hierin wordt de PIA op de juiste wijze ingesteld
340 *
350 PIAINT CLR CRA adresseer DDRA
360 CLR CRB en DDRB

```

```

370 LDA #$FF
380 STA DDRA poort PA wordt output
390 LDA #%11110000 PB7 t/m PB4 wordt output,
400 STA DDRB PB3 t/m PB0 input
410 LDA #00000111 adresseer PRA, opgaande flank
420 STA CRA op CA1 geeft interrupt
430 LDA #00000101 adresseer PRB, neergaande flank
440 STA CRA op CB1 geeft interrupt
450 TST PRA reset de interrupt request
460 TST PRB bits IRQA1 en IRQB1
470 RTS
480 *
490 *Subroutine X50ms
500 *Er wordt (TIJD) maal 50 ms gewacht.
510 *
520 X50ms LDA TIJD
530 LANG LDX #1724 wachttijd LANG=1724*29µs=50 ms
540 KORT LDB PRB toon PB0 t/m PB3 5 µs
550 LSRB op PB4 t/m PB7 2 µs
560 LSRB 2 µs
570 LSRB 2 µs
580 LSRB 2 µs
590 STB PRB 5 µs
600 LEAX -1,X verlaag (X) met 1 8 µs
610 BNE KORT 3 µs
620 DECA
630 BNE LANG
640 RTS
650 *
660 *Interrupt service routine CHANGE
670 *Hier wordt onder besturing van CA1 en CB1 de looptijd
680 *verdubbeld, respectievelijk gehalveerd.
690 *
700 CHANGE TST CRA spring als er
710 BPL TSTCB1 geen CA1-interrupt is
720 TST PRA reset CA1-interrupt
730 LSR TIJD halveer (TIJD)
740 BNE TSTCB1 spring als (TIJD) ≥ 50 ms
750 INC TIJD (TIJD) := 50 ms
760 TSTCB1 TST CRB spring als er
770 BPL ENDISR geen CB1-interrupt is
780 TST PRB reset CB1-interrupt
790 TST TIJD
800 BMI ENDISR spring als (TIJD) 6,4 s
810 LSL TIJD verdubbel (TIJD)
820 ENDISR RTI

```

- 4 Voor een pulsed handshake moeten de PIA's als volgt zijn ingesteld, ingeval er tevens sprake is van interrupt-I/O:
- "Uitvoer-PIA": (DDRB) = \$FF en (CRB) = %00101101
- "Invoer-PIA": (DDRA) = \$00 en (CRA) = %00101101
- De PIA's staan in de strobe mode; de opgaande flank mag eventueel ook de actieve interruptflank zijn.

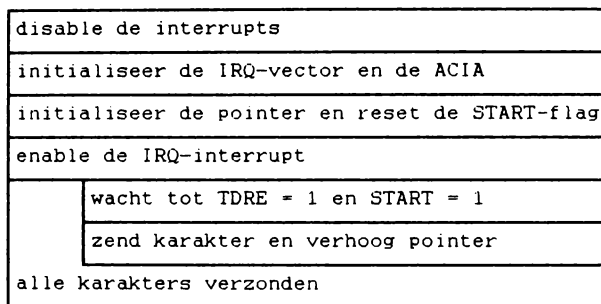


```

160 *
170 FOLHAN LDA CRB spring als geen
180 BPL ENDISR CB1-interrupt
190 TST PRB reset CB1-interrupt
200 EORA #$02
210 STA CRB maak andere CB1-flank actief
220 BITA #$FD spring als bit 2 = 1, dan
230 BNE RUST was CB1 neergaand actief
240 ZEND LDB DATA
250 STB PRB zend data
260 ORA #$08
270 STA CRB set CB2: "hier is data"
280 BRA ENDISR
290 RUST ANDA #$F7 reset CB2: "data ongeldig"
300 ENDISR RTI

```

- 6 Voor asynchroon transport mag de deelfactor voor de frequentie niet gelijk zijn aan 1. Derhalve moet gekozen worden voor de 78.6 kHz oscillator, zodat de deelfactor 64 bedraagt. In de ISR wordt de flag START geset na het ontvangen van XON, terwijl START weer wordt gereset na het ontvangen van XOFF. Het structogram voor het hoofdprogramma ziet er als volgt uit:



```

100 NAM SERIETRANSPORT
110 *Dit programma verzendt de bytes van de locaties $8000
120 *t/m $8700 met programmed I/O via een ACIA.
130 *Via die ACIA wordt met interrupt-I/O het startteken
140 *XON en het stopteken XOFF ontvangen.
150 *De communicatie vindt plaats met 8-bits data, 1
160 *stopbit en zonder pariteitsbit.
170 *
180 ORG $6000
190 START RMB 1
200 SR EQU $4000 adres statusregister
210 CR EQU SR adres controlregister
220 RDR EQU $4001 adres receive data register
230 TDR EQU RDR adres transmit data register
240 IRQVEC EQU $DFF8
250 *
260 SERIE ORCC #$50 blokkeer interrupts
270 LDX #INPUT
280 STX ISQVEC
290 LDA #3
300 STA CR reset ACIA
310 LDA #%10010110

```



```

320 STA CR initialiseer de ACIA
330 LDX #$8000 initialiseer de pointer
340 CLR START
350 ANDCC #$EF enable de IRQ-interrupt
360 NEXT LDA SR
370 BITA #2
380 BEQ NEXT wacht tot TDRE = 1
390 WACHT TST START
400 BEQ WACHT wacht tot START = 1
410 LDA ,X+ lees karakter, verhoog pointer
420 STA TDR zend karakter
430 CMPX #$8701 spring als nog niet alle
440 BNE NEXT karakters zijn verzonden
450 SWI
460 *
470 *ISR INPUT
480 *
490 XON EQU $11
500 XOFF EQU $13
510 *
520 INPUT LDA SR
530 BITA #1
540 BEQ ENDISR spring als RDRF = 0
550 LDA RDR lees ontvangen karakter
560 CMPA #XON
570 BNE NOXON
580 LDB #1
590 STB START
600 BRA ENDISR
610 NOXON CMPA #XOFF
620 BNE ENDISR
630 CLR START
640 ENDISR TST RDR reset receiver interrupts
650 RTI

7 100 * NAM DENDERTIJDMETING
110 *Met behulp van een timer en een PIA wordt de dendertijd
120 *van een contact gemeten en gedisplayed (in μ s).
130 *
140 ORG $7000
150 TIJD RMB 2
160 STOP RMB 1
170 IRQVEC EQU $DFF8
180 *
190 DENDER ORCC #$50 blokkeer interrupts
200 LDX #IRQ
210 STX IRQVEC
220 BSR TIMPIA initialiseer PTM en PIA
230 CLR STOP
240 ANDCC #$EF enable de IRQ-interrupt
250 TEST TST STOP
260 BEQ TEST wacht op einde meting
270 LDD TIJD
280 COMA
290 COMB
290 STA PRA trek (TIJD) af van latches 2
300 STB PRB = 65.535 μ s
310 SWI
320 *
330 *
340 *Initialisatieroutine voor PTM en PIA
350 *

```

```

360 TIMPIA LDA %#01110011 int. enable. single shot
370 STA PTMCR2 16-bits mode. initialisatie
380 * op gate input 2, E-klok
390 LDA #1
400 STA PTMCR1 reset de PTM
410 CLR PTMCR1 hef de reset op
420 LDD #$FFFF
430 STD PTMHR2 latches 2 := 65.535
440 CLR CRA adresseer DDRA
450 CLR CRB en DDRB
460 LDA #$FF
470 STA DDRA PA en PB
480 STA DDRB worden output
490 LDA %#000000101 adresseer PRA; CA1-interrupt
500 STA CRA op neergaande flank
510 LDA #4
520 STA CRB adresseer PRB, geen CB-int.
530 TST PRA reset IRQA1
540 RTS
550 *
560 *IRQ-ISR
570 *Bij een CA1-interrupt wordt de inhoud van timer 2
580 *opgeborgen in op adres TIJD.
590 *De timerinterrupt I2 geeft het einde van de meting aan.
600 *STOP wordt hierbij geset en de timerinterrupt wordt
610 *geblokkeerd.
620 *
630 IRQ TST CRA
640 BPL NOCA1
650 TST PRA reset IRQA1
660 LDD PTMHC2 zet tellerstand
670 STD TIJD in TIJD
680 NOCA1 LDA PTMSR
690 TST PTMLC2 reset interrupt van time 2
700 BITA #2
710 BEQ ENDIRQ spring als I2=0
720 LDA #1
730 STA STOP
740 ENDIRQ RTI

```

- 8 FDC heeft hoogste prioriteit. dus moet kanaal 0 zijn.  
 Data-acquisitiestation (kanaal 1) moet in data chain staan.  
 Er wordt in beide gevallen informatie in geheugen  
 geschreven. Dus:

```

FDC: (Address Register 0) = $3000
 (Byte Count Register 0) = 128
 (Channel Control Register 0) = %00XX 0000
data-acq.: (Address Register 1) = $9000
 (Byte Count Register 1) = 30
 (Channel Control Register 1) = %00XX 0000
verder: (Priority Control Register) = %0XXX XX11
 (Interrupt Control Register) = %0XXX 0000
 (Data Chain Register) = %XXXX 0011

```

## § 11.8

- 1 De SRAM in fig. 11.19 wordt geselecteerd als A12 t/m A15 0 zijn. dus voor de adressen \$0000 t/m \$0FFF.

- 2 Bij lezen in een PIA moet voldaan worden aan de setuptijd van de data voor de microprocessor. De data moet tenminste 80 ns voor de neergaande flank van E op de processoringenangen aanwezig zijn.
- t = 0 ns (opgaande flank van E bij processor)
  - + 35 ns (maximale buslooptijd voor E)
  - + 290 ns (maximale data delay van PIA)
  - + 80 ns (minimaal vereiste data setuptijd van processor)
  - + tDi ns (maximale buslooptijd voor Di)
- $$= 405 + tDi \text{ ns}$$

Dit moet kleiner zijn dan de minimale breedte van de 1-puls van E, dus kleiner dan 450 ns.

Hieruit volgt dat tDi maximaal 45 ns mag bedragen.

Bij schrijven in een PIA moet worden voldaan aan de data setuptijd van de PIA. Deze moet tenminste 165 ns bedragen.

- t = 0 ns (neergaande flank van E bij de processor)
  - + 250 ns (maximale tijd tussen E neergaand en Q opgaand)
  - + 25 ns (maximale stijgtijd van Q)
  - + 200 ns (maximale data delay van processor)
  - + tDi ns (maximale buslooptijd voor Di)
  - + 165 ns (minimaal vereiste data setuptijd van PIA)
  - 3 ns (minimale buslooptijd voor E)
- $$= 637 + tDi \text{ ns}$$

Dit moet kleiner zijn dan de minimale periodetijd van E, zijnde 1000 ns. Dit betekent dat de buslooptijd van de data maximaal 363 ns mag zijn.

Zowel lezen als schrijven moet goed gaan. Derhalve mag de delay van de databus hoogstens 45 ns bedragen.

- 3 a. Bij gebruik van de DMA/Bus request ingang wordt de interne processorklok stilgezet. Dit mag maximaal 16  $\mu$ s duren. Derhalve neemt de processor tijdig de besturing van de bus terug voor een self-refresh. Na elke 16 cycli moet deze self-refresh plaatsvinden. Van deze 16 klokcycli gaan er 2 af voor de overdracht van de busbesturing (dode cycli). Er blijven dus 14 cycli over voor DMA transfers. Dit betekent dat er maximaal 14 bytes kunnen worden overgedragen per 17 klokcycli. De volgorde van de diverse cycli is:
- dode cyclus - 14 DMA-cycli - dode cyclus - self-refresh-cyclus - dode cyclus - enzovoorts.
- Voor 7\$14 = 98 bytes zijn dus minstens 7\$17 = 119 klokcycli nodig. Voor de resterende 2 bytes zijn nog eens 4 cycli nodig, te weten 2 DMA-cycli en 2 dode cycli.
- Voor de 100 bytes zijn dus minimaal 123 klokcycli nodig.
- b. Bij gebruik van de halt-ingang lopen de processorklokken gewoon door en kunnen aaneensluitend alle bytes worden overgedragen. Alleen aan het begin en aan het einde van de bloktransfer komt een dode cyclus voor. Minimaal zijn in dit geval dus 102 klokcycli nodig.

## § 12.8

1 MODU ---

---

\* Einde van MODU  
LOCALS RMB 20

creeer ruimte voor 20 locals

De pointer voor de local storage kan op een positie-onafhankelijke manier worden geïnitieerd met de instructie

LEAX LOCALS.PC

2 JMP VERDER.PC

Nadelen ten opzichte van (L)BRA VERDER:

- kost meer geheugenruimte
- duurt langer

Voordeel ten opzichte van (L)BRA VERDER:

- + het is mogelijk via een jump table te springen (zie ook paragraaf 12.4.6).

3 Bij een IRQ worden alle registers op de stack gezet.

Hierbij is de volgorde van opbergen:

PCH, PCL, UH, UL, YH, YL, XH, XL, DP, B, A, CC.

De stackpointer wijst naar CC. Dit betekent dat de locatie met XH te bereiken is via het adres (S)+5.

De overdracht van een parameter via register X vanuit de IRQ-interrupt service routine naar het onderbroken programma kan derhalve worden gerealiseerd met behulp van de instructie

STX 5,S

als de bewuste parameter zich in de interrupt service routine in register X bevindt. Bevindt hij zich in een ander register, dan moet STX worden vervangen door STD, STY of STU, al naar gelang van toepassing is. In elk geval wordt X bij het uitvoeren van de RTI-instructie aan het einde van de interrupt service routine vervangen door de inhoud van de stacklocatie 5,S. Hiermee is de parameter inderdaad in X terechtgekomen.

4 NUM RMB 1  
ABS RMB 1  
KWAD RMB 2

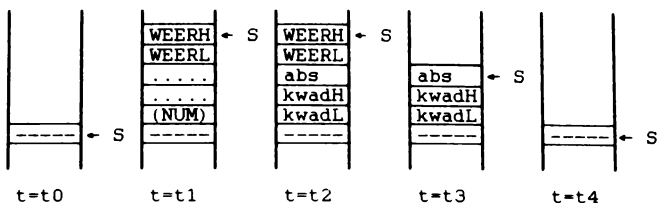
a. parameteroverdracht via de stack

```
PRGR ---

 LDA NUM to0
 PSHS A zet (NUM) op de stack
 LEAS -2,S creëer ruimte voor parameters
 BSR CALC t1
WEER PULS A,X lees parameters, hef ruimte op
 STA ABS t4
 STX KWAD

 SWI
 CALC LDA 4,S lees (NUM)
 BPL POS spring als (NUM) positief is
 NEGA NEG maak (NUM) positief
 POS STA 2,S berg absolute waarde op
 TFR A,B
 MUL MUL bepaal het kwadraat
 STD 3,S berg het kwadraat op t2
 RTS t3
```

De opbouw van de stack ziet er op de opeenvolgende tijdstippen als volgt uit:



#### b. parameteroverdracht via een pointer op de stack

```

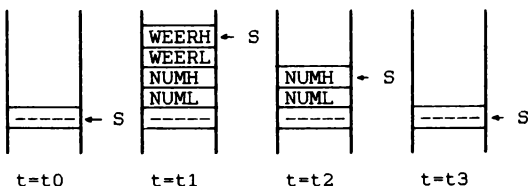
PRGR ---

 LDX #NUM (X):=beginadres getallen t0
 PSHS X zet de pointer op de stack
 BSR CALC t1
WEER LEAS 2.S herstel stackpointer

 SWI
CALC LDY 2.S initialiseer de pointer
 LDA .Y lees (NUM)
 BPL POS spring als (NUM) positief is
 NEGA
POS STA 1.Y maak (NUM) positief
 TFR A.B berg absolute waarde op
 MUL
 STD 2.Y bepaal het kwadraat
 RTS berg het kwadraat op
 t2

```

De opbouw van de stack ziet er op de opeenvolgende tijdstippen als volgt uit:



Let op!

NUM is een adres: (NUM) is de inhoud van de betreffende geheugenlocatie.

```

5 G RMB 1
 H RMB 1
 VALUE RMB 2

```

#### a. parameteroverdracht via registers

```

MAIN ---

 LDA G
 LDB H
 BSR REKEN
 STX VALUE

 SWI

```

|       |      |     |                  |
|-------|------|-----|------------------|
| REKEN | PSHS | B   | red (H)          |
|       | LDB  | #25 |                  |
|       | MUL  |     | (D) := 25 * (G)  |
|       | TFR  | D,X |                  |
|       | PULS | B   | haal (H) op      |
|       | ABX  |     | (X) := (X) + (H) |
|       | RTS  |     |                  |

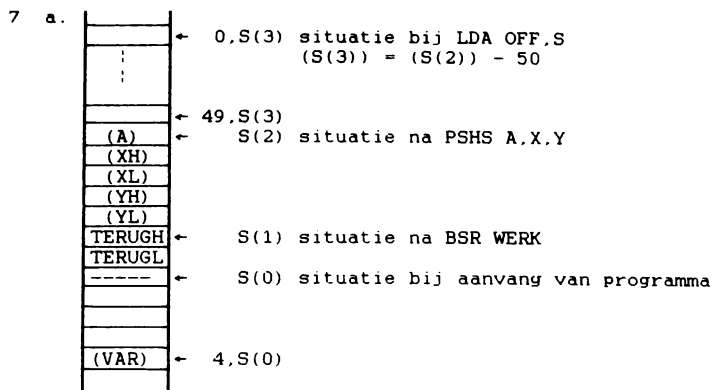
b. parameteroverdracht via vaste geheugenlocaties

|       |      |       |                  |
|-------|------|-------|------------------|
| MAIN  | ---- |       |                  |
|       | ---- |       |                  |
|       | BSR  | REKEN |                  |
|       | ---- |       |                  |
|       | ---- |       |                  |
|       | SWI  |       |                  |
| REKEN | LDA  | G     |                  |
|       | LDB  | #25   |                  |
|       | MUL  |       | (D) := 25 * (G)  |
|       | TFR  | D,X   |                  |
|       | LDB  | H     |                  |
|       | ABX  |       | (X) := (X) + (H) |
|       | STX  | VALUE |                  |
|       | RTS  |       |                  |

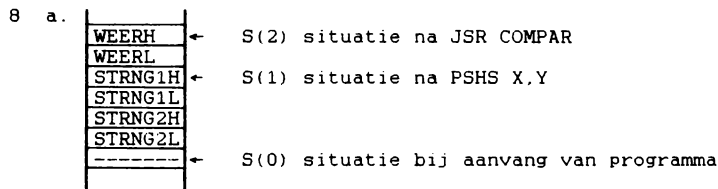
c. parameteroverdracht via een pointer op twee vaste geheugenlocaties

|       |      |       |                         |
|-------|------|-------|-------------------------|
| POINT | RMB  | 2     |                         |
| MAIN  | ---- |       |                         |
|       | ---- |       |                         |
|       | LDU  | #G    |                         |
|       | STU  | POINT | initialiseer de pointer |
|       | BSR  | REKEN |                         |
|       | ---- |       |                         |
|       | ---- |       |                         |
|       | SWI  |       |                         |
| REKEN | LDY  | POINT | laad het beginadres     |
|       | LDA  | ,Y    | lees (G)                |
|       | LDB  | #25   |                         |
|       | MUL  |       | (D) := 25 * (G)         |
|       | TFR  | D,X   |                         |
|       | LDB  | 1,Y   | lees (H)                |
|       | ABX  |       | (X) := (X) + (H)        |
|       | STX  | 2,Y   | berg uitkomst op        |
|       | RTS  |       |                         |

- 6 A wordt geladen met de inhoud van SWITCH, dus met de waarde \$0C. Vervolgens wordt (A) met 2 vermenigvuldigd en wordt zodoende \$18 = 24.  
 X is geladen met het adres ACTTAB. ACTTAB+0 bevat het getal \$F1, ACTTAB+1 bevat \$00, enzovoorts.  
 ACTTAB+24 en ACTTAB+25 bevatten derhalve de waarden \$F3 en \$A0, zodat de aangeroepen subroutine begint op adres \$F3A0.



b. Om de waarde van VAR te laden met LDA OFF.S moet OFF de waarde  $50+5+2+4 = 61$  hebben.



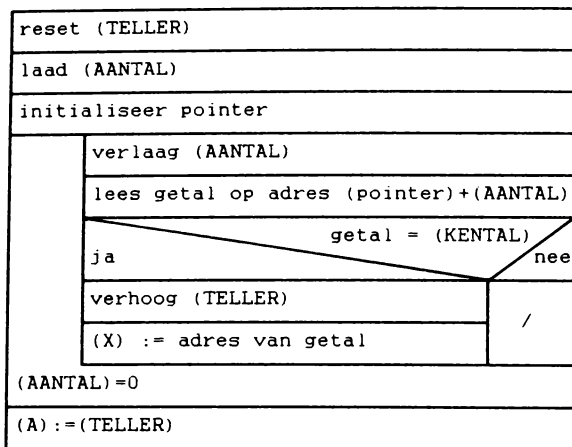
b. Om de waarde \$AA te laden met LDA [CON1.S] moet CON1 de waarde 2 hebben. Zo moet CON2 de waarde 4 hebben, teneinde \$BB te laden met de instructie LDB [CON2.S].

- 9 Bij het verplaatsen van een geheugenblok naar hogere adressen, moet van achter naar voren worden gewerkt om te voorkomen, dat er data verloren gaat. Dit zou anders gebeuren als  $(DISPL) < (LEN)$ .  
Gebruik geen absolute adressering!

|      |      |          |                                 |
|------|------|----------|---------------------------------|
| MOVE | LEAY | BLOK,PC  | Y wijst naar begin huidige blok |
| *    |      |          |                                 |
|      | TFR  | Y,X      |                                 |
|      | LDB  | DISPL,PC |                                 |
|      | ABX  |          | X wijst naar begin nieuwe blok  |
|      | LDB  | LEN,PC   |                                 |
|      | LEAY | B,Y      | laat de pointers                |
|      | LEAX | B,X      | achter de blokken wijzen        |
| NEXT | LDA  | ,Y       | verlaag de pointers en          |
|      | STA  | ,X       | verplaats een byte              |
|      | DECB |          |                                 |
|      | BNE  | NEXT     | spring als niet alles           |
| *    |      |          | verplaatst                      |
|      | RTS  |          |                                 |

- 10 Bij het lezen van de tabel getallen van boven naar beneden, wordt vanzelf bewerkstelligd, dat het laagste adres van het gevonden getal bewaard blijft.

Het structogram ziet er als volgt uit:



Gebruik geen absolute adressering!

|    |        |       |           |                             |
|----|--------|-------|-----------|-----------------------------|
|    | TELLER | RMB   | 1         | telt hoe vaak (KENTAL)      |
|    | *      |       |           | wordt gevonden              |
|    | ZOEK   | CLR   | TELLER,PC |                             |
|    |        | LDD   | AANTAL,PC |                             |
|    |        | LEAY  | TABEL,PC  | U wijst naar begin tabel    |
|    | GADOOR | SUBD  | #1        | verlaag (AANTAL)            |
|    |        | LDU   | D,Y       | lees getal                  |
|    |        | CMPU  | KENTAL,PC | spring als getal ongelijk   |
|    |        | BNE   | NOTEQU    | is aan (KENTAL)             |
|    |        | INC   | TELLER,PC |                             |
|    |        | LEAX  | D,Y       | (X) := adres getal          |
|    | NOTEQU | CMPD  | #0        |                             |
|    |        | BNE   | GADOOR    |                             |
|    |        | LDA   | TELLER,PC |                             |
|    |        | RTS   |           |                             |
| 11 | 100    | FLAG  | EQU       | 0                           |
|    | 110    | MULT2 | LEAS      | -1.S                        |
|    | 115    | *     |           | creëer ruimte op de stack   |
|    | 120    |       | CLR       | FLAG.S                      |
|    | 130    |       | TSTA      | reset FLAG                  |
|    | 140    |       | BPL       | POSA                        |
|    | 150    |       | NEGA      | spring als (A) positief is  |
|    | 160    |       | INC       | maak (A) positief           |
|    | 170    | POSA  | TSTB      |                             |
|    | 180    |       | BPL       | POSB                        |
|    | 190    |       | NEGB      | spring als (B) positief is  |
|    | 200    |       | INC       | maak (B) positief           |
|    | 210    | POSB  | MUL       |                             |
|    | 215    | *     |           | (D) := produkt van absolute |
|    | 220    |       | LSR       | waarden                     |
|    | 230    |       | BCC       | spring als FLAG is even,    |
|    | 235    | *     |           | dus als produkt positief    |
|    | 240    |       | COMA      | moet blijven                |
|    |        |       |           | maak                        |



|     |       |          |                            |
|-----|-------|----------|----------------------------|
| 250 | NEGB  |          | het produkt                |
| 260 | ADCA  | #0       | negatief                   |
| 270 | EINDE | LEAS 1.5 | hef de ruimte voor FLAG op |
| 280 | RTS   |          |                            |

Enkele opmerkingen:

- Regel 110 en 120 kunnen worden gecombineerd tot CLR ,-.S.  
Regel 220 en 270 kunnen worden gecombineerd tot  
LSR FLAG,S+ op regel 220. waarbij de label EINDE  
verschuift naar regel 280.  
Dit alles is vanwege de leesbaarheid echter nagelaten.
- Het negatief maken van (D) kan geschieden door (D) af te  
trekken van 0. Dit is gelijk aan het inverteren van (D)  
met COMA en COMB en vervolgens 1 erbij te tellen met  
ADDB #1, waarbij de onstane carry weer bij (A) moet  
worden opgeteld met ADCA #0. COMB en ADDB #1 kunnen  
worden gecombineerd tot NEGB.

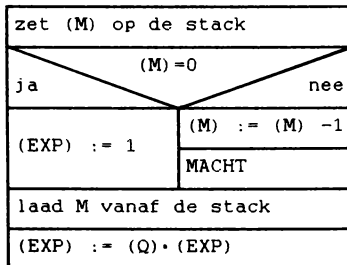
- ```

12 Als (X)>(Y), verwissel (X) en (Y), zodat
    (Y) de grootste van die twee wordt
    Als nu (Y)>(U), verwissel (Y) en (U), zodat
    (U) nu de grootste van de drie is
    Als hierna (X)>(Y), verwissel (X) en (Y), zodat
    (X) nu de kleinste van de drie is

```

HULP	EQU	0	definieer local
VOLG	LEAS	-1.S	HULP op de stack
	STY	HULP.S	
	CMPX	HULP.S	
	BLE	XLAGER	spring als (X)<=(Y)
	EXG	X,Y	
XLAGER	STU	HULP.S	
	CMFY	HULP.S	
	BLE	YLAGER	spring als (Y)<=(U)
	EXG	Y,U	
YLAGER	STY	HULP.S	
	CMPX	HULP.S	
	BLE	XKLEIN	spring als (X)<=(Y)
	EXG	X,Y	
XKLEIN	LEAS	1.S	hef local op
	RTS		

- 13 Het structogram van de recursieve subroutine **MACHT** ziet er als volgt uit:



130	BRA	LAAD	
140 NIETO	DECB		
150	BSR	TELOP	
160 LAAD	PULS	B	
170	ABX		som := som + (B)
180	RTS		

In regel 100 is de instructie STB $\leftarrow S$ gebruikt in plaats van PSHS B, omdat de laatste instructie de Z-bit in het condition code register niet beïnvloedt (zie appendix B). In dat geval zou er een aparte instructie TSTB nodig zijn.

Voor (B)=3 ziet de stackopbouw er achtereenvolgens als volgt uit (MAIN is het terugkeeradres in het aanroepende programma):

