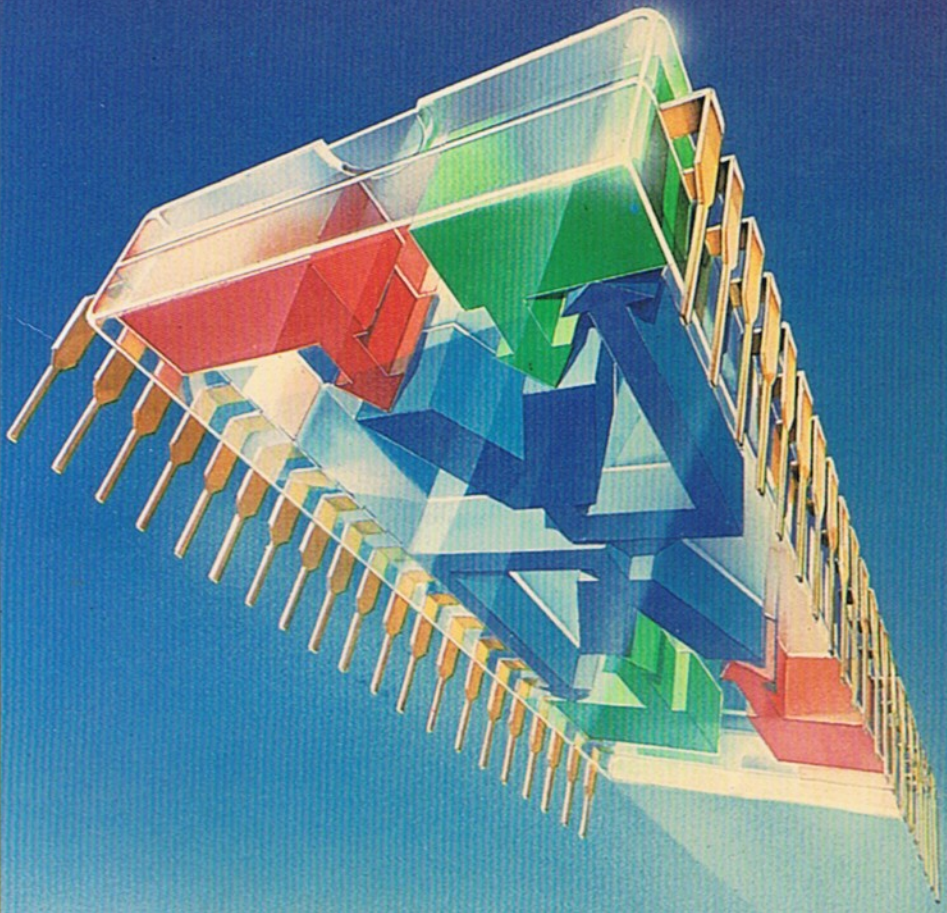
Programmeren van de 6502



RODNAY ZAKS

SYBEX

programmeren van de 6502

RODNAY ZAKS

EERSTE UITGAVE

Originele uitgave in het engels
Titel van de engelse uitgave: Programming the Z80

Druk: Drukkerij Salland BV, Deventer

Omslag ontwerp door **Daniel Le Noury**

Nederlandse vertaling door **Hans Scholten**

ISBN 3-88745-101-5
Eerste druk 1982, tweede druk 1983.

VOORWOORD

Dit boek is opgezet als een volledig en op zichzelf staand geheel om, gebruik makend van de 6502, te leren programmeren. Het is bedoeld voor hen, die nog nooit eerder geprogrammeerd hebben en is ook waardevol voor degenen die de 6502 gebruiken.

Lezers met programmeerervaring zullen in dit boek programmeertechnieken vinden, waarbij van de specifieke karakteristieken van de 6502 gebruik gemaakt wordt. Deze tekst behandelt de technieken van elementair tot gemiddeld niveau, die nodig zijn om te kunnen beginnen met doelmatig programmeren.

De doelstelling van dit boek is om degene, die deze microprocessor wil gebruiken, een redelijke programmeervaardigheid bij te brengen. Het spreekt vanzelf, dat geen enkel boek deze programmeervaardigheid bij kan brengen tenzij men de geleerde technieken in praktijk brengt. Hoe dan ook, we hopen dat dit boek de lezer zover zal brengen dat hij het gevoel heeft dat hij zelf kan beginnen met programmeren, en eenvoudige, of zelfs redelijk ingewikkelde problemen op kan lossen met behulp van een microcomputer.

Dit boek is gebaseerd op de ervaring die de auteur heeft opgedaan tijdens het onderwijzen van meer dan 1000 personen op het gebied van het programmeren van microcomputers. Het gevolg is, dat het in hoge mate gestruktureerd is. De moeilijkheidsgraad van de hoofdstukken loopt van eenvoudig tot ingewikkeld. Lezers die al elementaire programmeertechnieken beheersen, kunnen het inleidende hoofdstuk overslaan. Voor anderen, die nog nooit eerder geprogrammeerd hebben, kan het nodig zijn de laatste delen van sommige hoofdstukken nog eens door te lezen. Het boek is opgezet om de lezer systematisch door alle basisbegrippen en technieken te leiden, die nodig zijn om steeds complexere programma's te ontwikkelen. Het wordt daarom ten sterkste aangeraden de volgorde van de hoofdstukken aan te houden. Het is daarbij belangrijk dat de lezer, om effektief resultaat te krijgen, zoveel mogelijk oefeningen maakt. De moeilijkheidsgraad van de oefeningen is zorgvuldig opgebouwd. Ze zijn ontworpen om te verifieren of het gepresenteerde materiaal ook werkelijk begrepen is. Het is moeilijk om de onderwijskundige kwaliteiten van dit boek op hun waarde te schatten, als de programmeeroefeningen niet gemaakt worden. Enke-

le oefeningen zullen meer tijd vergen, zoals bijvoorbeeld de vermenigvuldigingsoefening. Maar door deze oefeningen te maken programmeer je en al doende leert men. Dit is onmisbaar.

Voor hen, die aan het eind van dit boek de smaak van het programmeren te pakken hebben gekregen, is een begeleidend boek te verkrijgen: het "6502 Applications Book".

Andere boeken uit deze serie behandelen het programmeren van andere populaire microprocessoren.

Zij, die hun hardware kennis willen ontwikkelen, worden aangeraden de naslagwerken "From Chips to Systems: An introduction to Microprocessing" (ref C201) en "Microprocessor Interfacing Techniques" (ref C207) te raadplegen.

De inhoud van dit boek is zorgvuldig gekontroleerd en we geloven dat die betrouwbaar is. Het is echter onvermijdelijk, dat er enkele typografische of andere fouten gevonden zullen worden. De auteur zal dankbaar zijn voor opmerkingen van alerte lezers, zodat toekomstige uitgaven voordeel kunnen trekken uit hun ervaringen. Andere suggesties voor verbetering, zoals andere programma's die gewenst of ontwikkeld zijn, of die de lezers waardevol vinden, worden gewaardeerd.

INHOUDSOPGAVE

1
BASIS BEGRIPPEN

INLEIDING

In dit hoofstuk worden de basisbegrippen en definities geintroduceerd, die betrekking hebben op het programmeren van een computer. De lezer die al op de hoogte is met deze begippen zal de inhoud van dit hoofdstuk snel door willen nemen, om dan door te gaan met hoofdstuk 2. Het wordt echter aangeraden, dat ook de ervaren lezer dit hoofdstuk doorneemt: veel belangrijke begrippen worden hier gepresenteerd, met inbegrip van bijvoorbeeld 2-komplement notatie, BCD en andere representaties.

WAT IS PROGRAMMEREN?

Gegeven een probleem moet eerst een oplossing bedacht worden. Deze oplossing, uitgedrukt als een stap-voor-stap procedure wordt een *algoritme* genoemd. Een algoritme is een stap-voor-stap specificatie van de oplossing van een gegeven probleem. Het moet ook in een eindig aantal stappen beeindigd zijn. Dit algoritme kan in iedere taal worden uitgedrukt.

Een voorbeeld van een typisch algoritme is:

1 - steek de sleutel in het slot
2 - draai de sleutel een volledige slag linksom
3 - pak de deurknop
4 - draai de deurknop linksom en duw tegen de deur

Nu zal, als dit algoritme juist is voor het betrokken slot, de deur opengaan. Deze vier-staps procedure komt in aanmerking als een algoritme voor het openen van een deur.

Als de oplossing van een probleem in de vorm van een algoritme is uitgedrukt, moet het algoritme door de computer uitgevoerd worden. Helaas is het nu een vaststaand feit, dat computers geen gewoon gesproken of geschreven Nederlands (of welke menselijke taal dan ook) kunnen begrijpen en uitvoeren. De reden hiervoor is de syntactische dubbelzinnigheid van alle gebruikelijke menselijke talen. Alleen een goed gedefinieerde deelverzameling van de natuurlijke taal kan door een computer "begrepen" worden. Dit wordt een *programmeertaal* genoemd.

Het omzetten van een algoritme in een serie instrukties in de programmeertaal wordt programmeren genoemd. Om nauwkeuriger te zijn, de eigenlijke vertaalfase van een algoritme in de programmeertal wordt *koderen* genoemd. Met programmeren wordt niet alleen het koderen bedoeld, maar het hele ontwerp van programma's en "gegevensstrukturen" waarmee het programma uitgevoerd wordt.

Doelmatig programmeren vereist niet alleen begrip van mogelijke technieken om standaardalgoritmes uit te voeren, maar ook vakkundig gebruik van alle computer-hardware, zoals interne registers, geheugen en randapparatuur, met daarbij een kreatief gebruik van geschikte gegevensstrukturen. Deze technieken zullen in de volgende hoofdstukken behandeld worden.

Programmeren vereist ook een strikte discipline m.b.t. het dokumenteren van programma's, zodat deze programma's zowel voor de auteur als voor anderen begrijpbaar zijn. Deze dokumentatie moet zowel intern als extern zijn.

Met interne dokumentatie wordt het kommentaar in het programma zelf bedoeld, waarmee de werking van het programma aangegeven wordt.

Met externe dokumentatie wordt die dokumentatie bedoeld die apart van het programma is, zoals geschreven verklaringen, handleidingen en stroomdiagrammen.

STROOMDIAGRAMMEN

Er is bijna altijd een stap tussen het algoritme en het programma, te weten het *stroomdiagram*. Een stroomdiagram is een symbolische representatie van het algoritme, uitgedrukt in een serie rechthoeken en ruiten, die de stappen van het algoritme bevatten. Rechthoeken wor-

den gebruikt voor kommando's, of "uitvoerbare opdrachten". Ruiten worden gebruikt voor testen zoals: "Als informatie X waar is, doe dan A, anders B". In plaats van hier een formele definitie van stroomdiagrammen te geven, zullen we stroomdiagrammen verderop in het boek inleiden en bespreken wanneer we programma's presenteren.

Het maken van een stroomdiagram is een sterk aanbevolen tussenstap tussen het algoritme en het eigenlijke koderen van de oplossing. Opmerkelijkerwijs heeft men waargenomen, dat zo'n 10% van de programmerende bevolking succesvol een programma kan schrijven zonder een stroomdiagram. Helaas heeft men ook waargenomen, dat 90% van die bevolking meent tot deze 10% te horen. Het resultaat: gemiddeld 80% van deze programma's faalt de eerste keer dat ze op een computer uitgevoerd worden. (Deze getallen zijn natuurlijk niet nauwkeurig.) Om kort te gaan, de meeste beginnende programmeurs zien zelden de noodzaak in van het maken van een stroomdiagram. Dit resulteert gewoonlijk in "rommelige" of foute programma's. Ze hebben dan veel tijd nodig om hun programma's te testen en te verbeteren (dit wordt de "ontluizingsfase" (debugging phase) genoemd). Het wordt daarom ten sterkste aanbevolen om zich de discipline van het maken

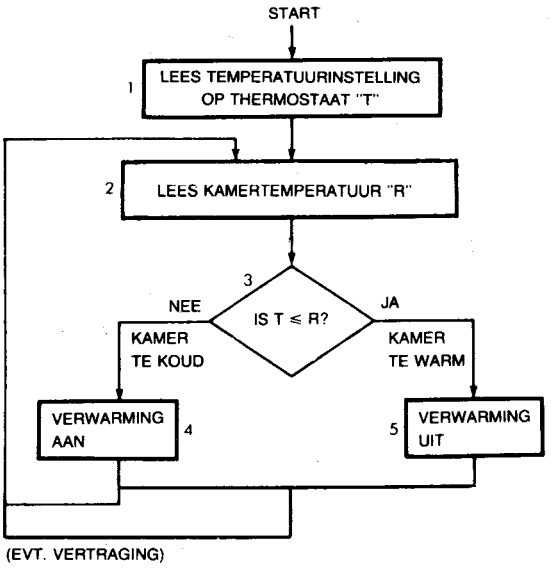


Fig.1-1: Een stroomdiagram om de kamertemperatuur konstant te houden

van een stroomdiagram eigen te maken. Dit kost slechts weinig tijd voor het eigenlijke koderen, maar zal meestal een duidelijk programma dat snel goed uitgevoerd wordt, tot gevolg hebben. Zodra het maken van een stroomdiagram goed begrepen wordt, kan een klein percentage van de programmeurs deze stap uit het hoofd doen, zonder dat ze het op papier hoeven te doen. Helaas zijn in dergelijke gevallen de programma's die ze schrijven meestal moeilijk te begrijpen door anderen zonder de dokumentatie van de stroomdiagrammen. Het resultaat is, dat het maken van een stroomdiagram, als een strikte programmeerdiscipline, algemeen aanbevolen wordt voor ieder belangrijk programma. In dit boek staan vele voorbeelden.

VOORSTELLING VAN INFORMATIE

Alle computers manipuleren informatie, in vorm van getallen of in de vorm van karakters. Laten we hier eens de interne en externe voorstellingen van informatie in een computer bekijken.

INTERNE VOORSTELLING

Alle informatie in een computer wordt opgeslagen in groepen bits. *Bit* is een afkorting van binary digit, dat wil zeggen "0" of "1". Wegens de beperkingen van de konventionele electronica, gebruikt de enige praktische voorstelling van informatie de twee-toestanden logica, dat wil zeggen de voorstelling van de toestanden "0" en "1". De twee toestanden die in de digitale electronica gebruikt worden zijn "aan" en "uit", en deze worden voorgesteld door de logische toestanden "0" en "1". Omdat deze circuits gebruikt worden om "logische" funkties uit te voeren, wordt dit binaire "logika" genoemd.
Het resultaat is dat zo goed als alle informatieverwerking vandaag de dag in het binaire formaat geschiedt. Bij microprocessoren in het algemeen, en de 6502 in het bijzonder zijn deze bits gestruktureerd per 8. Een groep van 8 bits wordt een *byte* genoemd. Een groep van 4 bits een *nibble*.

Laten we nu eens bekijken hoe informatie intern voorgesteld wordt in dit binaire formaat. Twee eenheden moeten in een computer voorgesteld worden. De eerste is het programma, dat een opeenvolging van instrukties is. De tweede eenheid wordt gevormd door de gegevens die het zal bewerken, en die uit getallen of alfanumerieke tekst kan bestaan. Laten we deze drie voorstellingen, programma, getallen en alfanumerieke tekst, eens bekijken:

Programma voorstelling

Alle instrukties worden intern voorstelling in een of meerdere bytes. Een zogenaamde "verkorte instruktie" wordt door en enkel byte voorgesteld. Een lange instruktie wordt voorgesteld door twee of meer bytes. Omdat de 6502 een 8-bits microprocesor is, haalt hij achtereenvolgens bytes uit zijn geheugen. Daarom kan een een-byte instruktie sneller uitgevoerd worden dan een twee- of drie-byte instruktie. Het zal later duidelijk worden, dat dit een belangrijk kenmerk van de instruktieset van een micrprocessor is, en van de 6502 in het bijzonder, waarvoor extra moeite is gedaan om in zoveel mogelijk een-byte instrukties te voorzien als haalbaar was, om daarmee de effektiviteit van het uitvoeren van programma te verhogen. De beperking tot een lengte van 8 bits heeft belangrijke beperkingen die uiteengezet zullen worden. Dit is een klassiek voorbeeld van een kompromis tussen doelmatige snelheid en flexibel programmeren. De binaire voorstelling van instrukties is voorgeschreven door de fabrikant, en de 6502 heeft, zoals alle andere microprocessoren, een vaste instruktieset. Deze instrukties zijn gedefinieerd door de fabrikant en staan achterin het boek vermeld. Ieder programma wordt uitgedrukt als een opeenvolging van deze binaire instrukties. De eigenlijke binaire kodering van de 6502 instrukties wordt gegeven in hoofdstuk 4.

Voorstelling van numerieke gegevens

Het voorstellen van getallen is niet helemaal rechttoe, rechtaan en er moeten verschillende gevallen onderscheiden worden. We moeten eerst gehele getallen voorstellen. Ook moeten we getallen met teken, dat wil zeggen positieve en negatieve getallen voorstellen en tenslotte moeten we nog decimale getallen voor kunnen stellen. Laten we nu deze eisen en mogelijke oplossingen eens onder de loep nemen.

Gehele getallen kunnen voorgesteld worden met de *direkt-binaire* voorstelling. De direkt binaire voorstelling is gewoon de voorstelling van de decimale waarde een getal in het twee-tallig stelsel. In het twee-tallig stelsel stelt het meest rechtse bit 2 tot de macht 0 voor. Het volgende stelt voor 2 tot de macht 1, het volgende 2 tot de macht 2, en het meest linkse bit stelt voor 2 tot de macht 7 = 128.

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

stelt voor

$$b_72^7 + b_62^6 + b_52^5 + b_42^4 + b_32^3 + b_22^2 + b_41^1 + b_02^0$$

De machten van twee zijn:

$2^7 = 128, 2^6 = 64, 2^5 = 32, 2^4 = 16, 2^3 = 8, 2^2 = 4, 2^1 = 2, 2^0 = 1$

De binaire voorstelling is analoog aan de decimale voorstelling van getallen, waarbij "123" het volgende voorstelt:

$$\begin{array}{rrr} & 1 \times & 100 = 100 \\ + & 2 \times & 10 = 20 \\ + & 3 \times & 1 = 3 \\ \hline & & = 123 \end{array}$$

Merk op, dat $100 = 10^2$, $10 = 10^1$, $1 = 10^0$.
Bij deze "plaatsafhankelijke notatie" stelt ieder cijfer een macht van 10 voor. Bij het binaire systeem stelt ieder cijfer een macht van 2 voor, in plaats van een macht van 10 zoals in het decimale systeem.

Voorbeeld: "00001001" stelt in het binaire systeem voor:

$$\begin{array}{rrrr} 1 \times & 1 = 1 & (2^0) \\ 0 \times & 2 = 0 & (2^1) \\ 0 \times & 4 = 0 & (2^2) \\ 1 \times & 8 = 8 & (2^3) \\ 0 \times & 16 = 0 & (2^4) \\ 0 \times & 32 = 0 & (2^5) \\ 0 \times & 64 = 0 & (2^6) \\ 0 \times & 128 = 0 & (2^7) \\ \hline \text{decimaal:} & = 9 & \end{array}$$

Laten we nog een voorbeeld nemen:

"10000001" stelt voor:

$$\begin{array}{rrrr} 1 \times & 1 & = & 1 \\ 0 \times & 2 & = & 0 \\ 0 \times & 4 & = & 0 \\ 0 \times & 8 & = & 0 \\ 0 \times & 16 & = & 0 \\ 0 \times & 32 & = & 0 \\ 0 \times & 64 & = & 0 \\ 1 \times & 128 & = & 128 \\ \hline \text{decimaal:} & & = & 129 \end{array}$$

"10000001" stelt dus het decimale getal 129 voor.

Door de binaire voorstelling van getallen te onderzoeken zul je begrijpen waarom de bits van 0 tot 7 genummerd zijn, van rechts naar links. Bit 0 is "b_0" en korrespondeert met 2^0. Bit 1 is "b_1" en korrespondeert met 2^1, enz.

Decimaal	Binair	Decimaal	Binair
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010	•	
3	00000011	•	
4	00000100	•	
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000	•	
9	00001001	•	
10	00001010	127	01111111
11	00001011	128	10000000
12	00001100	129	10000001
13	00001101		
14	00001110	•	
15	00001111	•	
16	00010000		
17	00010001	•	
•			
•			
•		254	11111110
31	00011111	255	11111111

Fig.1-2: Decimaal-binair tabel

In fig.1-2 zijn de binaire equivalenten van de getallen 0 tot 255 getoond.

Oefening 1.1: Waar is "1111 1110" de voorstelling van?

Decimaal naar binair

Laten we nu het binaire equivalent van "11" eens berekenen:

11 : 2 = 5 rest 1 → 1
5 : 2 = 2 rest 1 → 1
2 : 2 = 1 rest 0 → 0
1 : 2 = 0 rest 1 → 1

Het binaire equivalent is 1011 (lees de meest rechtse kolom van onder naar boven).
Het binaire equivalent van een decimaal getal kan verkregen worden door steeds door twee te delen tot er een rest van 0 over is.

Oefening 1.2: Wat is het binaire equivalent van 257?

Oefening 1.3: Konverteer 19 naar binair en terug.

Werken met binaire gegevens

De rekenkundige regels voor binaire getallen zijn erg eenvoudig. De regels voor optelling zijn:

```
0 + 0 =     0
0 + 1 =     1
1 + 0 =     1
1 + 1 =   (1) 0
```

Hierbij stelt (1) een overdracht voor van 1 (carry). Merk op dat "10" het binaire equivalent van "2" decimaal is. Binair aftrekken gebeurt door het "komplement" er bij op te tellen, en zullen we uitleggen als we de voorstelling van negatieve getallen bespreken.

Voorbeeld:

```
  (2)        10
+(1)        +01
-----      ----
 =(3)       =11
```

Optellen gaat net als bij decimaal, van rechts naar links kolommen optellen:

De meest rechtse kolom optellen:

```
  10
+01
----
```

(0 + 1 = 1. Geen overdracht.)

De volgende kolom optellen:

```
  10
+01
----
  11   (1+0=1. Geen overdracht.)
```

Oefening 1.4: Bereken 5 + 10 in binair. Kontroleer dat het resultaat 15 is.

Nog enkele voorbeelden van binaire optelling:

```
 0010    (2)        0011    (3)
+0001    (1)       +0001    (1)
------             ------
=0011    (3)       =0100    (4)
```

Dit laatste voorbeeld illustreert de funktie van de overdracht (carry).

De meest rechtse bits: 1 + 1 = (1) 0.
Er is een overdracht van 1 gegenereerd, die bij de volgende bits opgeteld moet worden:

```
   001   -kolom 0 is net opgeteld
  +000   -
  +  1   (overdracht)
= (1)0   -hierbij geeft (1) een nieuwe overdracht naar kolom 2 aan.
```

Het eindresultaat is: 0100

Nog een voorbeeld:

```
 0111                    (7)
+0011                  + (3)
------                 ------
 1010                  =(10)
```

In dit voorbeeld vindt er een carry plaats naar de linkse kolom.

Opgave 1.5: Bereken:

```
  1111
 +0001
------
  =?
```

Past het resultaat in 4 bits?

Op deze manier kunnen we met 8 bits alle getallen van "00000000" tot "11111111" voorstellen, d.w.z. van "0" tot "255". Twee hindernissen zijn direct zichtbaar. Ten eerste: we kunnen alleen positieve getallen weergeven. Ten tweede: bij gebruik van 8 bits is de grootte van het getal beperkt tot 255. Laten we deze hindernissen eens bekijken.

Binair met teken

Bij de binair met teken voorstelling wordt het meest linkse bit als tekenbit gebruikt. Traditioneel betekend een "0" een positief getal, en een "1" een negatief. Op deze manier is "11111111" "−127" decimaal, terwijl "01111111" "+127" decimaal is. We kunnen dus nu positieve en negatieve getallen voorstellen, maar de absolute grootte is nu beperkt tot 127.

Voorbeeld: "00000001" is +1
"10000001" is −1
???
|
Tekenbit

Opgave 1.6: Wat is de binair met teken voorstelling van "−5"?

Nu het probleem van de grootte van het getal. Om grote getallen voor te stellen is het noodzakelijk veel bits te gebruiken. Als we bijvoorbeeld 16 bits nemen, kunnen we getallen van −32K tot +32K voorstellen (1K betekent in computer jargon 1024). Bit 15 bevat het teken en bits 14 tot 0 bevatten de absolute waarde. Wanneer dat nog te klein is kunnen we 3 of meer bytes nemen. Hoe groter de getallen, des te meer bytes zijn er nodig. Daarom hebben de meest eenvoudige versies van BASIC slechts een beperkt bereik voor gehele getallen. Betere versies gebruiken meer bytes, en kunnen dus grotere getallen weergeven.

Dan is er nog een ander probleem dat opgelost moet worden: de snelheid. We proberen eens twee getallen binair bij elkaar op te tellen. bijvoorbeeld "−5" en "+7".

```
+7    00000111
−5    10000101
      --------
      10001100 = −12 decimaal
```

Het resultaat klopt niet. Dat moet +2 zijn, i.p.v. −12. Om toch het goede resultaat te krijgen met deze manier van voorstellen, moeten we enkele handelingen meer uitvoeren, die afhankelijk zijn van het teken. De optelling wordt complexer en dat komt de efficientie niet ten goede. Met andere woorden: een optelling binair met teken werkt niet zoals het zou moeten. Dat is vervelend. Om het nog eens duidelijk te stellen, de computer moet niet alleen de getallen voorstellen, maar er ook mee kunnen rekenen.

De oplossing voor dit probleem heet de twee-complement voorstelling, die gebruikt wordt i.p.v. de binair met teken voorstelling. Om het twee-complement duidelijker te maken, maken we eerst een tussenstap: het een-complement.

Een-complement

In het een-complement worden alle positieve getallen normaal binair voorgesteld. Dus "+3" wordt "00000011". Het complement, −3 dus, wordt verkregen door de binaire representatie van +3 te inverteren. D.w.z. iedere "0" wordt een "1" en omgekeerd. In een-complement wordt "−3": 11111100

Nog een voorbeeld:

+2 00000010
−2 11111101

Merk op dat positieve getallen beginnen met een "0" en negatieve met een "1".

Opgave 1.7: Decimaal "+6"is binair "00000110". Wat is "−6" in het een-complement?

Ter controle tellen we −4 en +6 op:

	−4 11111011	
	+6 00000110	
de som is:	(1)00000001	(1) is een carry.

Het juiste antwoord moet "2" of "00000010" zijn. We proberen het nog een keer:

$$\begin{array}{rr} -3 & 11111100 \\ -2 & 11111101 \\ \hline & (1)\ 00000001 \end{array}$$

de som is:

Dit resultaat is gelijk aan "1" plus een carry. Het had "−5" moeten zijn, binair met teken: "11111010". Ook hier klopt het niet.

Met deze representatie kunnen we positieve en negatieve getallen maken. Het resultaat van een optelling klopt echter niet. We zullen daarom nog een andere voorstelling gebruiken, welke ontwikkeld is uit het een-complement: het twee-complement.

Twee-complement voorstelling

M.b.v. het twee-complement worden positieve getallen op dezelfde manier voorgesteld als met de binair met teken voorstelling, die gelijk is aan de een-complement notatie. Het verschil zit in de negatieve getallen. Bij twee-complement krijgen we een negatief getal door eerst het een-complement te berekenen en er dan 1 bij op te tellen. Ter illustratie het volgende voorbeeld:

+3 binair met teken: 00000011
hiervan het een-complement: 11111100
tel hier 1 bij op: 11111101

We proberen eerst een optelling:

$$\begin{array}{rr} 3 & 00000011 \\ +5 & +00000101 \\ \hline =8 & =00001000 \end{array}$$

Het resultaat klopt.
Nu trekken we af:

$$\begin{array}{rr} 3 & 00000011 \\ -5 & +11111011 \\ \hline & =11111110 \end{array}$$

Van dit antwoord berekenen we het twee-complement:

het een-complement is:	00000001
tel er 1 bij op:	1
	00000010 = +2

Het antwoord is dus −2, wat klopt.

We hebben nu een optelling en een aftrekking gedaan en de antwoorden kloppen (afgezien van een eventuele carry). Het lijkt erop, dat het twee-complement werkt!

Opgave 1.8: Bereken het twee-complement van +127

Opgave 1.9: Wat is het twee-complement van −128?

Tel +4 en −3 op. (Doe dit door de twee-complementen op te tellen).

+4	00000100
−3	11111101
(1)	00000001

Het resultaat is 1, als we de carry negeren. Dat klopt dus. Zonder het wiskundige bewijs te leveren, volstaan we hier met de constatering dat deze wijze van voorstelling werkt. In het twee-complement is het mogelijk getallen met een teken op te tellen en af te trekken. Daarbij kan gebruik worden gemaakt van dezelfde rekenregels als van de normale binaire optelling. Het resultaat klopt altijd, inclusief het teken. Dit laatste is zeer belangrijk. Als dat namelijk niet het geval zou zijn geweest, dan zouden we correctieslagen moeten toepassen om toch het juiste teken te krijgen. Het zou dus meer tijd kosten een optelling of een aftrekking uit te voeren.

Om volledig te zijn, moeten we toegeven dat de twee-complement voorstelling de meest gemakkelijke is voor eenvoudige computers, zoals microprocessors. Op meer complexe computers kunnen nog andere voorstellingen worden gebruikt. Dit kan het een-complement zijn, maar dan zijn speciale circuits nodig om het resultaat te corrigeren.

Vanaf nu zullen alle gehele getallen in dit boek worden genoteerd in het twee-complement. In figuur 1.3 staat een tabel voor twee-complement getallen.

Oefening 1.10: Wat is het grootste en het kleinste getal dat we nog kunnen voorstellen in twee-complement, gebruik makende van 1 byte?

Oefening 1.11: Bereken het twee-complement van 20. Bereken dan het twee-complement van het antwoord. Is dat weer 20?

De volgende voorbeelden dienen ter illustratie van de regels van het twee-complement. C betekent een mogelijke carry of borrow. C is bit 8, dus het negende bit, van het resultaat. V betekent een overflow, d.w.z. V wordt 1, als het teken per ongeluk verandert omdat het resultaat te groot is. In feite is het een interne carry van bit 6 naar bit 7, het tekenbit. Dit zal verduidelijkt worden in het nu volgende.

De carry C

Een voorbeeld van een carry:

128	10000000
+129	10000001
=257 = (1)	00000001

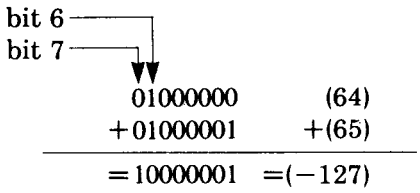
(1) is een carry.
Het resultaat heeft een negende bit nodig (dit zou bit 8 worden). Dit negende bit wordt het carrybit genoemd. Het resultaat is dus 100000001 = 257. De carry moet echter als zodanig herkenbaar zijn em we moeten er dus zeer zorgvuldig mee omspringen. De registers in de microprocessor zijn in het algemeen slechts 8 bits breed. Van het resultaat wordt alleen bit 0 tot en met bit 7 bewaard. De carry moet met een speciale instructie gedetecteerd, en op de juiste wijze verder behandeld worden. Dat kan bijvoorbeeld zijn: opbergen van de carry, de carry negeren, of besluiten dat er een fout is opgetreden (Dit laatste als het grootste toegestane getal 11111111 is).

+	2's complement code	−	2's complement code
+127	01111111	−128	10000000
+126	01111110	−127	10000001
+125	01111101	−126	10000010
. . .		−125	10000011
		. . .	
+65	01000001	−65	10111111
+64	01000000	−64	11000000
+63	00111111	−63	11000001
. . .		. . .	
+33	00100001	−33	11011111
+32	00100000	−32	11100000
+31	00011111	−31	11100001
. . .		. . .	
+17	00010001	−17	11101111
+16	00010000	−16	11110000
+15	00001111	−15	11110001
+14	00001110	−14	11110010
+13	00001101	−13	11110011
+12	00001100	−12	11110100
+11	00001011	−11	11110101
+10	00001010	−10	11110110
+9	00001001	−9	11110111
+8	00001000	−8	11111000
+7	00000111	−7	11111001
+6	00000110	−6	11111010
+5	00000101	−5	11111011
+4	00000100	−4	11111100
+3	00000011	−3	11111101
+2	00000010	−2	11111110
+1	00000001	−1	11111111
+0	00000000		

Fig. 1.3: twee-complement tabel

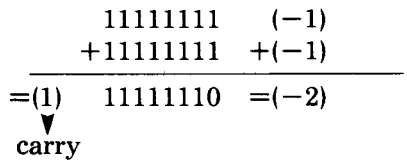
De overflow V

Een overflow treedt op in het volgende voorbeeld:

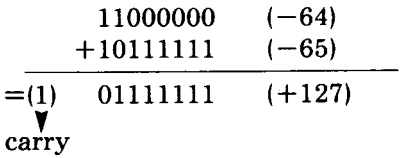


Een carry van bit 6 naar bit 7 is gegenereerd. Dit heet een overflow. Het resultaat is "per ongeluk" negatief geworden. Dit moet worden gedetecteerd om het resultaat te kunnen corrigeren.

We bekijken nu de volgende situatie:



Er komt tweemaal een carry voor: een van bit 6 naar bit 7 en een van bit 7 naar het carrybit. De rekenregels voor het twee-complement vertellen ons, dat we de carry van bit 7 naar het carrybit kunnen negeren. Het resultaat klop zo. De interne carry immers veranderde het teken niet. Er is dus geen sprake van een overflow. Bij het werken met negatieve getallen is een interne carry dus niet eenvoudig een overflow.
Nog een voorbeeld:



Er is geen interne, maar wel een externe carry. Het resultaat is fout, omdat bit 7 veranderd is. Hier is wel sprake van een overflow.

Een overflow kan voorkomen in de volgende situaties:

1–bij het optellen van grote positieve getallen,
2–bij het optellen van grote negatieve getallen,
3–bij het aftrekken van een groot positief van een groot negatief getal,
4–bij het aftrekken van een groot negatief van een groot positief getal.

We kunnen dus nu een betere definitie geven van een overflow:
Technisch bekeken is een overflow indicator een speciaal voor dit doel gereserveerd bit, een "vlag" genoemd, dat 1 gemaakt wordt, als er een carry plaats vindt van bit 6 naar bit 7, indien er geen externe carry is, of als er een externe carry is en geen interne. Deze vlag geeft aan, dat het tekenbit, bit 7, per ongeluk veranderd is. Voor de technisch geinteresseerde lezer: de overflow vlag is de uitgang van een exclusieve of van de carry-in en de carry-uit van bit 7. Praktisch iedere microprocessor is met een dergelijke vlag uitgerust.
Een overflow geeft aan, dat het resultaat van een optelling of een aftrekking meer bits nodig heeft dan de standaard 8 van een register.

De carry en de overflow

De carry en de overflow bits worden *vlaggen* genoemd. Ze komen voor in iedere microprocessor en in het volgende hoofdstuk zullen we ze leren gebruiken om effectief te kunnen programmeren. Deze twee indicatorbits zitten in een speciaal register, het vlag of "status" register. Dit register bevat nog meer indicatorbits, maar deze komen pas in hoofdstuk 4 aan de orde.

Voorbeelden

In de volgende voorbeelden wordt het gebruik van de carry en de overflow nog eens geillustreerd. V is overflow en C is carry. Is V=0, dan is er geen overflow. Als V=1 wel. Hetzelfde geldt voor C. Onthoudt goed, dat de rekenregels van het twee-complement aangeven, dat de carry genegeerd moet worden. (Het wiskundige bewijs daarvoor wordt hier niet geleverd).

Positief-positief

```
    00000110   (+6)
+   00001000   (+8)
-------------------
=   00001110  (+14)   V:0      C:0
```

Correct

Positief-positief met overflow

```
    01111111  (+127)
+   00000001    (+1)
--------------------
=   10000000  (-128)  V:1      C:0
```

Het resultaat is niet geldig, want er is een overflow.

Fout

Positief-negatief (positief resultaat)

```
    00000100  (+4)
+   11111110  (-2)
-----------------
=(1)00000010  (+2)    V:0      C:1 (negeren)
```

Correct

Positief-negatief (negatief resultaat)

```
    00000010  (+2)
+   11111100  (-4)
-----------------
=   11111110  (-2)    V:0      C:0
```

Correct

Negatief-negatief

```
    11111110  (-2)
+   11111010  (-4)
-----------------
=(1)11111010  (-6)    V:0      C:1 (negeren)
```

Correct

Negatief-negatief met overflow

```
    10000001  (-127)
+   11000010   (-62)
--------------------
=(1)01000011    (67)  V:1      C:1
```

Fout

Er treedt een overflow op, door twee grote negatieve getallen op te tellen. Het resultaat moet −189 zijn, wat te groot is voor 8 bits.

Oefening 1.12: Maak de volgende optellingen af. Noteer het resultaat, carry C, overflow V en geef aan of het resultaat klopt.

```
  10111111   (____)          11111010   (____)
+11000001    (____)        +11111001    (____)
-----------                -----------
= _______ V:____ C:____    = _______ V:____ C:____
☐ CORRECT    ☐ FOUT        ☐ CORRECT    ☐ FOUT

  00010000   (____)          01111110   (____)
+01000000    (____)        +00101010    (____)
-----------                -----------
= _______ V:____ C:____    = _______ V:____ C:____
☐ CORRECT    ☐ FOUT        ☐ CORRECT    ☐ FOUT
```

Vast formaat voorstelling

We weten nu hoe we gehele getallen met een teken kunnen voorstellen. Het grootte probleem is daarmee nog niet opgelost. Als grote getallen moeten worden voorgesteld, zijn meerdere bytes nodig. Het is echter efficienter om rekenkundige operaties uit te voeren met een vast aantal bytes, i.p.v. met een aantal dat kan varieren. Wanneer het aantal bytes is gekozen, is daarmee de maximale grootte van de getallen vastgelegd.

Oefening 1.14: Wat zijn de grootste en de kleinste getallen die te maken zijn met twee bytes in twee-complement voorstelling?

Het probleem van de grootte der getallen

We hebben ons bij het optellen beperkt tot getallen van acht bits, omdat dat het formaat is, waarop de processor intern bewerkingen uitvoerd. Daarmee zijn we ook beperkt tot getallen van −128 tot +127. Het is duidelijk dat dit voor veel toepassingen niet voldoende is.

Daarom wordt dubbele of zelfs meervoudige precisie gebruikt om meer cijfers voor te stellen. Twee, drie of N bytes zijn daarvoor te gebruiken. Bijvoorbeeld het 16 bits "dubbele precisie" formaat:

00000000	00000000	is "0"
00000000	00000001	is "1"
01111111	11111111	is "32767"
11111111	11111111	is "−1"
11111111	11111110	is "−2"

Oefening 1.15: Wat is het grootste negatieve getal in het twee-complement in het drie-voudige precisie formaat?

Er zijn nadelen verbonden aan deze methode. In het algemeen kunnen bij een optelling slechts acht bits per keer worden opgeteld. Dit wordt verder verklaard in hoofdstuk 3 (Basis programmeer technieken). Een volledige optelling is nu langzamer geworden. Een ander nadeel is dat ook kleine getallen 16 bits gebruiken i.p.v. 8. Daarom worden zelden meer dan 16 of 32 bits gebruikt.

Dan is er nog het volgende belangrijke punt: welk aantal bits we ook nemen (stel dit aantal N), dit aantal kan daarna niet meer veranderd worden. Als het resultaat van een berekening meer dan N bits nodig heeft, zullen bits verloren gaan. Normaal bewaart het programma de meest significante bits, en de minst significante worden weggegooid. Dit heet afkappen.

Ter illustratie een voorbeeld in het decimale stelsel, waarin we 6 cijfers gebruiken:

```
  123456
×   1,2
------
  246912
 1234560
------
=148147,2
```

Het resultaat heeft 7 cijferplaatsen nodig! De 2 achter de komma wordt weggegooid en het uiteindelijke resultaat is 148147. Het wordt dus afgekapt. Zolang de positie van de komma niet verloren gaat, is dit een gebruikelijke manier om het bereik van de operaties te vergroten. Dit gaat wel ten koste van de precisie.

In het binaire stelsel geldt het zelfde probleem. De vermenigvuldiging komt in hoofdstuk 4 aan de orde.

De vaste formaat voorstelling kan een verlies van de precisie tot gevolg hebben, maar voor gewone en wiskundige berekeningen kan deze voorstelling voldoen.

Helaas is het verlies van precisie niet toegestaan bij boekhoudkundige operaties. Stel je voor, dat ieder bedrag van 5 cijfers afgerond zou worden op hele guldens. Niemand accepteert dat.

Waar precisie essentieel is, moeten we iets anders verzinnen. De meest gebruikte oplossing is BCD (afkorting van het Engelse Binary Coded Decimal=binair gecodeerde decimale voorstelling).

BCD voorstelling

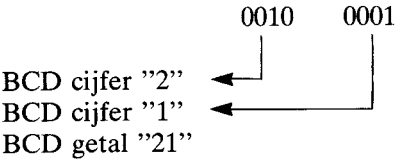
Het principe dat bij BCD codering gebruikt wordt is als volgt: codeer ieder decimaal cijfer apart en gebruik zoveel bits als nodig zijn om het getal precies voor te stellen. Om ieder van de cijfers 0 tot 9 te coderen zijn vier bits nodig. Met drie bits zouden maar acht combinaties mogelijk zijn, waarmee dus niet alle tien cijfers gecodeerd zouden kunnen worden. Met vier bits kunnen zestien combinaties gemaakt worden en dit is dus voldoende om om de cijfers "0" to "9" te coderen. Er moet opgemerkt worden dat in de BCD voorstelling zes van de mogelijke codes niet gebruikt worden (zie Fig 1-4). Dit heeft enkele grote problemen ten gevolge bij het optellen en het aftrekken waar we nog en op-

CODE	BCD SYMBOOL	CODE	BCD SYMBOOL
0000	0	1000	8
0001	1	1001	9
0010	2	1010	unused
0011	3	1011	unused
0100	4	1100	unused
0101	5	1101	unused
0110	6	1110	unused
0111	7	1111	unused

Fig. 1-4: BCD tabel

lossing voor zullen moeten vinden. Aangezien voor het coderen van ieder BCD-cijfer maar vier bits nodig zijn, kunnen er in ieder byte twee BCD cijfers gecodeerd worden. Dit noemen we "packed BCD". Zo is bijvoorbeeld "00000000" in BCD: "00". "10011001" wordt "9".

Een BCD code wordt als volgt gelezen:

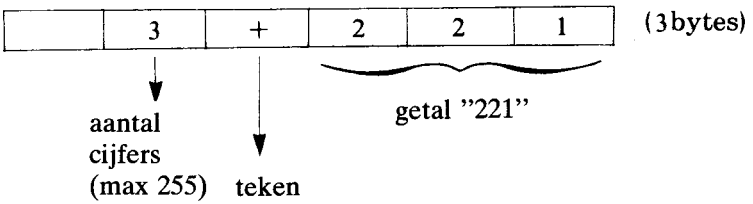


Oefening 1.16: Wat is de BCD voorstelling voor "29" en "91"?

Oefening 1.17: Is "10100000" een geldige BCD voorstelling.

Er worden zoveel bytes gebruikt als nodig zijn om alle BCD cijfers weer te geven. Vaak worden een of meer nibbles gebruikt aan het begin van de voorstelling om aan te geven hoeveel nibbles gebruikt worden, d.w.z. het totale aantal cijfers. Een ander nibble zal gebruikt worden om aan te geven wat de positie van de komma is. De afspraken kunnen echter per systeem verschillen.

Hier volgt een voorbeeld van gehele BCD getallen die uit meerdere bytes opgebouwd zijn:



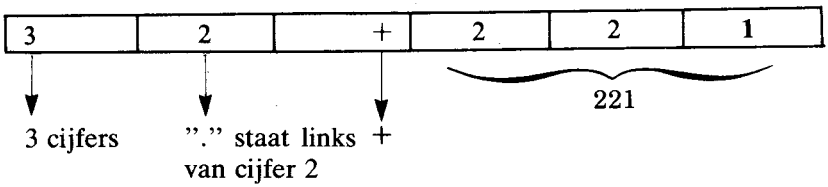
Dit stelt voor: +221
(Het teken kan voorgesteld worden door 0000 voor de + en 0001 voor de − bijvoorbeeld.)

Oefening 1.18: Schrijf, gebruik makend van bovenstaande afspraak, de voorstelling van "−23123". Eerst in BCD formaat, dan binair.

Oefening 1.19: Geef de BCD kode voor "111" en "222" en vervolgens voor het resultaat 222 × 111. (Bereken het resultaat met de hand en bepaal dan de voorstelling.)

Met de BCD voorstelling kunnen eenvoudig gebroken getallen weergegeven worden.

Bijvoorbeeld: +2.21 kan als volgt weergegeven worden:



Het voordeel van BCD voorstelling is dat de resultaten absoluut korrekt zijn. Het nadeel is dat er veel geheugen voor nodig is en dat de berekeningen traag zijn.

Oefening 1.20: Hoeveel bits zijn nodig om "9999" in BCD te koderen. En in twee-komplement?

We hebben nu de problemen opgelost die samenhangen met de voorstelling van gehele getallen, getallen met teken en zelfs van grote getallen. We hebben ook al een mogelijke methode gegeven om met de BCD voorstelling gebroken getallen weer te geven. Laten we nu eens het probleem onderzoeken van het weergeven van gebroken getallen in het vaste lengte formaat (fixed length format).

Vlottende komma voorstelling (Floating point representation)

Het uitgangspunt is dat gebroken getallen in een vast formaat weergegeven moeten worden. Om geen bits te verspillen normaliseert de voorstelling alle getallen.

Zo worden bijvoorbeeld bij "0.000123" drie nullen links van het getal verspild, die geen andere betekenis hebben dan de positie van de decimale punt aan te geven. Het resultaat van de normalisatie is .123 × 10 tot de macht −3. ".123" wordt de *genormaliseerde mantissa* genoemd, "−3" wordt de *exponent* genoemd. We hebben dit getal genormaliseerd door alle zinloze nullen links van het getal te verwijderen en de exponent aan te passen.

Laten we een ander voorbeeld nemen:

22.1 geeft genormaliseerd: .221 × 10 tot de macht 2

of M × 10 tot de macht E, waarbij M de mantissa is en E de exponent.

Het is meteen duidelijk, dat een genormaliseerd getal gekenmerkt wordt door een mantissa die kleiner is dan 1 en groter of gelijk is aan .1 in al die gevallen waarin het getal ongelijk is aan nul. Met andere woorden, het kan wiskundig voorgesteld worden door:

$$.1 \leqslant M < 1 \text{ of } 10^{-1} \leqslant M < 10^{0}$$

Overeenkomstig in de binaire voorstelling:

$$2^{-1} \leqslant M < 2^{0} \text{ (of } .5 \leqslant M < 1)$$

Waarbij M de absolute waarde is van de mantissa (negeer het teken).

Bijvoorbeeld:

$$111.01 \text{ wordt: } .11101 \times 2^{3}.$$

De mantissa is 11101.

De exponent is 3.

Nu we het principe van de voorstelling hebben behandeld kunnen we het eigenlijke formaat gaan bekijken. Een kenmerkende vlottende komma voorstelling staat hieronder:

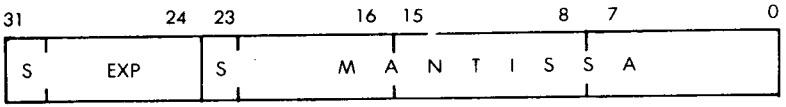


Fig. 1-5: Vlottende komma voorstelling

In de bovenstaande voorstelling worden vier bytes gebruikt voor totaal 32 bits. Het eerste byte links wordt gebruikt om de exponent weer te geven. Zowel de exponent als de mantissa worden in twee-komplement weergegeven. Dit heeft tot gevolg dat de maximale exponent −128 is. "S" in Fig. 1-5 geeft het teken aan.

Voor de mantissa worden drie bytes gebruikt. Aangezien het eerste bit in twee-komplement het teken aangeeft, blijven er 23 bits over voor de voorstelling van de grootte van de mantissa.

Oefening 1.21: Hoeveel decimale cijfers kan de mantissa weergeven met 23 bits?

Dit is slechts een voorbeeld van een vlottende komma voorstelling. Er zouden slechts drie bytes gebruikt kunnen worden of eventueel meer bytes. De vier byte voorstelling die we hierboven gebruikt hebben is slechts een veel voorkomende die een redelijk kompromis vormt in termen van nauwkeurigheid, grootte van de getallen geheugengebruik en effectiviteit bij de rekenkundige bewerkingen.

Voorstelling van Alfanumerieke gegevens

De voorstelling van alfanumerieke gegevens is erg eenvoudig: alle karakters worden in een acht bits kode gekodeerd. Er worden in computerland maar twee kodes algemeen gebruikt, de ASCII kode en de EBCDIC kode. ASCII betekent American Standard Code for Information Interchange, en wordt in de microprocessor wereld universeel toegepast. EBCDIC is een variant van ASCII die door IBM gebruikt wordt, en niet in een microcomputer, tenzij men een IBM terminal zou willen aansluiten.

Laten we de ASCII kode even kort bekijken. We moeten de 26 leters van het alfabet koderen, zowel kleine als grote letters, plus 10 numerieke symbolen en verder nog zo'n 20 andere symbolen. Dit kan gemakkelijk met 7 bits, waarmee 128 kodes mogelijk zijn. (Zie figuur 1-6.) Alle karakters worden dus in 7 bits gekodeerd. Het achtste bit wordt eventueel voor de *pariteit* gebruikt. Pariteit is en techniek om te kontroleren of de inhoud van een byte niet per ongeluk is veranderd. Het aantal enen in een byte wordt geteld en het achtste bit wordt dan zo gezet, dat de som even is. Dit wordt even pariteit genoemd. Er kan ook oneven pariteit gebruikt worden.

Voorbeeld: Laten we de pariteit berekenen voor "0010011" met even pariteit. Het aantal enen is drie. Het pariteitsbit moet dus 1 worden zodat het totale aantal enen even wordt (4). Het resultaat is 10010011, waarbij de eerste 1 het pariteitsbit is en 0010011 identificeert het karakter.

In Fig 1-6 wordt de tabel met 7 bits ASCII kodes getoond.
Zonder pariteitsbit.

Oefening 1.22: Bereken de 8 bits voorstelling van de cijfers "0" tot "9" met even pariteit. (Deze kode wordt gebruikt in voorbeelden van hoofdstuk 8.)

Oefening 1.23: Doe hetzelfde voor de letter A t/m F.

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

Fig. 1-6: ASCII konversie tabel

(Zie appendix E voor de afkortingen)

In speciale toepassingen zoals telekommunikatie kunnen andere kodes gebruikt worden zoals foutherstellende kodes. Dit gaat echter buiten het bestek van dit boek.

We hebben nu de gebruikelijke voorstellingen van zowel programma's als gegevens behandeld. Laten we nu de mogelijke externe voorstellingen eens beschouwen.

EXTERNE VOORSTELLING VAN INFORMATIE

Met externe voorstelling van informatie bedoelen we de wijze waarop informatie aan de gebruiker wordt voorgesteld. Informatie wordt in essentie op drie manieren weergegeven: binair, octaal of hexadecimaal en symbolisch.

1 - *Binair*

We hebben gezien dat informatie intern in bytes opgeslagen wordt. Bytes zijn series van 8 bits (nullen en enen). Soms is het wenselijk om informatie op deze interne informatie direct in dit binaire formaat weer te geven en dat noemen we dan de *binaire voorstelling*. Bijvoorbeeld door een aantal LED's (Licht Emitterende Dioden, een soort miniatuur lampjes) op het voorpaneel van de computer. Bij acht bits processors zal het voorpaneel meestal uitgerust zijn met 8 LED's om de inhoud van interne registers weer te geven. (Een register wordt gebruikt om 8 bits informatie te bevatten en wordt beschreven in hoofdstuk 2.) Een brandende LED geeft dan een "1" aan. Een dergelijke voorstelling wordt gebruikt om op een laag niveau fouten te zoeken in ingewikkelde programma's, maar is natuurlijk voor menselijke gebruikers niet erg praktisch. Een "9" is duidelijker te herkennen dan "1001". Er zijn betere voorstellingen ontwikkeld waarmee het mens-machine interface verbeterd is.

2 - *Octaal en hexadecimaal*

"Octaal" en "hexadecimaal" gebruiken resp. 3 en 4 bits om een uniek symbool weer te geven. In de octale kode worden alle acht kombinaties van de drie bits weergegeven door de cijfers 0 tot 7:

binair	octaal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Fig. 1-7: Octale symbolen

Bijvoorbeeld, "00 100 100" binair wordt voorgesteld door:

0 4 4

of wel "044" octaal.

Een ander voorbeeld, 11 111 111 is:

3 7 7

of wel "377" octaal.
Omgekeerd stelt octaal "211" het volgende binaire getal voor:
010 001 001

of wel "100001001".

Octaal werd doorgaans toegepast op oudere computers waar verschillende aantallen bits varierend van 8 tot 64 gebruikt werden. Tegenwoordig is met de opkomst van de 8 bits microprocessor het acht bits formaat standaard geworden, en werd een andere voorstelling gebruikt. De *hexadecimale*.

Bij de hexadecimale voorstelling wordt een groep van vier bits gekodeerd in een hexadecimaal cijfer. Hexadecimale cijfers worden gevormd door de symbolen 0 t/m 9 en de letters A t/m F.
Bijvoorbeeld: "0000" geeft "0" weer, "0001" geeft "1" weer en "1111" geeft de letter "F" weer. (Zie Fig. 1-8.)

DECIMAAL	BINAIR	HEX	OCTAAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Fig. 1-8: Hexadecimale Codes

Voorbeeld: $\underbrace{1010}\ \underbrace{0001}$ binair geeft

A 1 hexadecimaal

Oefening 1.25: Wat is de hexadecimale voorstelling van "10101010"?

Oefening 1.26: Wat is het binaire equivalent van hexadecimaal "FA"?

Oefening 1.27: Wat is octaal voor "01000001"?

Het voordeel van hexadecimale kodering is dat acht bits in slechts twee cijfers gekodeerd worden. Dit kan veel eenvoudiger onthouden worden en ingevoerd worden in de computer dan het binaire equivalent. Daarom wordt bij de meeste nieuwe microcomputers aan deze notatie de voorkeur gegeven voor het voorstellen van groepen bits.

Het is natuurlijk wel zo, dat als de informatie in het geheugen en bepaalde betekenis heeft zoals het voorstellen van tekst of getallen, de hexadecimale notatie niet praktisch is om de betekenis aan menselijke gebruikers over te brengen.

Symbolische voorstelling

Met symbolische voorstelling van informatie doelen we op de externe voorstelling van informatie in de eigenlijke symbolische vorm. Zo worden decimale getallen bijvoorbeeld voorgesteld als decimale getallen en niet als een serie hexadecimale cijfers of bits. Zo wordt ook tekst als zodanig voorgesteld. Het spreekt vanzelf dat symbolische voorstelling voor de menselijke gebruiker het meest praktisch is.. Het wordt toegepast als er een geschikt apparaat beschikbaar is om de informatie weer te geven, zoals een CRT display (Cathode Ray Tube = Kathode Straal Buis, een televisieachtig scherm om tekst of grafische voorstellingen weer te geven.) Helaas is het in de meeste kleine microcomputers niet economisch verantwoord om in een dergelijk display te voorzien en moet de gebruiker zich beperken tot hexadecimale communicatie met de computer.

Samenvatting van Externe voorstellingen

Symbolische voorstelling is de meest wenselijke vorm aangezien deze het natuurlijkst is voor de menselijke gebruiker. Er is echter wel een kostbaar interface voor nodig in de vorm van een alfanumeriek toetsenbord en een drukker (printer) voor nodig. Het is om deze reden

doorgaans niet beschikbaar op kleine systemen. Dan wordt een alternatieve voorstelling toegepast, en wel de hexadecimale. Slechts in enkele gevallen, als op laag niveau fouten opgespoord moeten worden wordt de de binaire voorstelling toegepast. In dit geval wordt de inhoud van registers meteen in binair formaat weergegeven.

We hebben nu gezien hoe we informatie intern en extern kunnen weergeven. We zullen nu onderzoeken hoe de eigenlijke processor deze informatie manipuleert.

Nog enkele opgaven

Oefening 1.28: Wat is het voordeel van twee-komplement boven andere notaties om getallen met teken weer te geven?

Oefening 1.29: Hoe zou je "1024" in binair weergeven? En hoe in binair met teken en in twee-komplement?

Oefening 1.30: Wat is het V-bit? Moet de programmeur dit bit testen na een optelling of een aftrekking?

Oefening 1.31: Bereken het twee-komplement van "+16", "+17", "+18", "−16", "−17" en "−18".

Oefening 1.32: Bepaal de hexadecimale voorstelling van de volgende tekst die intern in ASCII formaat zonder pariteit is opgeslagen: "MESSAGE".

2
6502 HARDWARE ORGANISATIE

INLEIDING

Om op een elementair niveau te programmeren is het niet nodig om de interne struktuur van de processor die gebruikt wordt tot in detail te kennen. Om echter efficient te programmeren is een dergelijke kennis wel vereist. Het is het doel van dit hoofdstuk om de basis hardware begrippen te presenteren die nodig zijn om de werking van de 6502 te begrijpen. Het volledige microcomputer systeem omvat niet alleen de microprecesor (in dit geval de 6502), maar ook andere komponenten. In dit hoofdstuk zullen we alleen de 6502 presenteren, andere komponenten (voornamelijk invoer/uitvoer komponenten) zullen in Hoofdstuk 7 besproken worden.

We zullen eerst de algemene architektuur van de microcomputer beschouwen en vervolgens meer in detail de interne organisatie van de 6502 bestuderen. We zullen in het bijzonder de verschillende registers van de 6502 onderzoeken. Dan zullen we het mechanisme onderzoeken waarmee programma's uitgevoerd worden en het mechanisme waarmee de volgorde bepaald wordt waarin het programma uitgevoerd wordt. Wat de hardware betreft geeft dit hoofdstuk slechts een vereenvoudigde weergave. De lezer die in meer details geinteresseerd is wordt verwezen naar ons boek C201 ("From Chips to Systems: An Introduction to Microprocessing", door dezelfde auteur).

SYSTEEM ARCHITEKTUUR

De architektuur van het microcomputersysteem is te zien in figuur

2-1. De *microprocessor eenheid* (MPE), in ons geval een 6502, is links in de illustratie te zien. Hij verzorgt de funkties van een *centrale verwerkings eenheid* (CVE) op een enkele chip: deze omvat een *rekenkundige logische eenheid* (RLE), samen met de interne registers, en een *controle eenheid* (CE) die ervoor zorgt dat binnen het systeem alles in de juiste tijdsvolgorde gebeurt. De werking wordt in dit hoofdstuk uiteen gezet.

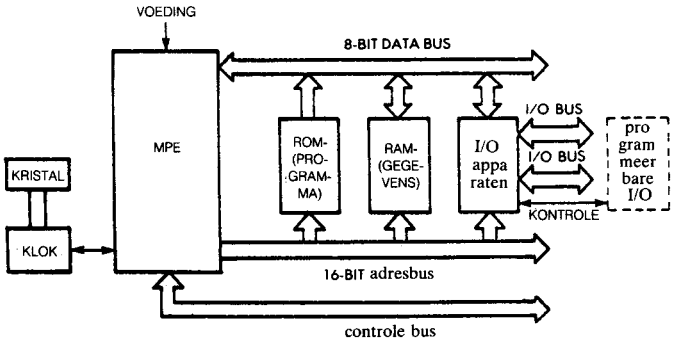


Fig.2-1: Architektuur van een Standaard Microprocessor Systeem

Uit de MPE komen drie *bussen*: een 8 bits bi-directionele (twee richtingen) *databus*, die bovenin de figuur te zien is, een 16 bits eenrichtings *adresbus* en een *controlebus* die onderin te figuur te zien is. We zullen nu de funktie van ieder van de *bussen* beschrijven.

De *databus* transporteert de gegevens (data) die uitgewisseld worden tussen de verschillende elementen van het systeem. In de meeste gevallen worden gegevens vervoerd van het geheugen naar de MPE, van de MPE naar het geheugen of van de MPE naar een invoer/uitvoer chip (Input/Output chip= I/O chip). Via een I/O chip kan met een extern apparaat gekommuniceerd worden.

De *adresbus* vervoerd de adressen die door de MPE gegenereerd worden, waarmee een intern register van een van de met het system verbonden chips geselekteerd wordt. Dit adres specificeert de bron of de bestemming van de gegevens die over de databus vervoerd worden.

De *controlebus* vervoert de diverse signalen die nodig zijn om het systeem te synchroniseren.

Nu we de funktie van de bussen gegeven hebben kunnen we de verdere komponenten aansluiten die voor een volledig systeem nodig zijn.

Iedere MPE heeft een nauwkeurige tijdrefenrentie nodig, die verzorgd wordt door een *klok* en een *kristal*. In de meeste "oudere" microprocessoren is de klokoscillator extern en is hier een aparte chip voor nodig. Bij de meeste recente microprocessoren is de oscillator ingebouwd in de MPE. Het kwarts kristal wordt wegens de afmetingen altijd buiten de MPE geplaatst. Het kristal en de klok zijn in de figuur links van de MPE getekend.

Laten we nu onze aandacht eens richten op de andere komponenten in het systeem. Van links naar rechts kunnen we in de figuur onderscheiden:

ROM (Read Only Memory), is het lees-alleen geheugen en bevat het programma van het systeem. Het voordeel van ROM is dat de inhoud permanent is en niet verdwijnt als de spanning uitgeschakeld wordt. Het ROM bevat dus altijd een *bootstrap* of *monitor* programma (de funktie van deze programma's wordt later uitgelegd) waarmee de eerste opstart van het systeem mogelijk is. Bij industriele toepassingen als proceskontrole zijn bijna alle programma's doorgaans in ROM opgeslagen omdat deze waarschijnlijk weinig meer veranderd worden. In die gevallen moet de gebruiker het syteem beschermen tegen spanningstoringen: programma's mogen niet vluchtig zijn. Ze moeten in ROM opgeslagen zijn.

Bij hobbyisten of in een omgeving waar programma's uitgetest worden zullen de programma's in RAM opgeslagen zijn zodat ze eenvoudig veranderd kunnen worden. Later kunnen ze of in RAM blijven of eventueel naar ROM geheugen overgebracht worden. RAM is echter vluchtig. De inhoud gaat verloren als de spanning uitgeschakeld wordt.

RAM (Random Acces Memory) is het lees/schrijf geheugen van het systeem. Bij kontrole systemen is de hoeveelheid RAM meestal klein (alleen gegevens). In een omgeving waar programma's ontwikkeld worden echter zal de hoeveelheid RAM groot zijn, omdat het zowel programma als ontwikkelings software zal bevatten. De inhoud van RAM moet voor gebruik van een extern apparaat geladen worden.

Tenslotte zal het systeem enkele interface chips bevatten zodat met de buitenwereld gekommuniceerd kan worden. De meest gebruikte chip is de *PIO* of wel de Parallel Input Output chip. Deze is in de figuur te zien. Deze PIO is net als alle andere chips in het systeem met de drie bussen verbonden en voorziet in tenminste twee 16 bits *poorten* voor kommunikatie met de buitenwereld. Zie voor meer details van de wer-

king van een PIO het boek C201 of anders voor kenmerken van het 6502 systeem, hoodstuk 7 (Invoer/Uitvoer komponenten.)

Al deze chips zijn met de drie bussen verbonden, inclusief de controlebus. Om de tekening wat duidelijker te houden zijn de verbindingen tussen de kontrolebus en de komponenten niet getekend.

De funktionele eenheden die beschreven zijn hoeven niet noodzakelijkerwijs op een chip geplaatst te zijn. We zullen in feite doorgaans *kombinatiechips* gebruiken, waarin zowel een PIO als een kleine hoeveelheid ROM en/of RAM zit. Zie voor meer details hoofdstuk 7.

Om een echt systeem te bouwen hebben we nog meer komponenten nodig. In het bijzonder moeten de bussen *gebufferd* worden. Verder is er nog wat *dekodering* nodig voor de RAM chips en tenslotte kan het nodig zijn dat enkele signalen versterkt worden door *drivers*. Deze afzonderlijke chips zullen hier niet beschreven worden en zijn ook niet van belang voor de programmeur. De geinteresseerde lezer wordt verwezen naar boek C207 ("Microprocessor Interfacing Techniques").

INTERNE ORGANISATIE VAN DE 6502

Een vereenvoudigd diagram van de interne organisatie van de 6502 is te zien in Fig. 2-2.

De rekenkundige logische eenheid (RLE) staat rechts in de illustratie. Deze kan gemakkelijk herkend worden aan de kenmerkende "V" vorm. De funktie van de RLE is het verrichten van rekenkundige en logische bewerkingen op de gegevens die via de twee invoerpoorten aangevoerd worden. De twee invoerpoorten van de RLE zijn de linker en de rechter invoerpoort. Ze vormen de bovenste uiteinden van de "V" vorm. Na het verrichten van een rekenkundige bewerking zoals en optelling of een aftrekking voert de RLE zijn inhoud aan het onderste uiteinde uit.

De RLE is uitgerust met een speciaal register, de *accumulator* (A). De accumulator is verbonden met de linker invoer. De RLE zal deze accumulator automatisch als een van de invoeroperanden gebruiken. (Dit kan echter omzeild worden.) Dit is een klassiek op een accumulator gebaseerd ontwerp. Bij rekenkundige en logische bewerkingen zal een van de operanden de accumulator zijn en de andere zal in de meeste gevallen een geheugenplaats zijn. Het resultaat wordt in de accumulator geplaatst. Het gebruik van de accumulator voor zowel de bron als de bestemming van gegevens is de reden van de naam: hij verzamelt (accumuleert) resultaten. Het voordeel van deze aanpak is dat er zeer

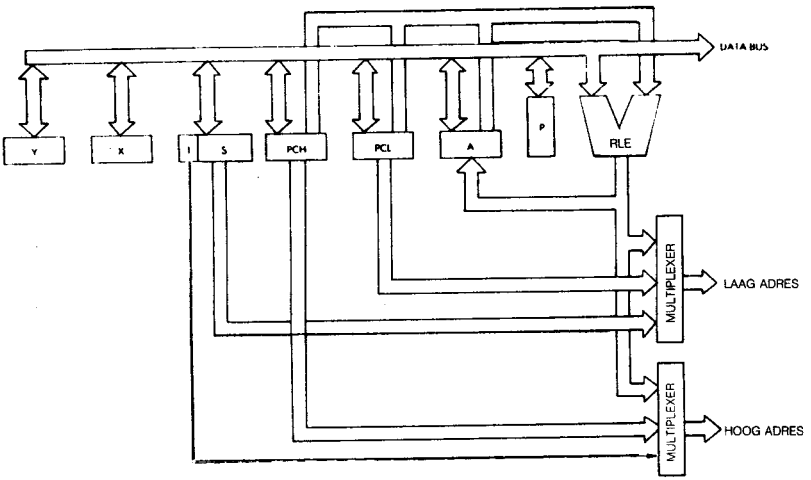


Fig.2-2: Interne Organisatie van de 6502

korte instructies gebruikt kunnen worden, slechts een enkel byte (8 bits) om de "opcode" aan te geven, d.w.z. de aard van de te verrichten bewerking.
Als de operand uit een van de andere registers (anders dan de accumulator) gehaald had moeten worden, waren er in de instruktie extra bits nodig om dit register aan te geven. De accumulator architektuur resulteert daarom in een hogere snelheid waarmee programma's uitgevoerd worden. Het nadeel is dat de accumulator voor gebruik altijd met de gewenste gegevens geladen moet worden. Dit kan soms inefficient zijn.

Laten we teruggaan naar de illustratie. Links van de RLE staat een speciaal 8-bits register, de *statusvlaggen* (P) van de processor. Dit register bevat 8 statusbits. Ieder van deze bits, fysisch uitgevoerd als een flip-flop in het register, wordt gebruikt om een bijzondere konditie aan te geven. De funktie van de verschillende statusbits zal uitgelegd worden in de loop van de programmeervoorbeelden van het volgende hoofdstuk, en zullen volledig beschreven worden in hoofdstuk 4, waar de instruktie set behandeld wordt. Als voorbeeld: drie van deze vlaggen zijn de N, Z en C bits.

N staat voor "negatief". Het is bit 7 (het meest linkse) van register P. Als dit bit 1 is betekent dit dat het resultaat van de bewerking van de RLE negatief is.

Z staat voor "zero" (nul). Als dit bit 1 is werd een nul-resultaat verkregen.

Bit C, op de meest rechtse positie (positie 0), is het *overdrachtbit*. Als twee 8 bits getallen opgeteld worden en het resultaat past niet in 8 bits, vormt het C bit het negende bit van het resultaat. De carry wordt bij rekenkundige berekeningen intensief gebruikt.

Deze statusbits worden automatisch gezet door de verschillende instrukties. Een volledige lijst met de instrukties en de wijze waarop ze de statusbits van het systeem beinvloeden is gegeven in Appendix A en in hoofdstuk 4. Deze bits worden door de programmeur gebruikt om op verschillende speciale of uitzonderlijke kondities te testen of om snel op foutieve resultaten te testen. Zo kan bijvoorbeeld het Z bit met speciale instrukties getest worden en kan meteen bepaald worden of het resultaat van een bewerking nul was of niet. Alle beslissingen die in een assembleertaal programma, dus ook in alle programma's die in dit boek behandeld worden, zullen gebaseerd zijn op het testen van deze bits. Deze bits zijn of bits die van buiten het systeem ingelezen worden, of de statusbits van het systeem. Het is dus erg belangrijk dat de funktie en het gebruik van deze statusbits goed begrepen wordt. De RLE hier is uitgerust met een statusregister dat deze bits bevat. Alle andere invoer/uitvoer chips zijn ook uitgerust met statusbits. Dzeze zullen we in hoofdstuk 7 bestuderen.

Laten we nu eens links van de RLE kijken in figuur 2-2. De horizontale rechthoeken stellen de interne registers van de 6502 voor.

PC is de *programmateller* (Program Counter). Het is een 16 bits register en is fysisch uitgevoerd als twee 8 bits registers: PCL en PCH. PCL bevat de laagste helft van de programmateller, d.w.z. bits 0 t/m 7. PCH bevat het hoge deel van de programmateller, bits 8 t/m 15. De programmateller is een 16 bits register dat het adres bevat van de volgende instruktie die uitgevoerd moet worden.. Iedere computer is uitgerust met een programmateller zodat hij weet welke instruktie als volgende uitgevoerd moet worden. We zullen nu in het kort het mechanisme bespreken waarmee toegang tot het geheugen wordt verkregen om de rol van de programmateller te illustreren.

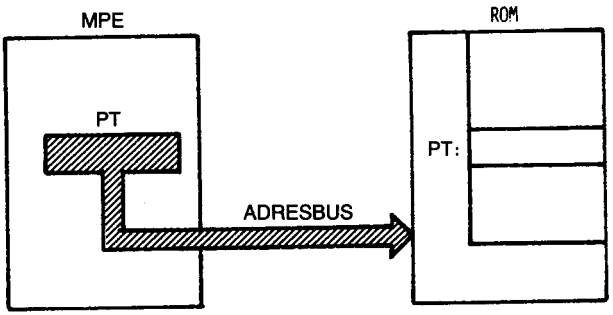


Fig.2-3: Het ophalen van een instruktie uit het geheugen

DE INSTRUKTIE EXECUTIE CYCLUS

Zie figuur 2-3. De microprocessor-eenheid is links getekend, het geheugen rechts. De geheugenchip kan ROM of RAM zijn, dan wel een andere chip waar geheugen in zit. Het geheugen wordt gebruikt om programma en gegevens in op te slaan. Hier zullen we een instruktie uit het geheugen halen om de funktie van de programmateller te illustreren. We nemen aan dat de programmateller een geldige inhoud heeft. Hij bevat nu een 16 bits adres dat het adres is van de instruktie die uit het geheugen gehaald moet worden. Iedere processor doet dit in drie stappen:

1 - Haal de volgende instruktie op
2 - Decodeer de instruktie
3 - Voer de instruktie uit

Ophalen

Laten we het rijtje afgaan. In de eerste cyclus wordt de inhoud van de programmateller op de adresbus geplaatst en doorgeschakeld naar het geheugen (via de adresbus). Eventueel kan tegelijkertijd een leessignaal op de controlebus gegeven worden. Het geheugen ontvangt het adres. Dit adres wordt gebruikt om een plaats in het geheugen te speci-

ficeren. Bij het ontvangen van het leessignaal zal het geheugen het ontvangen adres dekoderen via interne dekoders en de door het adres gespecificeerde locatie selekteren. Een paar honderd nanoseconden later zal het geheugen het 8 bits gegeven dat overeenkomt met het gespecificeerde adres op de databus plaatsen. Dit 8 bits woord is de instruktie die we op wilden halen.

Laten we in het kort de volgorde samenvatten. De inhoud van de programmateller wordt op de adresbus geplaatst. Er wordt een leessignaal gegenereerd. Zo'n 300 nanoseconden later plaatst het geheugen de gespecificeerde instruktie op de databus. De microprocessor leest dan de databus en plaatst de inhoud in een speciaal register, het IR register. Dit IR register is het *instruktie register*. Het is 8 bits breed en wordt gebruikt om de instruktie te bevatten die net opgehaald is. De ophaalcyclus is nu volledig. De 8 bits van de instruktie zijn nu fysisch opgeslagen in het speciale interne register van de 6502, het IR register. Het IR register is links in figuur 2-4 getekend.

Dekoderen en Uitvoeren

Als de instruktie eenmaal in de IR opgeslagen is, zal de controle eenheid de inhoud dekoderen en in staat zijn om de juiste interne en externe signalen te genereren die nodig zijn om de instruktie uit te voeren. Daarom is er een korte dekodeervertraging gevolgd door een uitvoerfase, waarvan de lengte afhangt van de instruktie die gespecificeerd is. Sommige instrukties worden geheel binnen de MPE uitgevoerd. Andere instrukties halen gegevens uit het geheugen of brengen gegevens naar het geheugen. Daarom hebben de instrukties van de 6502 verschillende tijden nodig om uitgevoerd worden. Deze tijdsduur wordt uitgedrukt in aantallen klokcycli. Zie Appendix A voor het aantal cycli dat vor de verschillende instrukties nodig is. De meeste 6502's hebben een 1 megahertz klok. De lengte van iedere cyclus is dus 1 microseconde. Omdat bij verschillende komponenten verschillende klokfrequenties gebruikt worden, wordt de uitvoersnelheid doorgaans in aantallen cycli opgegeven in plaats van in aantallen nanoseconden.

Bij de 6502 is de *klok* intern, voorgesteld door de interne oscillator (zie Fig 2-1).

Het ophalen van de volgende instruktie

We hebben nu beschreven hoe gebruik makend van de programmateller een instruktie uit het gheugen opgehaald kan worden. Tijdens de uitvoer van een programma worden de instrukties in een *bepaalde volgorde* uit het geheugen gehaald. Er moet dus een automatisch mechanisme zijn dat de instrukties in volgorde ophaalt. Deze funktie wordt uitgevoerd door een eenvoudige opteller die verbonden is met de programmateller. Dit is geillustreerd in figuur 2-4. Steeds als de inhoud van de programmateller (onderin de tekening) op de adresbus geplaatst wordt, wordt de inhoud met één verhoogd en teruggeschreven in de programmateller. Als de programmateller bijvoorbeeld de waarde 0 bevatte, dan zou de waarde nul op de adresbus geplaatst worden. Vervolgens zou de inhoud van de programmateller vermeerderd worden en de waarde 1 zou weer in de programmateller terugeschreven worden. Op deze wijze wordt als de programmateller weer gebruikt wordt de instruktie van adres 1 opgehaald. We hebben zojuist een automatisch mechanisme uitgevoerd voor het op volgorde zetten van instrukties.

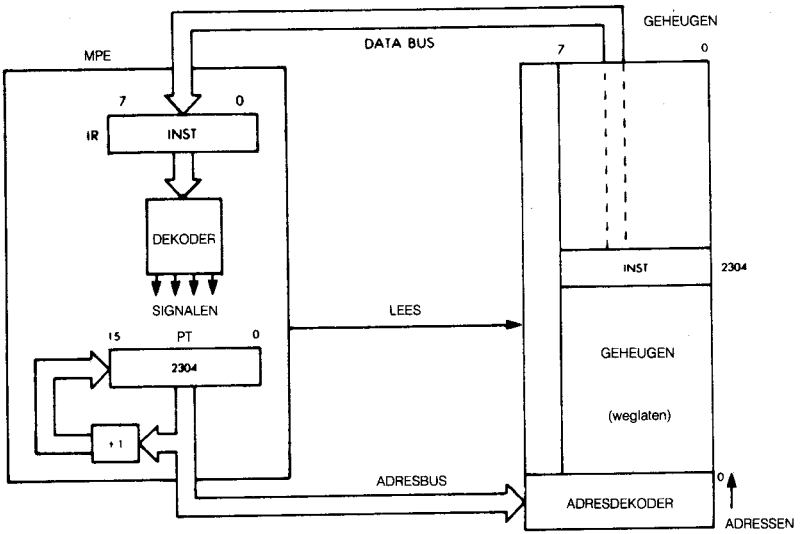


Fig. 2-4: Automatische bepaling van de volgorde

Er moet op gewezen worden, dat bovenstaande beschrijving sterk vereenvoudigd is. In werkelijkheid kunnen instrukties 2 of zelfs 3 bytes lang zijn, zodat op deze manier opeenvolgende bytes uit het geheugen gehaald worden. Het mechanisme is echter overeenkomstig. De programmateller, samen met de opteller voorziet in een automatisch mechanisme om opeenvolgende geheugenplaatsen te adresseren.

Andere 6502 registers

Een laatste deel van figuur 2-2 is nog niet uitgelegd. Het is de verzameling registers die met X, Y en S is gemerkt. Ze kunnen gebruikt worden om gegevens te bevatten waarop het programma bewerkingen uitvoert. Ze worden echter doorgaans als indexregisters gebruikt.

De funktie van de indexregisters wordt beschreven in hoofdstuk 5 bij de adresseertechnieken. In het kort: de inhoud van de indexregisters kan op verschillende manieren bij een gespecificeerd adres in het systeem opgeteld worden om zo een automatische verschuiving (offset) te verkrijgen. Dit is een belangrijke eigenschap om efficient toegang tot gegevens te krijgen als die opgeslagen zijn in tabellen. Deze twee registers zijn niet geheel symmetrisch. Het verschil zullen we in het hoofdstuk over adresseertechnieken uitleggen.

Het stapelregister S wordt gebruikt om een wijzer naar de top van het stapelgebied in het geheugen te bevatten.

We zullen nu het formele begrip van de stapel introduceren.

DE STAPEL

Een stapel wordt formeel een LIFO struktuur genoemd (Last In, First Out - Laatst In, Eerst Uit). Een stapel is een verzameling registers of geheugenplaatsen die toegewezen zijn aan een gegevensstruktuur. Het wezenlijke kenmerk van deze struktuur is dat het een *chronologische* struktuur is. Het element dat het eerst in de stapel geintroduceerd is bevindt zich op de bodem van de stapel. Het element dat het laatst op de stapel geplaatst is bevindt zich bovenop de stapel. Er is een analogie met een stapel borden op de balie van een restaurant. In de balie zit een gat met op de bodem een veer. In dit gat zijn de borden opgestapeld. Met deze organisatie wordt gegarandeerd, dat het bord dat het eerst op de stapel geplaatst is zich onderaan bevindt. Het meest recent op de stapel geplaatste bord bevindt zich bovenaan. Dit voorbeeld illustreert nog een ander kenmerk van de stapel. Bij normaal gebruik is de stapel alleen toegankelijk via twee instrukties: "PUSH" en "POP" (of

PULL). De push bewerking resulteert in het plaatsen van een element bovenop de stapel. De pull operatie verwijdert een element van de top van de stapel. In de praktijk wordt bij de meeste microprocessoren de accumulator op de stapel geplaatst. De pop operatie plaatst dan de inhoud van de top van de stapel in de accumulator.

De beschikbaarheid van een stapel is vereist om drie programmeerfaciliteiten uit te voeren in het computersysteem: subroutines, interrupts en tijdelijke gegevensopslag. De funktie van de stapel bij subroutines wordt uiteengezet in hoofdstuk 3. De funktie van de stapel bij interrupts wordt uiteengezet in hoofdstuk 6. Tenslotte zal tijdens enkele programmeervoorbeelden de funktie van de stapel bij tijdelijke gegevensopslag duidelijk gemaakt worden.

We zullen nu maar gewoon aannemen dat de stapel een vereiste is in ieder computersysteem.

Een stapel kan op twee manieren uitgevoerd zijn:

1 - Er kan een vast aantal registers voor gereserveerd zijn in de microprocessor zelf. Dit wordt een "hardware" stapel genoemd. Het voordeel is de hoge snelheid. Het beperkte aantal registers is een nadeel.

2 - De meeste microprocessoren voor algemeen gebruik kiezen een andere aanpak, de software stapel, om zo niet de lengte van de stapel te beperken tot een klein aantal registers. Deze aanpak is in de 6502 gekozen.. Bij de software benadering wordt in een speciaal register in de microprocessor, in dit geval het S register, de stapelwijzer opgeslagen, d.w.z. het adres van het bovenste element van de stapel (of om precies te zijn het adres van het bovenste element, vermeerderd met een). De stapel is dan uitgevoerd als een deel van het geheugen. De stapelwijzer (stackpointer) moet daarom 16 bits breed zijn om iedere plaats in het geheugen aan te kunnen wijzen.

Bij de 6502 is de stapelwijzer echter beperkt tot 8 bits. Hij bevat een negende bit op de meest linkse positie, dat echter altijd op 1 gezet is. Met andere woorden, het gebied dat aan de stapel toegewezen is bij de 6502 reikt van adres 256 tot adres 511. In binair is dit "100000000" to "111111111". De stapel begint altijd op adres 111111111 en kan 255 woorden lang zijn. Dit kan gezien worden als een beperking van de 6502, en zal verderop in dit boek besproken worden. Bij de 6502 begint de stapel op een hoog adres en groeit "achteruit"; de stapelwijzer wordt verminderd door een PUSH.

Om de stapel te gebruiken hoeft de programmeur alleen maar het S register te initialiseren. De rest gaat automatisch.

We zeggen dat de stapel in *pagina 1* van het geheugen staat. We zullen nu het begrip *paginering* introduceren.

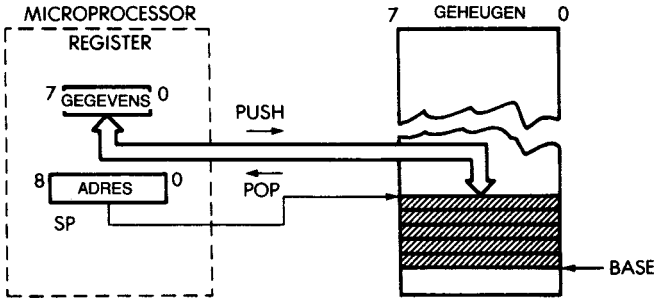


Fig. 2-5: De twee stapel instrukties

HET BEGRIP PAGINERING

De 6502 microprocessor is uitgerust met een 16 bits adresbus. Met 16 bits kunnen 64k kombinaties gevormt worden (1K = 1024). Wegens de adresseringskenmerken van de 6502 is het handig om het geheugen in *logische pagina's* in te delen. Een pagina is gewoon een blok van 256 woorden. Zo vormen geheugenplaatsen 0 t/m 255 pagina 0 van het geheugen. Deze wordt gebruikt voor pagina 0 adressering. Pagina 1 van het geheugen bevat locaties 256 t/m 511. We hebben net vastgesteld dat deze pagina doorgaans voor de stapel gereserveerd is. Aan de andere pagina's zijn geen beperkingen opgelegd door het ontwerp en ze kunnen vrij gebruikt worden. Het is bij de 6502 belangrijk om de gepagineerde organisatie van het geheugen in gedachten te houden. Steeds als een paginagrens overschreden moet worden, zal er een extra vertraging optreden in de executie van een instruktie.

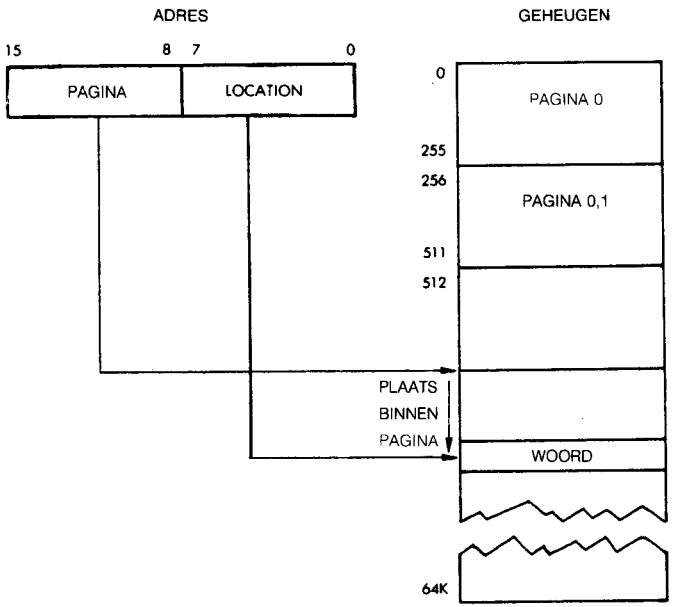


Fig. 2-6: Het begrip paginering

DE 6502 CHIP

Om de beschrijving van het diagram volledig te maken: de databus bovenin figuur 2-2 representeert de externe databus. Deze wordt gebruikt om te kommuniceren met externe apparaten, en het geheugen in het bijzonder. A0-7 en A8-15 stellen resp. het lage en het hoge gedeelte van de adresbus van de 6502 voor.

Voor de volledigheid geven we hier de eigenlijke pen aansluitingen van de 6502. Je hoeft dit niet te lezen om de rest van dit boek te begrijpen. Als je echter van plan bent om apparaten aan te sluiten zal deze beschrijving waardevol zijn.

De eigenlijke pen aansluitingen van de 6502 is gegeven in figuur 2-7. De databus is gemerkt DB0-7 en is gemakkelijk te herkennen rechts in de illustratie. De adresbus is gemerkt A0-11 en A12-15. Deze zit op pennen 9 tot 20 links van de chip en pennen 22 tot 25 rechts.

De rest van de signalen zijn voedings en controle signalen.

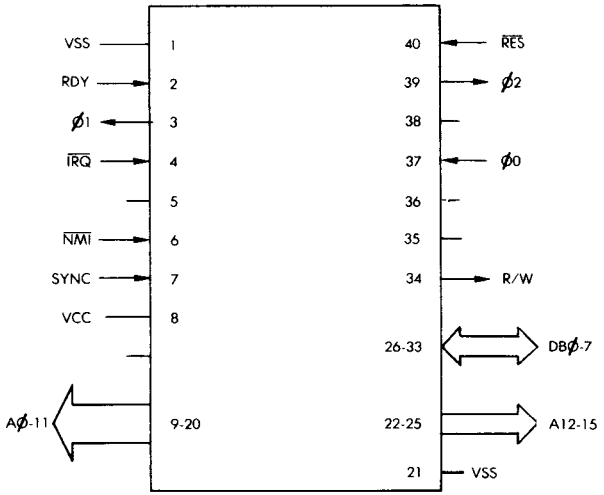


Fig. 2-7: 6502 pen aansluitingen

De controlesignalen

- R/W: het READ/WRITE (lees/schrijf) signaal bepaalt de richting van de gegevensoverdracht over de databus.

- IRQ en NMI zijn "Interrupt Request" (verzoek tot onderbreking) en "Non-Maskable Interrupt" (niet af te schermen onderbreking). Deze twee interruptlijnen worden gebruikt in hoofdstuk 7.

- SYNC is een signaal dat aangeeft dat er een opcode gehaald wordt.

- RDY wordt doorgaans gebruikt om met traag geheugen te synchroniseren: deze lijn stopt de processor.

- SO zet de overflowvlag. Wordt normaal niet gebruikt.

- $\emptyset_0$, $\emptyset_1$ en $\emptyset_2$ zijn kloksignalen.

- RES is RESET (herzet) om te initialiseren.

- V_{ss} en V_{cc} zijn voor de voeding (5V).

SAMENVATTING VAN DE HARDWARE

Hiermee is de beschrijving van de hardware van de interne organisatie van de 6502 vollledig. Op dit punt is de exacte interne busstruktuur niet belangrijk. De funktie van ieder van de registers moet echter duidelijk zijn voor de lezer verder gaat, omdat dit bijzonder belangrijk is. Als je vertrouwd bent met de gepresenteerde begrippen, lees dan maar verder, anders raden we je aan om de relevante stukken uit dit hoofdstuk nogmaals door te lezen, omdat ze nodig zijn in de volgende hoofdstukken. Kijk nog eens naar Fig. 2-2 en verzeker je ervan dat je de funktie van ieder van de registers in de illustratie begrijpt.

3
BASIS PROGRAMMEER-TECHNIEKEN

INLEIDING

Het is doel van dit hoofdstuk de lezer vertrouwd te maken met alle basistechnieken, die nodig zijn om, gebruik makend van de 6502, programma's te schrijven. In dit hoofdstuk worden begrippen beschreven als het werken met registers, loops en subroutines. De nadruk zal liggen op programmeertechnieken, waarbij alleen interne bronnen van de 6502 gebruikt worden, d.w.z. de registers. We zullen bijvoorbeeld rekenkundige programma's ontwikkelen. Het doel van deze programma's is het illustreren van tot nu toe behandelde begrippen, gebruik makend van de eigenlijke instrukties. Op deze manier zal duidelijk worden, hoe instrukties gebruikt kunnen worden om informatie te manipuleren tussen het geheugen en de RLE, en binnen de RLE zelf. In het volgende hoofdstuk zullen dan de instrukties van de 6502 besproken worden. In hoofdstuk 6 worden dan technieken gepresenteerd die nodig zijn im informatie *buiten* de 6502 te manipuleren: de input/output technieken.

In dit hoofdstuk zullen we de theorie leren vanuit de praktijk. Door steeds kompleksere programma's te bestuderen zullen we de funktie van de diverse instrukties en de registers leren en de tot nu toe behandelde begrippen toepassen. Een belangrijk begrip zal hier niet behandeld worden, en wel dat van de adresseertechnieken. Omdat dit op het eerste gezicht nogal ingewikkeld lijkt, zal dit apart besproken worden in hoofdstuk 5.

Laten we nu beginnen met een paar rekenkundige programma's

REKENKUNDIGE PROGRAMMA'S

Rekenkundige programma's omvatten optellen, aftrekken vermenigvuldigen en delen. De programma's die hier gepresenteerd worden zullen met gehele getallen werken. Deze getallen kunnen positieve binaire getallen zijn, of anders in de 2-komplement notatie uitgedrukt zijn, zodat dan het meest linkse bit het tekenbit is. (Zie hoofdstuk 1 voor een opfrissing van de 2-komplement notatie.)

8-BITS OPTELLING

We zullen nu twee 8-bits operanden, genaamd OP1 en OP2, die op resp. adres ADR1 en ADR2 opgeslagen zijn, optellen. De som, genaamd RES, wordt opgeslagen op geheugenadres ADR3. Dit is geillustreerd in fig. 3-1. Het programma is als volgt:

```
LDA     ADR1    LAADT OP1 IN A
ADC     ADR2    TEL OP2 BIJ OP1 OP
STA     ADR3    BERG RES OP IN ADR3
```

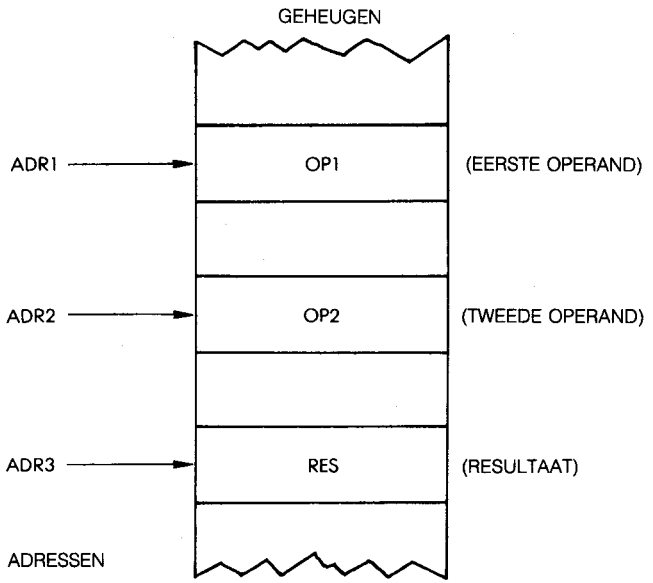


Fig. 3-1: 8-bits optelling, RES = OP1 + OP2

Dit is een programma van drie instrukties. Iedere regel is en instruktie in symbolische notatie. Ieder van deze instrukties wordt door en *assembleerprogramma* vertaald in 1, 2 of 3 bytes. We zullen ons hier niet met de vertaling bezighouden en alleen maar naar de symbolische notatie kijken. Op de eerste regel staat een LDA instruktie. LDA betekent: "laad de accumulator van het adres dat volgt".

Het adres dat op de eerste regel gespecificeerd is, is ADR1. ADR1 is een symbolische representatie van een 16-bits adres. Ergens anders in het programma wordt het ADR1 symbool gedefinieerd. Het zou bijvoorbeeld adres 100 kunnen zijn.

De instruktie LDA specificeert dan: "laad de accumulator met de inhoud van het geheugen met adres 100". Dit heeft dan een leesoperatie tot gevolg van adres 100, waarvan de inhoud dan via de databus in de accumulator geladen wordt. Je zult je herinneren, dat de logische en rekenkundige bewerkingen de accumulator als en van de operanden gebruiken. (Zie vorige hoofdstuk.) Aangezien we de twee waarden OP1 en OP2 bij elkaar op willen tellen, laden we eerst OP1 in de accumulator. Dan kunnen we de inhoud van de accumulator (OP1) bij OP2 optellen.

Het meest rechtse veld van de instruktie wordt het *kommentaarveld* genoemd. Het wordt genegeerd door de processor, maar is er om de leesbaarheid van het programma te verhogen. Om te begrijpen, wat een programma doet, is het erg belangrijk om dit van goed kommentaar te voorzien. Dit wordt het *dokumenteren* van programma's genoemd. In dit geval spreekt het kommentaar voor zich. De waarde van OP1 op adres ADR1 wordt in de accumulator geladen.

Het resultaat is geillustreerd in figuur 3-2.

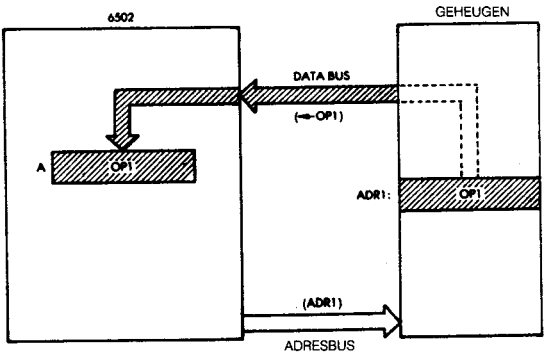


Fig. 3-2: LDA ADR1: OP1 wordt vanuit het geheugen geladen.

De tweede instruktie van ons programma is:

```
ADC     ADR2
```

Deze specificeert:"tel de inhoud van geheugenplaats ADR2 bij de accumulator op". Zie fig. 3-1: de inhoud van geheugenplaats ADR2 is OP2, onze tweede operand. De huidige inhoud van de accumulator is OP1, onze eerste operand. Het gevolg van het uitvoeren van de tweede instruktie is, dat OP2 uit het geheugen gehaald wordt en bij OP1 opgeteld wordt. De som wordt in de accumulator geplaatst. De lezer zal zich herinneren, dat het resultaat van een rekenkundige bewerking bij de 6502 weer terug in de accumulator geplaatst wordt. Bij andere microprocessoren is het mogelijk om deze resultaten in andere registers of weer terug in het geheugen te plaatsen. De som van OP1 en OP2 zit nu in de accumulator. We hoeven nu alleen nog maar de inhoud van de accumulator in geheugenplaats ADR3 op te slaan om het resultaat op de gespecificeerde plaats op te bergen. Ook hier is het meest rechtse veld een kommentaarveld, dat de funktie van de instruktie verklaart. (Tel OP2 bij A op.)

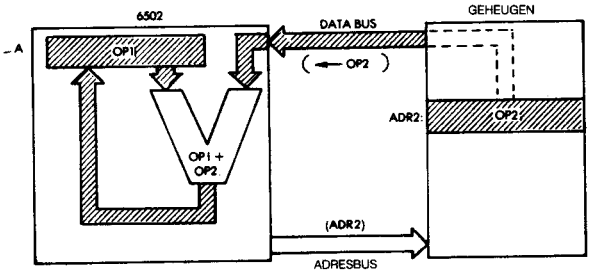


Fig. 3-3: ADC ADR2

Het effekt van de tweede instruktie is te zien in fig. 3-3. Hierin is te zien, dat in eerste instantie de accumulator OP1 bevatte. Na de optelling is een nieuw resultaat in de accumulator geschreven, te weten OP1 + OP2. De inhoud van alle registers in het systeem, zowel als die van alle geheugenplaatsen, blijft ongewijzigd als er een leesoperatie uitgevoerd wordt. Met andere woorden, *het lezen van de inhoud van een register of een geheugenplaats verandert deze inhoud niet.* Alleen, en uitsluitend, een schrijfoperatie verandert de inhoud van een register of

een geheugenplaats. In dit voorbeeld blijft de inhoud van de geheugenplaatsen ADR1 en ADR2 onveranderd. Nadat echter de tweede instruktie van dit programma uitgevoerd is, is de inhoud van de accumulator veranderd, omdat de uitvoer van de RLE in de accumulator geschreven wordt. De vorige inhoud van de accumulator is nu verloren.

Laten we nu dit resultaat op adres ADR3 opbergen en daarmee onze eenvoudige optelling afronden.

De derde instruktie luidt: STA ADR3. Dit betekent: "berg de inhoud van de accumulator op in adres ADR3". Dit spreekt voor zichzelf, en is geillustreerd in fig. 3-4.

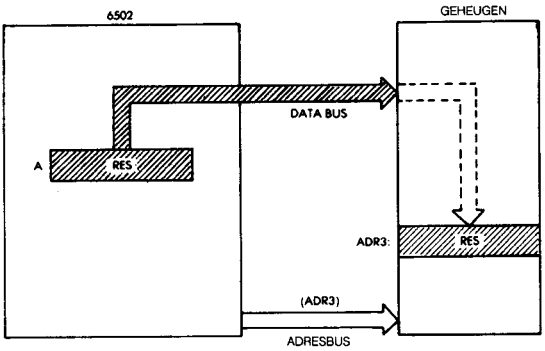


Fig. 3-4: STA ADR3 (Bewaar accumulator in geheugen)

Eigenaardigheden van de 6502

Het bovenstaande programma van drie instrukties zou voor de meeste microprocessoren inderdaad een kompleet programma zijn. De 6502 heeft echter twee eigenaardigheden, waardoor er meestal twee extra instrukties nodig zijn.

Ten eerste betekent de ADC instruktie eigenlijk: "*optellen met carry*". Het verschil is, dat een normale optelinstruktie twee getallen optelt. Een tel-op-met-carry instruktie telt twee getallen op, samen met de waarde van de carry. Aangezien we hier twee getallen optellen, hebben we het carrybit niet nodig. Omdat we op het moment dat we de optelling uitvoeren, niet altijd de toestand van de carrybit kennen (het zou gezet kunnen zijn door een vorige instruktie), moeten we het "clearen", d.w.z. op nul zetten. Dit kunnen we doen met een CLC instruktie: "clear carry".

Helaas heeft de 6502 niet beide optelinstrukties. Hij kent alleen de ADC instruktie. Het gevolg is, dat als we alleen twee 8-bits getallen willen optellen, het clearen van het carrybit altijd noodzakelijk is. Dit is geen belangrijk nadeel, maar mogen we niet vergeten.

De tweede eigenaardigheid van de 6502 ligt in het feit, dat hij uitgerust is met krachtige decimale instrukties, die we in het volgende deel over BCD berekeningen zullen gebruiken. De 6502 werkt altijd in een van twee modes: binair of decimaal. De mode waarin hij werkt wordt bepaald door een statusbit, het D-bit (van register P). Aangezien we in dit voorbeeld binair werken, moeten we zeker zijn in welke toestand het D-bit staat. Dit kunnen we doen met de CLD instruktie, die het D-bit cleared. Natuurlijk is het zo, dat als alle berekeningen binair zijn, het D-bit eenmaal aan het begin van het programma gecleared wordt, zodat het niet steeds opnieuw gezet hoeft te worden. Daarom kan deze instruktie in de meeste programma's weggelaten worden. Deze instruktie is hier toegevoegd, omdat de lezer, die deze oefeningen op een computer uitvoert, waarschijnlijk zowel binaire als BCD oefeningen zal maken, en omdat hij tenminste eenmaal in het programma voor een binaire optelling moet staan.

Samenvattend is nu ons volledig, en veilig, 8-bits programma nu:

```
CLC                     CLEAR CARRYBIT
CLD                     CLEAR DECIMAALBIT
LDA     ADR1
ADC     ADR2
STA     ADR3
```

Er kunnen in plaats van ADR1, ADR2 en ADR3, werkelijke fysische adressen gebruikt worden. Als men symbolische adressen wil houden, moeten er zogenaamde "pseudo-instrukties" gebruikt worden, waarmee een waarde aan deze symbolische adressen toegewezen wordt, zodat het vertaalprogramma tijdens de vertaling de werkelijke adressen hiervoor kan vervangen.
Dergelijke pseudo-instrukties zijn bijvoorbeeld:

```
ADR1 = $100
ADR2 = $120
ADR3 = $200
```

Oefening 3.1: Sluit nu dit boek. Kijk alleen naar de lijst met instrukties

achterin dit boek. Schrijf nu een programma, dat twee getallen optelt, die opgeslagen zijn in geheugenplaatsen LOC1 en LOC2. Berg het resultaat op in geheugenplaats LOC3. Vergelijk jouw programma dan met het bovenstaande.

16-Bits Optelling

Bij een 8-bits optelling kunnen alleen twee 8-bits getallen opgeteld worden, d.w.z. getallen tussen 0 en 255, als de absoluut-binaire notatie gebruikt wordt. In de meeste praktische toepassingen is *meervoudige precisie* vereist en moeten getallen van 16 bits en meer opgeteld worden. We zullen hier een paar voorbeelden geven van berekeningen met 16-bits getallen. Ze kunnen eenvoudig uitgebreid worden tot 24-, 32-bits of meer. (Er worden altijd veelvouden van 8 gebruikt.) We zullen aannemen dat de eerste operand opgeslagen is op adressen ADR1 en ADR1-1. Omdat OP1 nu een 16-bits getal is, zijn er twee geheugenplaatsen nodig. Op dezelfde manier wordt OP2 opgeslagen op adressen ADR2 en ADR2-1. Het resultaat moet opgeborgen worden op adressen ADR3 en ADR3-1. Dit is geillustreerd in fig. 3-5.

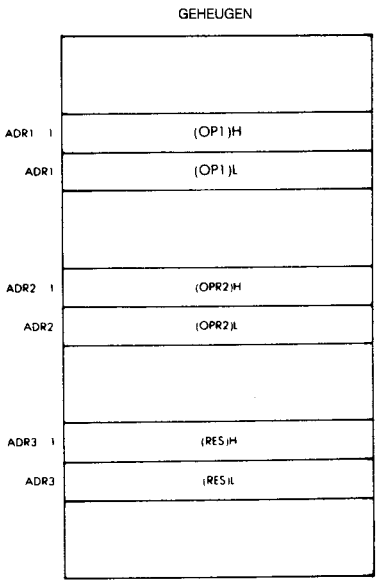


Fig. 3-5: 16-bits optelling: de operanden

De logika van dit programma is identiek aan die van het vorige. Eerst wordt de laagste helft van de twee operanden opgeteld, aangezien een microprocessor maar 8-bits tegelijk kan optellen. Een eventuele overdracht, die bij deze optelling van de lage orde bytes wordt gegenereerd, wordt automatisch opgeslagen in het interne carrybit ("C"). Vervolgens worden de hoge orde bytes van de twee operanden opgeteld, samen met de carry, en het resultaat wordt opgeborgen in het geheugen. Het programma is als volgt:

```
CLC
CLD
LDA     ADR1      LAGE HELFT VAN OP1
ADC     ADR2      (OP1 + OP2) LAAG
STA     ADR3      RED LAGE HELFT RES
LDA     ADR1-1    HOGE HELFT VAN OP1
ADC     ADR2-1    (OP1 + OP2) HOOG + CARRY
STA     ADR3-1    RED HOGE HELFT RES
```

De eerste twee instrukties van dit programma worden uit voorzorg gebruikt: CLC en CLD. Hun funktie is in de vorige paragraaf verklaard. Laten we het programma eens bekijken: de volgende drie instrukties zijn in wezen indentiek aan die van de 8-bits optelling uit het vorige programma. Het resultaat is de optelling van de minst signifikante helften (bits 0-7) van OP1 en OP2. De som, genaamd RES, wordt opgeslagen in geheugenplaats ADR3.

Altijd, als er een optelling verricht wordt, wordt een eventuele carry opgeslagen in het carrybit van het statusregister (register P). Als de 8 bits getallen een carry genereren, zal het carrybit "1" zijn.

De volgende drie instrukties zijn in wezen ook identiek aan het vorige optelprogramma. Ze tellen de meest signifikante helft (bits 8-15) op, vermeerderd met een eventuele carry, en slaan het resultaat op in adres ADR3-1.

Nadat dit programma uitgevoerd is, is het 16-bits resultaat opgeborgen op geheugenplaatsen ADR3 en ADR3-1. Hier wordt aangenomen, dat het resultaat een 16-bits getal is. Als de programmeur, om welke reden dan ook, vermoedt dat voor het resultaat 17 bits nodig zijn, moeten er extra instrukties toegevoegd worden om het carrybit te testen na deze optelling.

De plaats van de operanden in het geheugen is geillustreerd in fig. 3-5.

N.B.: We hebben hier aangenomen, dat het meest signifikante deel

van de operand "boven" het minst signifikante deel opgeslagen wordt, d.w.z. op een lager geheugenadres. Dit hoeft niet altijd het geval te zijn. Het is zelfs zo, dat de 6502 adressen andersom opslaat. Eerst wordt het lagere deel in het geheugen opgeborgen, en vervolgens het hogere deel op de volgende geheugenplaats. Om zowel voor adressen als gegevens dezelfde gewoonte te hebben, wordt het aangeraden dat gegevens ook opgeslagen worden met het lagere deel bovenop het hogere deel. Dit is geillustreerd in fig. 3-6.

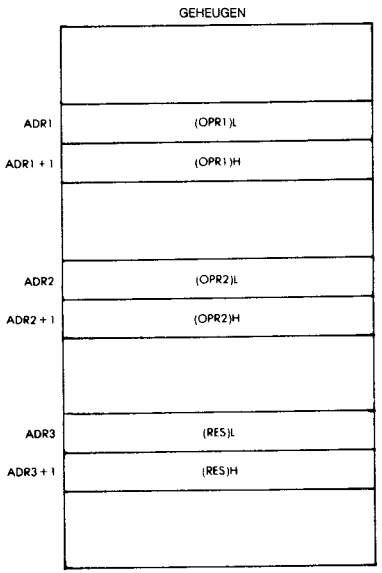


Fig. 3-6: Operanden in omgekeerde volgorde opslaan

Oefening 3.2: Herschrijf het bovenstaande 16-bits optelprogramma met de geheugenindeling van fig. 3-6.

Oefening 3.3: Neem nu aan, dat ADR1 niet naar de laagste helft van OPR1 (zie figuur 3-6) wijst, maar naar de hoogste helft van OPR1. Dit is geillustreerd in fig. 3-6. Schrijf ook nu het overeenkomstige programma.

Het is aan de programmeur, jij dus, om te bepalen hoe 16-bits getallen opgeslagen worden (d.w.z. het hoge deel eerst of het lage deel

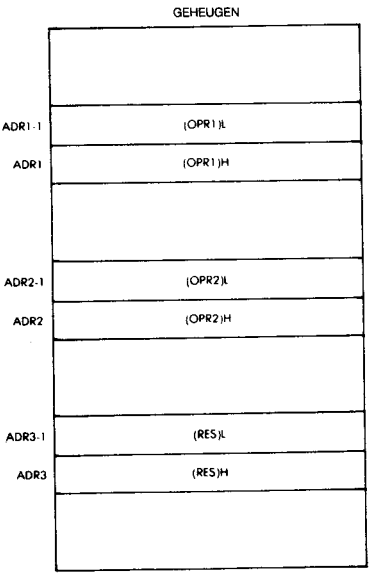


Fig. 3-6[1]: Naar het hoge byte wijzen

eerst) en ook of de adresreferenties naar het hoge of het lage deel wijzen. Dit is de eerste van vele keuzen die je moet leren te maken als je algoritmes of gegevensstrukturen ontwerpt. We hebben nu geleerd om binair op te tellen. Laten we nu eens naar het aftrekken kijken.

Aftrekken van 16-bits getallen

Aftrekken van 8-bits getallen zou te eenvoudig zijn. Laten we het nu als oefening beschouwen en meteen 16-bits getallen aftrekken. Zoals gebruikelijk zijn onze twee getallen, OPR1 en OPR2, opgeslagen op adressen ADR1 en ADR2. We nemen aan, dat de geheugenindeling die van figuur 3-6 is. Om af te trekken gebruiken we een aftrekbewerking (SBC) in plaats van een optel bewerking (ADC). Het enige andere verschil vergeleken met de optelling is, dat we een SEC instruktie gebruiken aan het begin van het programma in plaats van een CLC. SEC betekent "zet carry op 1". Dit geeft een "no borrow" (niet lenen) toestand aan. De rest van het programma is identiek aan dat voor een optelling. Het progamma is als volgt:

```
CLD
SEC                     ZET CARRY OP 1
LDA     ADR1            (OPR1)L IN A
SBC     ADR2            (OPR1)L-(OPR2)L
STA     ADR3            BERG (RESULTAAT)L OP
LDA     ADR1 + 1        (OPR1)H IN A
SBC     ADR2 + 1        (OPR1)H-(OPR2)H
STA     ADR3 + 1        BERG (RESULTAAT)H OP
```

Oefening 3-4: Schrijf een aftrekprogramma voor 8-bits operanden.

Je moet er aan denken, dat bij 2-komplement berekeningen, de uiteindelijke waarde van de carryvlag geen betekenis heeft. Als er tengevolge van de aftrekking een overflow-toestand optreedt, wordt het overflowbit (bit V) van het statusregister gezet. Hierop kan getest worden. De zojuist gegeven voorbeelden zijn eenvoudige binaire optellingen. Er kan echter een ander type optelling nodig zijn, de BCD-optelling.

BCD berekeningen

8-bits BCD optelling

Het begrip van de BCD berekeningen is besproken in hoofdstuk 1. Het wordt voornamelijk in zakelijke toepassingen gebruikt, waar het van het hoogste belang is om ieder signifikant cijfer van een resultaat te bewaren. In de BCD-notatie wordt een 4-bits nibble gebruikt om een decimaal cijfer op te slaan (0 tot 9). Er kunnen dus twee BCD cijfers in een byte opgeslagen worden. (Dit wordt *packed BCD* genoemd.) Laten we nu eens twee bytes met ieder twee BCD cijfers optellen.

Om wat inzicht in de problemen te krijgen, proberen we eerst een paar numerieke voorbeelden.

Laten we eens "01" bij "02" optellen.

"01" wordt voorgesteld door	0000.0001
"02" wordt voorgesteld door	0000.0010
Het resultaat is	0000.0011

Dit is de BCD voorstelling voor "3". (Als je onzeker bent t.a.v. het BCD equivalent, zie dan de conversietabel achterin het boek). Tot nu

toe werkte alles erg eenvoudig. Laten we nu een ander voorbeeld proberen.

"08" wordt voorgesteld door	0000.1000
"03" wordt voorgesteld door	0000.0011

Oefening 3.5: Bereken de som van de twee bovenstaande getallen in BCD notatie. Wat is je resultaat? (antwoord volgt)

Als je 0000.1011 krijgt, heb je de *binaire* som van "8" en "3" berekend. Je hebt inderdaad "11" verkregen in de *binaire* notatie. Helaas is "1011" een *ongeldige* BCD kode. Je had de BCD voorstelling van "11" moeten krijgen, d.w.z. "0001.0001"!

Het probleem komt voort uit het feit, dat de BCD voorstelling alleen de eerste tien kombinaties van de vier cijfers gebruikt om de decimale symbolen "0" tot "9" te koderen. De andere zes mogelijke kombinaties worden niet gebruikt, en de ongeldige "1011" is een van die kombinaties. Met andere woorden, steeds als de som van twee binaire cijfers groter is dan "9", moet bij het resultaat "6" opgeteld worden om over de zes ongeldige kodes heen te springen. Tel de binaire voorstelling van "6" bij "1011" op:

```
          1011   (ongeldig binair resultaat)
        + 0110   (+ 6)
        ---------
Res:    00010001.
```

Dit is inderdaad "11" in BCD notatie. We hebben nu het juiste resultaat.

Dit voorbeeld illustreert één van de moeilijkheden van de BCD notatie. De zes ontbrekende kodes moeten gekompenseerd worden. Bij de meeste microprocessoren moet een speciale instruktie, genaamd "decimal adjust" (decimale aanpassing) gebruikt worden om het resultaat van een optelling aan te passen. Bij de 6502 doet de ADC instruktie dit automatisch. Dit is een duidelijk voordeel van de 6502 als er BCD berekeningen uitgevoerd moeten worden.

Het volgende probleem wordt door hetzelfde voorbeeld geillustreerd. In ons voorbeeld wordt een overdracht (carry) gegenereerd vanuit het lagere BCD cijfer (het meest rechtse) naar het linker. Met deze interne carry moet rekening gehouden worden, en hij moet opgeteld worden bij het tweede BCD cijfer. De optelinstruktie van de 6502

zorgt hier automatisch voor. Het is echter vaak handig om deze interne carry van bit 3 naar bit 4 (de *half-carry*) te detekteren. Hiervoor is in de 6502 geen statusbit aanwezig.

Tenslotte moeten net als bij een binaire optelling de gebruikelijke SED en CLC instrukties gebruikt worden, om de BCD optelling korrekt uit te voeren. Als voorbeeld is een programma om de BCD getallen "11" en "22" op te tellen hieronder gegeven:

```
CLC                 CLEAR CARRY
SED                 ZET DECIMALE MODE
LDA     #$11        BCD LITERAL "11"
ADC     #$22        BCD LITERAL "22"
STA     ADR
```

In dit programma gebruiken we twee nieuwe symbolen: "#" en "$". Het "#" symbool geeft aan, dat er een "literal" (of konstante) volgt. Het "$" teken in het operandveld van de instruktie geeft aan, dat de gegevens die volgen hexadecimaal genoteerd staan. De hexadecimale en de BCD representatie van de cijfers "0" tot "9" zijn hetzelfde. Hier willen we de literals "11" en "22" optellen. Het resultaat wordt opgeslagen op adres ADR. Als de operand gespecificeerd wordt als deel van de instruktie, zoals hierboven, wordt dit *immediate* adressering genoemd (immediate = meteen). (De verschillende manieren om te

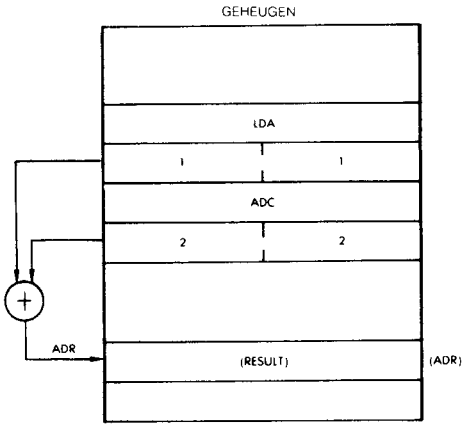


Fig. 3-7: Opslaan van BCD cijfers

adresseren worden in detail besproken in hoofdstuk 5.) Het opslaan van het resultaat op een gespecificeerd adres, zoals STA ADR, wordt *absolute* adressering genoemd, als ADR een 16-bits adres voorstelt.

Oefening 3.6: Zouden we de CLC instruktie in het programma onder de LDA instruktie kunnen plaatsen?

BCD aftrekking

Aftrekken in BCD lijkt op het eerste gezicht ingewikkeld. Om in BCD af te trekken, moet het 10-komplement van het getal opgeteld worden, net als het 2-komplement van een getal opgeteld wordt om binair af te trekken. Het 10-komplement wordt verkregen, door het komplement van 9 uit te rekenen en daar 1 bij op te tellen. Bij de meeste microprocessoren zijn hier drie tot vier berekeningen voor nodig. De 6502 is echter met een speciale instruktie uitgerust die dit in een keer doet! Natuurlijk moet het programma, net als in het binaire voorbeeld, voorafgegaan worden door de instrukties SED (waarmee de decimale mode gezet wordt, tenzij die al eerder gezet is) en SEC, waarmee de carry op 1 gezet wordt. Het programma om BCD "25" van BCD "26" af te trekken wordt dan als volgt:

```
SED             ZET DECIMALE MODE
SEC             ZET CARRY
LDA   #$26      LAAD BCD 26
SBC   #$25      TREK BCD 25 AF
STA   ADR       BERG HET RESULTAAT OP
```

16-bit BCD optelling

Een 16-bits optelling is net zo eenvoudig als in het binaire geval. Het programma voor een dergelijke optelling is:

```
CLC
SED
LDA   ADR1
ADC   ADR2
STA   ADR3
LDA   ADR1-1
ADC   ADR2-1
STA   ADR3-1
```

Oefening 3.7: Vergelijk bovenstaand programma met dat voor de binaire 16-bits optelling. Wat is het verschil?

Oefening 3.8: Schrijf een programma om twee 16-bits BCD getallen af te trekken. (CLC en ADC niet gebruiken!)

BCD Vlaggen

In de BCD mode geeft het carrybit tijdens een optelling aan, dat het resultaat groter dan 99 is. Dit verschilt dus van 2-komplement, aangezien BCD cijfers binair weergegeven worden. Het nul zijn van de carry geeft een borrow aan.

Programmeertips voor optellen en aftrekken

- Voor een optelling altijd de carry clearen.
- Voor een aftrekking altijd de carry op 1 zetten
- Zet de juiste mode: binair of decimaal.

Instruktietypen

We hebben nu drie typen microprocessor-instrukties gebruikt. We hebben LDA en STA gebruikt, die respectievelijk de inhoud van de accumulator laden van een geheugenadres en de inhoud op een bepaald geheugenadres opbergen. Deze twee instrukties worden *gegevensoverdracht*-instrukties genoemd.

Vervolgens hebben we *rekenkundige* instrukties gebruikt, zoals ADC en SBC. Ze verrichten respectievelijk een optelling en een aftrekking. Meer RLE instrukties zullen we verderop in dit hoofdstuk introduceren.

Tenslotte hebben we instrukties zoals CLC en SED gebruikt en andere, waarmee de statusbits gemanipuleerd kunnen worden (resp. het carrybit en het decimaalbit in onze voorbeelden). Dit zijn de *statusmanipulatie-* of kontrole-instrukties. Een uitgebreide beschrijving van de 6502 instrukties wordt gegeven in hoofdstuk 4.

De microprocessor heeft nog andere typen instrukties die we nog niet gebruikt hebben. In het bijzonder zijn dit de "sprong"-instrukties, waarmee de volgorde waarin het programma uitgevoerd wordt veranderd kan worden. Dit nieuwe type instruktie zal in ons volgende voorbeeld geintroduceerd worden.

Vermenigvuldiging

Laten we nu eens een wat ingewikkelder rekenkundig probleem onderzoeken: het vermenigvuldigen van binaire getallen. Laten we voordat we het algoritme voor een binaire vermenigvuldiging geven, eerst een gewone decimale vermenigvuldiging onderzoeken: we zullen 12 met 23 vermenigvuldigen.

```
  12 (vermenigvuldigtal)
 x23 (vermenigvuldiger)
-----
  36 (deelprodukt)

 + 24
-----
 =276 (eindresultaat)
```

De vermenigvuldiging wordt uigevoerd door het meest rechtse cijfer van de vermenigvuldiger met het vermenigvuldigtal te vermenigvuldigen, dus "3" × "12". Het deelprodukt is "36". Vervolgens wordt het volgende cijfer van de vermenigvuldiger vermenigvuldigd, dus "2" met "12". "24" wordt dan bij het deelprodukt opgeteld.

Maar er is nog een bewerking: 24 staat een positie *naar links*. We zullen zeggen, dat 24 een positie *naar links opgeschoven* is. We hadden ook kunnen zeggen, dat het deelprodukt een positie *naar rechts opgeschoven* was voor de optelling. De twee getallen, op de juiste manier opgeschoven worden dan opgeteld en de som is 276. Dat is eenvoudig. Laten we nu eens naar de binaire vermeniguldiging kijken. De binaire vermenigvuldiging wordt op precies dezelfde manier uitgevoerd.

We nemen een voorbeeld. We vermenigvuldigen 5 × 3:

```
(5)      101   (VMR)
(3)    x 011   (VMT)
-------------
         101   (DP)
       101
       000
-------------
(15)   01111 (RES)
```

Om de vermenigvuldiging uit te voeren gaan we op precies dezelfde wijze als bovenstaand te werk. De formele representatie van dit algoritme is gegeven in figuur 3-8. Het is een stroomdiagram voor dit algo-

ritme, ons eerste stroomdiagram. Laten we dit eens wat nader bekijken.

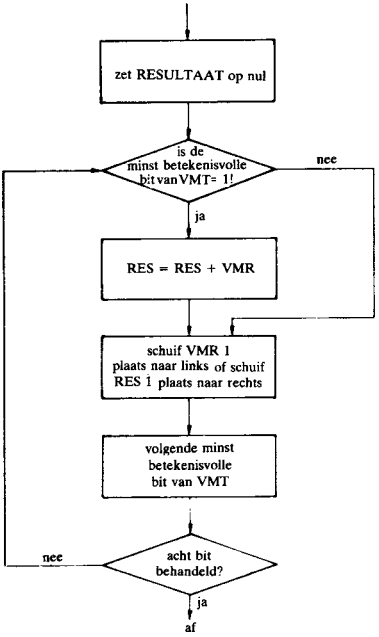


Fig. 3-8: Het basisalgoritme voor een vermenigvuldiging: stroomdiagram.

Dit stroomdiagram is een symbolische weergave van het algoritme dat we zojuist hebben gegeven. Iedere rechthoek stelt een uit te voeren opdracht voor. Deze wordt vertaald in een of meerdere programma-instrukties. Iedere ruit stelt een test voor. Dit is een plaats in het programma waar gesprongen wordt. Als de test waar is, wordt naar de gespecificeerde plaats gesprongen. Als de test niet waar is, springen we naar een andere plaats. Het begrip "springen" wordt later in het programma zelf besproken. Bekijk dit stroomdiagram eens en verzeker je ervan, dat het inderdaad een juiste weergave van het zojuist gegeven algoritme is. Merk op dat er een pijl uit de laatste ruit onderin het diagram terugwijst naar de eerste ruit bovenin. Dat is, omdat dit deel van het diagram acht keer uitgevoerd moet worden, eenmaal voor ieder bit van de vermenigvuldiger. Een dergelijke situatie, waar het uitvoeren steeds op dezelfde plaats begint, wordt om voor de hand liggende redenen een *programma-loop* (programma-lus) genoemd.

Oefening 3.9: Vermenigvuldig binair "4" met "7", gebruik makend van het stroomdiagram, en ga na dat je "28" krijgt. Als dit niet lukt, probeer het dan nogmaals. Pas als dit lukt, kun je dit stroomdiagram in een programma omzetten.

We zullen nu het stroomdiagram omzetten in een programma voor de 6502. Het volledige programma is gegeven in fig. 3-9. We zullen dit nu in detail bestuderen. Zoals je je zult herinneren van hoofdstuk 1, bestaat het programmeren uit het vertalen van het stroomdiagram van fig. 3.8 in het programma van fig. 3.9. Ieder blok van het stroomdiagram wordt vertaald naar een of meerdere instrukties.

Aangenomen wordt, dat VMR en VMT al een waarde hebben.

```
        LDA   #0        ZET ACCUMULATOR OP NUL
        STA   TIJD      ZET DIT ADRES OP NUL
        STA   RESAD     ZET OP NUL
        STA   RESAD+1   ZET OP NUL
        LDX   #8        X = TELLER
VERM    LSR   VMRAD     SCHUIF VMR RECHTS
        BCC   NIETOP    TEST CARRYBIT
        LDA   RESAD     LAAD A MET RES LAAG
        CLC             BEREID OPTELLING VOOR
        ADC   VMTAD     TEL VMT BIJ RES OP
        STA   RESAD     BERG RES OP
        LDA   RESAD+1   TEL REST VAN VERSCHOVEN
                        VMT OP
        ADC   TIJD
        STA   RESAD+1
NIETOP  ASL   VMTAD     SCHUIF VMT LINKS
        ROL   TIJD      BEWAAR BIT UIT VMT
        DEX             VERLAAG TELLER
        BNE   VERM      HERHAAL ALS TELLER #0
```

Figuur 3-9: 8 × 8 vermenigvuldiging

Het eerste blok van het stroomdiagram is een *initialisatie-blok*. Het is nodig om een aantal registers of geheugenplaatsen op nul te zetten, omdat het programma ze nodig heeft. De registers die door het programma gebruikt worden zijn te zien in fig. 3.10. Links staat het deel van de 6502 dat hier gebruikt wordt. Rechts staat het gebruikte deel van het geheugen. We zullen hier aannemen, dat geheugenadressen van boven naar beneden toenemen. De omgekeerde volgorde had natuurlijk ook gebruikt kunnen worden. Het X-register uiterst links (één

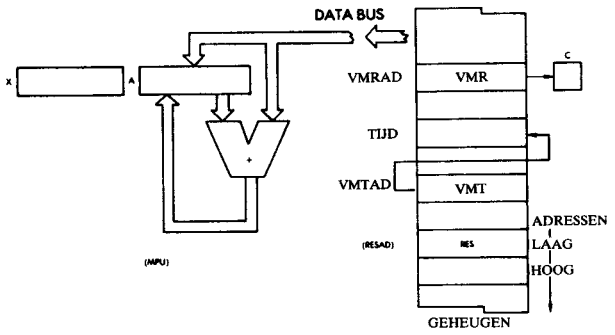


Fig. 3-10: Vermenigvuldiging: de registers

van de twee indexregisters) zal als *teller* gebruikt worden. Omdat we een 8-bits vermenigvuldiging doen, moeten we 8 bits van de vermenigvuldiger testen. Helaas heeft de 6502 geen instruktie om deze bits na elkaar te testen. De enige bits die eenvoudig getest kunnen worden, zijn de bits van het statusregister. Het gevolg van deze beperking van bijna alle microprocessoren is, dat om alle bits van de vermenigvuldiger succesvol te testen, de waarde van de vermenigvuldiger naar de accumulator overgebracht moet worden. Dan wordt de inhoud van de accumulator naar rechts geschoven. Het bit dat uit het registr geschoven wordt komt in het carrybit van het statusregiser. Het effekt van een schuifbewerking is getekend in onderstaande fig.3.11. Er zijn veel variaties mogelijk, afhankelijk van het bit dat in het register komt, maar deze verschillen worden in hoofdstuk 4 besproken (Instruktieset van de 6502).

Laten we teruggaan naar het achtereenvolgens testen van de 8 bits van de vermenigvuldiger. Aangezien het carrybit gemakkelijk getest kan worden, wordt de vermenigvuldiger 8 maal een positie verschoven. Iedere keer valt het meest rechtse bit in het carrybit, waar het getest wordt.

Het volgende probleem dat opgelost moet worden is het feit dat voor het deelprodukt, dat tijdens de optellingen verhoogd wordt, 16 bits nodig zijn. Het vermenigvuldigen van twee 8-bits getallen kan een 16-bits resultaat tot gevolg hebben, omdat $2^8 \times 2^8 = 2^{16}$. We moeten 16 bits voor dit resultaat reserveren. Helaas heeft de 6502 weinig interne registers, zodat dit deelprodukt niet in de 6502 zelf opgeborgen kan worden. Het is zelfs zo, dat we vanwege het beperkte aantal registers noch de vermenigvuldiger, noch het vermenigvuldigtal, noch het deelprodukt

in de 6502 kunnen opbergen. Ze worden allen in het geheugen opgeslagen. Dit heeft een langzamere uitvoering van het programma tot gevolg dan wanneer het mogelijk was geweest ze in interne registers op te slaan. Dit is een beperking die inherent is aan de 6502. Het geheugengebied, dat voor de vermenigvuldiging gebruikt wordt staat links in fig. 3.10. Bovenin is het geheugenwoord, dat aan de vermenigvuldiger toegewezen is, te herkennen. We zullen bijvoorbeeld aannemen dat het binair "3" bevat. Het adres van deze geheugenplaats is VMRAD. Daaronder vinden we een "tijdelijke" geheugenplaats, waarvan het adres TIJD is. De rol van deze plaats wordt straks uitgelegd. We schuiven het vermenigvuldigtal in deze plaats naar links, voordat het bij het deelprodukt opgeteld wordt. De volgende is het vermenigvuldigtal en wordt verondersteld de waarde binair "5" te hebben. Het adres is VMTAD.

Tenslotte zijn onderin twee plaatsen toegewezen aan het resultaat. Hun adres is RESAD.

Deze geheugenplaatsen zullen onze "werkregisters" zijn en het woord "register" mag in deze context verwisseld worden met "geheugenplaats".

De pijl die bovenin de tekening uit VMR naar bit C wijst, is een symbolische manier om aan te geven hoe de vermenigvuldiger in het carrybit geschoven wordt. Dit carrybit zit natuurlijk in de 6502 en niet in het geheugen.

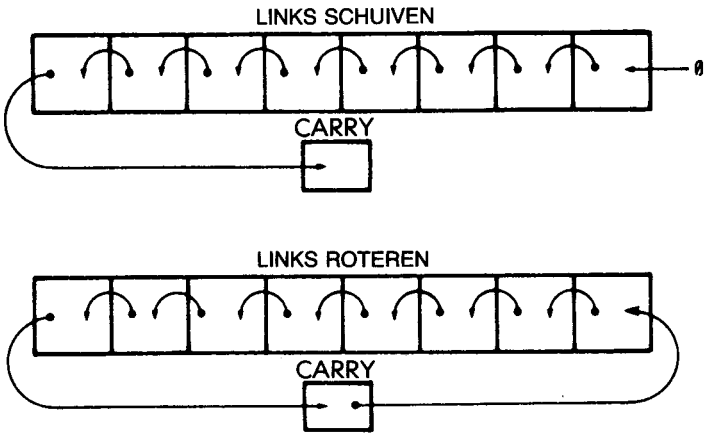


Fig. 3-11: Schuiven en Roteren

Laten we nu teruggaan naar het programma van figuur 3-9. De eerste vijf instrukties zijn initialisatie-instrukties:

De eerste vier instrukties zetten de inhoud van de "registers" TIJD, RESAD en RESAD+1 op nul. Laten we dit eens kontroleren.

```
LDA #0
```

Deze instruktie laadt de accumulator met de konstante "0". Het gevolg van deze instruktie is dat de accumulator "00000000" bevat.

De inhoud van de accumulator wordt nu gebruikt om de drie geheugen"registers" op nul te zetten. Je moet er aan denken, dat door het lezen van een register er niets aan de inhoud verandert. Er kan zo vaak als nodig is uit gelezen worden. De inhoud verandert niet door een leesbewerking. Vervolgens:

```
STA  TIJD
```

Deze instruktie bergt de inhoud van de accumulator op in geheugenplaats TIJD. Zie fig.3-11 om de stroom van de gegevens in het systeem te begrijpen. De accumulator bevat "00000000". Het resultaat van deze instruktie is dat er allemaal nullen geschreven worden in geheugenplaats TIJD. Denk eraan dat de inhoud van de accumulator nul blijft na de bewerking. De inhoud blijft onveranderd. We zullen hem weer gebruiken.

```
STA  RESAD
```

Deze instruktie werkt net als de voorafgaande en zet de inhoud van adres RESAD op nul. Laten we dit nogmaals doen:

```
STA  RESAD+1
```

Hiermee zetten we geheugenplaats RESAD+1 op nul, die gereserveerd is voor het hoge gedeelte van het resultaat. (Het hoge gedeelte wordt gevormd door bits 8-15, het lage deel door bits 0-7.)

Tenslotte is het nodig om het aantal verschuivingen dat we plegen te tellen, om op het juiste moment te kunnen stoppen met het verschuiven van van de vermenigvuldigerbits.

Er moet acht keer geschoven worden. Register X zal als teller gebruikt worden en krijgt de beginwaarde "8". Steeds als er eenmaal geschoven is, wordt de teller met 1 verminderd. Als de waarde van de

teller gelijk aan nul is, is de vermenigvuldiging beeindigd. We geven nu dit register de beginwaarde "8":

```
LDX #8
```

Deze instruktie laadt de konstante "8" in register X.

Als we het stroomdiagram van fig. 3-8 nog eens bekijken zien we, dat we het minst signifikante bit (Least Significant Bit = LSB) van de vermenigvuldiger nog moeten testen. Zoals we hierboven al hebben aangegeven, kan deze test niet met een enkele instruktie verricht worden. Hier zijn twee instrukties voor nodig. Eerst wordt de vermenigvuldiger naar rechts geschoven, en vervolgens wordt het uitgeschoven bit getest. Dit is het carrybit. Dit kan met de volgende instrukties:

```
LSR  VMRAD
```

Deze instruktie is een Logisch Schuiven naar Rechts van de inhoud van geheugenplaats VMRAD.

Oefening 3.10: Aannemende, dat de vermenigvuldiger in ons voorbeeld "3" is, welk bit wordt dan rechts uit geheugenplaats VMRAD geschoven? (Met andere woorden, welke waarde heeft carry na deze verschuiving?)

De volgende instruktie test de waarde van het carrybit:

```
BCC  NIETOP
```

Deze instruktie betekent: "Branch (spring) als Carry Clear" (dus als carry nul is) naar adres NIETOP.

Hier komen we voor het eerst een spronginstruktie tegen. Alle programma's die we tot nu toe behandeld hebben waren strikt sequentieel. Iedere instruktie werd na de vorige uitgevoerd. Na het uitvoeren van een logische test, zoals het testen van het carrybit, moet het mogelijk zijn om iedere willekeurige instruktie van het programma uit te voeren. Dit kan met een spronginstruktie. Deze instruktie test de waarde van het carrybit. Als de waarde "0" was, dat wil zeggen als carry op nul gezet werd, dan springt het proramma naar NIETOP. Dit betekent, dat als de test succesvol is, de volgende instruktie die uitgevoerd wordt na de BCC, die op adres NIETOP is.

Anders, als de test faalt, wordt er niet gesprongen en wordt gewoon de instruktie na BCC NIETOP uitgevoerd.

Er moet nog iets verteld worden over NIETOP: dit is een *symbolisch label*. Het representeert een fysiek adres in het geheugen. Voor het gemak van de programmeur staat het assembleerprogramma toe, dat er symbolische namen in plaats van de eigenlijke adressen gebruikt worden. Tijdens het assembleren substitueert de assembler voor het symbool "NIETOP" het eigenlijke adres. Dit geeft een aanzienlijke verbetering van de leesbaarheid van de programma's en maakt het ook mogelijk, dat de programmeur extra instrukties plaatst tussen het springpunt en NIETOP, zonder dat alles herschreven moet worden. Deze voordelen zullen in detail besproken worden in hoofdstuk 9 over de assembler.
We zullen nu beide alternatieven bestuderen:

Alternatief 1: De carry was "1".
Als de carry "1" was, faalt de test die door BCC gespecificeerd wordt, en wordt de volgende instruktie na de BCC uitgevoerd.

```
        LDA   RESAD
```

Alternatief 2: De carry was "0".
De test slaagt en de volgende instruktie is die met label "NIETOP".

Volgens het stroomdiagram van figuur 3-8 moet het vermenigvuldigtal bij het deelprodukt (hier de RES registers) opgeteld worden als carry 1 is. Er moet ook geschoven worden. Het deelprodukt moet 1 positie naar rechts geschoven worden, of anders moet het vermenigvuldigtal naar links geschoven worden.

Het vermenigvuldigtal is opgeslagen in registers TIJD en VMTAD. (Ter vereenvoudiging noemen we geheugenplaatsen "registers", een gebruikelijke term.) De 16 bits van het deelprodukt zijn opgeslagen in geheugenplaatsen RESAD en RESAD+1.

Om dit te illustreren, nemen we aan dat het vermenigvuldigtal "5" was. De verschillende registers zijn in fig. 3-10 getekend.
We hoeven slechts twee 16-bits getallen op te tellen. Dit probleem hebben we al opgelost. (Als je twijfelt, zie de paragraaf over 16-bits optelling hierboven.)

We tellen eerst de lage orde bytes op en dan de hoge orde bytes:

```
        LDA   RESAD
```

De accumulator wordt geladen met het lage gedeelte van RES:

```
CLC
```

Voor een optelling moet bij de 6502 het carrybit op nul worden gezet. Het is belangrijk om dit te doen, omdat we weten, dat het carrybit op 1 gezet was.

```
ADC  VMTAD
```

Het vermenigvuldigtal wordt bij de accumulator opgeteld, die (RES)LAAG bevat.

```
STA  RESAD
```

Het resultaat van de optelling wordt op de juiste geheugenplaatsen opgeslagen, (RES)LAAG. Dan wordt het tweede gedeelte van de optelling uitgevoerd. Als je later de uitvoering van dit proramma met de hand kontroleert, vergeet dan niet dat door de optelling het carrybit gezet wordt. Het carybit wordt op "0" of "1" gezet, afhankelijk van het resultaat van de optelling. Een eventuele gegenereerde carry zal automatisch verwerkt worden in het hoge gedeelte van het resultaat.

Laten we nu de optelling afronden:

```
LDA  RESAD+1
ADC  TIJD
STA  RESAD+1
```

Deze drie instrukties kompleteren onze 16-bits optelling.
We hebben nu het vermenigvuldigtal bij RES opgeteld. Dit moeten we nog een positie naar links opschuiven voor de volgende optelling. We hadden ook kunnen overwegen om het vermenigvuldigtal een positie naar links op te schuiven *voor de optelling*, behalve de eerste keer. Dit is een van de vele programmeer-opties die de programmeur open staan.

Laten we het vermenigvuldigtal naar links schuiven:

```
NIETOP  ASL  VMTAD
```

Deze instruktie is een "Rekenkundig (Arithmetic) Schuiven naar Links". Hij schuift de inhoud van geheugenplaats VMTAD een positie naar links. In deze geheugenplaats is nu het lage orde gedeelte van het vermenigvuldigtal opgeslagen. Dit is niet voldoende. We kunnen ons niet veroorloven het bit dat links uit het vermenigvuldigtal valt te verliezen. Dit bit valt in het carrybit. Het mag daar niet permanent opgeslagen blijven, omdat het door iedere rekenkundige bewerking vernietigd kan worden. Dit bit zou in een "permanent" register opgeslagen moeten worden. Het zou in geheugenplaats TIJD geschoven moeten worden. Dit is precies het effekt dat verkregen wordt met de instruktie:

```
ROL  TIJD
```

Deze specificeert: "Roteer Links" de inhoud van TIJD.

Hier kunnen we iets interessants opmerken. We hebben net twee verschillende schuifinstrukties gebruikt om een positie naar links te schuiven. De eerste is ASL. De tweede is ROL. Wat is het verschil?

De ASL instruktie schuift de inhoud van het register. De ROL instruktie is een roteerinstruktie. Hij schuift de inhoud van een register een positie naar links en het bit dat links uitgeschoven wordt, komt zoals gebruikelijk in het carrybit. Het verschil is, dat *de vorige inhoud van de carry nu rechts ingeschoven wordt.* Dit wordt in de wiskunde een circulaire rotatie genoemd (een 9-bits rotatie). Dit is nu net wat we willen. Het gevolg van de ROL is, dat het bit dat nu links uit TIJD geschoven was, en opgeslagen werd in het carrybit, nu op de meest rechtse positie van TIJD komt. Het werkt.

We zijn nu klaar met het rekenkundige deel van het programma. We moeten nog wel testen of we de bewerking wel acht maal uitgevoerd hebben, dus of we klaar zijn. Zoals bij de meeste microprocessoren zijn hier twee instrukties voor nodig:

```
DEX
```

Deze instruktie vermindert de inhoud van register X. Als de inhoud 8 was, is de inhoud na uitvoering van deze instruktie 7.

```
BNE  VERM
```

Dit is een andere test-en-spring instruktie. De betekenis is: "als het resultaat niet gelijk aan nul is, spring dan naar geheugenplaats VERM". Zolang ons tellerregister nog ongelijk aan nul is, wordt er

automatisch teruggesprongen naar label VERM. Dit noemen we de vermenigvuldigingslus. In het stroomdiagram voor de vermenigvuldiging komt dit overeen met de pijl die uit de laatste ruit komt. Deze loop wordt acht maal doorlopen.

Oefening 3.11: Wat gebeurt er als X na vermindering 0 wordt? Welke is de volgende instruktie die uitgevoerd wordt?

In de meeste gevallen zal het programma, dat we zojuist ontwikkeld hebben, een subroutine zijn, en de laatste instruktie in de subroutine zal dan een RTS zijn. Het subroutine-mechanisme wordt verderop in dit hoofdstuk uitgelegd.

BELANGRIJKE TEST VOOR JEZELF

Als je programmeren wilt leren, is het van het uiterste belang, dat je een dergelijk programma tot in de details begrijpt. We hebben veel nieuwe instrukties geintroduceerd. Het algoritme is redelijk eenvoudig, maar het programma is veel langer dan de tot nu toe ontwikkelde programma's. *Het wordt ten sterkste aanbevolen dat je de volgende oefening helemaal en foutloos maakt, voor je verder leest in dit hoofdstuk.* Pas als je dit foutloos kunt, heb je het mechanisme begrepen, waarmee de instrukties de inhoud van het geheugen en de registers van de microprocessor manipuleren, en hoe de carryvlag gebruikt wordt. Anders is het waarschijnlijk, dat je moeilijkheden zult ondervinden bij het schrijven van programma's. Om programmeren te leren, moet je zelf ook programmeren. Neem even de tijd om een stuk papier te pakken en de volgende oefening te doen.

Oefening 3.12: Steeds als een programma geschreven wordt, moet het met de hand gekontroleerd worden, om er van verzekerd te zijn dat de resultaten juist zijn. Dit gaan we nu doen: de bedoeling van deze opgave is, dat de tabel van fig. 3-12 ingevuld wordt.

Je kunt deze tabel zelf invullen of anders er een kopie van maken. De bedoeling is, dat de inhoud van de betrokken registers en geheugenplaatsen bepaald wordt, na iedere instruktie die door het programma uitgevoerd wordt, van begin tot einde. Horizontaal zijn in fig. 3-12 de registers uitgezet die door het programma gebruikt worden: X, A, VMR, C (de carryvlag), TIJD, VMT, RESL en RESH. Links in de tabel moet je de instruktie en eventueel het label invullen. Rechts in de

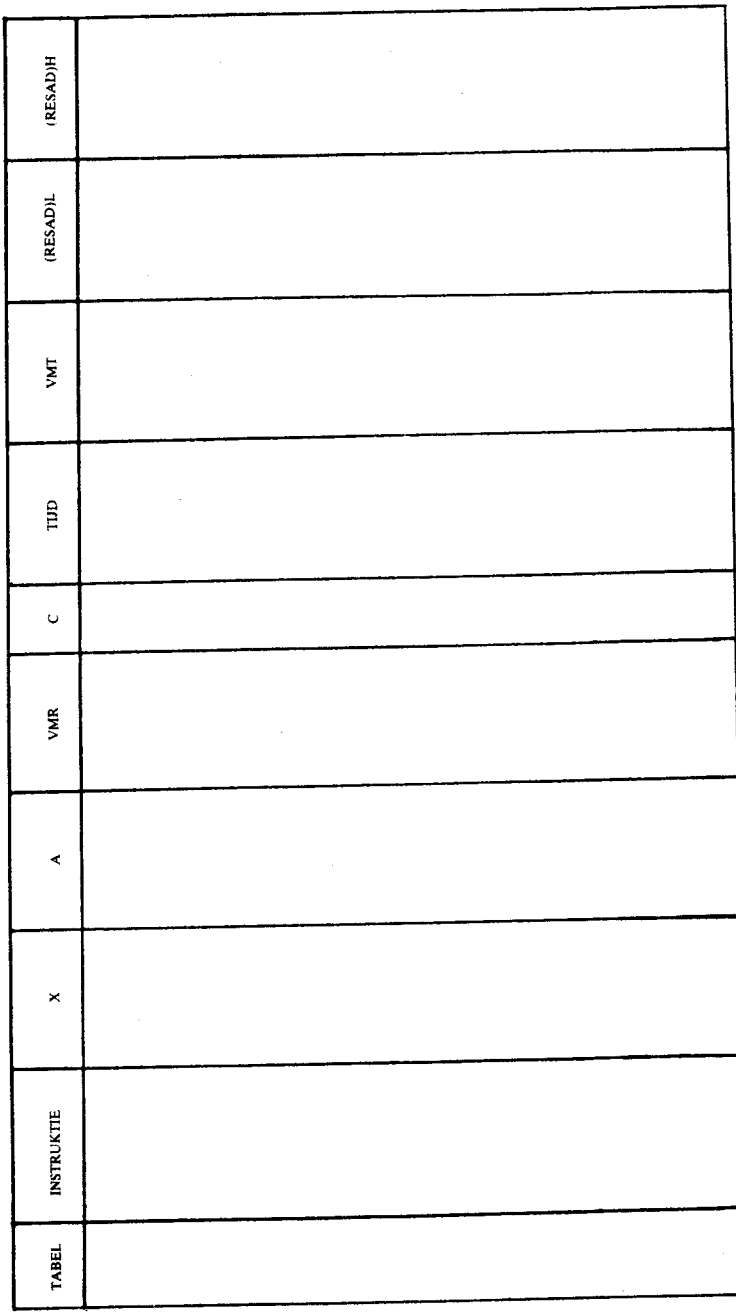


Fig.3-12: Formulier behorende bij oefening 3.12.

tabel moet je de inhoud van de registers invullen nadat de betreffende instruktie uitgevoerd is. Als de inhoud van een register niet gedefinieerd is, gebruiken we streepjes. We zullen nu samen het begin van deze tabel invullen, de rest moet je zelf invullen.

De eerste regel is als volgt:

LABEL	INSTRUKTIE	X	A	VMR	C	TIJD	VMT	(RES)L	(RES)H
	LDA #0	-----	00000000	00000011	--	-----	00000101	-----	-----

Fig. 3-13: Eerste instruktie van vermenigvuldiging

De eerste instruktie die uitgevoerd moet worden is LDA #0.

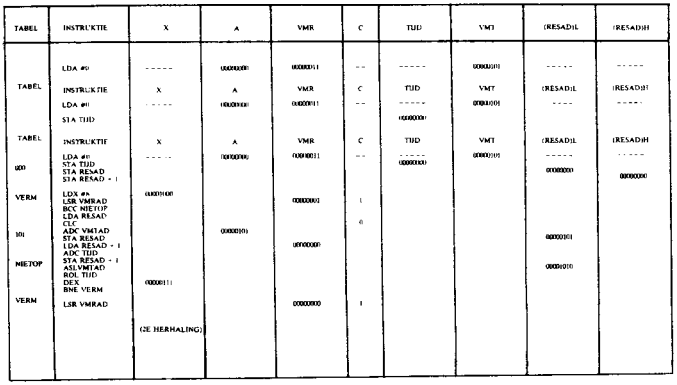
Nadat deze instruktie uitgevoerd is, is de inhoud van register X onbekend. Dit is aangegeven door de streepjes. De accumulator bevat allemaal nullen. We nemen aan, dat de vermenigvuldiger en het vermenigvuldigtal al door de programmeur geladen zijn, voordat het programma uitgevoerd wordt. (Anders zouden er extra instrukties nodig zijn om de inhoud van VMR en VMT te zetten.) VMR heeft de binaire waarde "3". VMT heeft de binaire waarde "5". Het carrybit is ongedefinieerd. En de beide registers die voor RES gebruikt worden zijn ook ongedefinieerd. We vullen nu de volgende regel in. Hij is hieronder gegeven: het enige verschil is, dat de inhoud van register TIJD nul gemaakt is. De volgende instruktie maakt de inhoud van register RESAD "0", en de daaropvolgende maakt de inhoud van RESAD+1 "0".

LABEL	INSTRUKTIE	X	A	VMR	C	TIJD	VMT	(RES)L	(RES)H
	LDA #0	-----	00000000	00000011	--	-----	00000101	-----	-----
	STA TEMP					00000000			

Fig. 3-14: Eerste twee regels van vermenigvuldiging

De vijfde instruktie is LDX #8, en laadt "8" in X. We zullen nog een instruktie doen (zie fig. 3-15).

De instruktie LSR VMRAD schuift de inhoud van VMRAD een positie naar rechts. Je kunt zien, dat na het schuiven de inhoud van VMR "0000 0001" is. De meest rechtse "1" van VMR is in het carrybit gevallen. Bit C is nu "1" geworden. De andere registers zijn niet veranderd.



TABEL	INSTRUKTIE	X	A	VMR	C	TIJD	VMT	(RESADIL	(RESADIH
	LDA #0	-----	00000000	00000011	--	-----	00000000	-----	-----
TABEL	INSTRUKTIE	X	A	VMR	C	TIJD	VMT	(RESADIL	(RESADIH
	LDA #11	-----	00000011	00000011	--	-----	00000000	-----	-----
	STA TIJD					00000011			
TABEL	INSTRUKTIE	X	A	VMR	C	TIJD	VMT	(RESADIL	(RESADIH
100	LDA #0 STA TIJD STA RESAD STA RESAD + 1	-----	00000000	00000011	--	----- 00000000	00000000	----- 00000000	----- 00000000
VERM	LDX #8 LSR VMRAD BCC NIETOP LDA RESAD CLC	00001000		00000001	1 0				
101	ADC VMTAD STA RESAD LDA RESAD + 1 ADC TIJD		00000101	00000000				00000101	
NIETOP	STA RESAD + 1 ASL VMTAD ROL TIJD DEX	00000111						00001010	
VERM	BNE VERM LSR VMRAD			00000000	1				
		(ZIE HERHALING)							

Fig. 3-15: Gedeeltelijk ingevulde tabel voor oefening 3.12

Nu is het jouw beurt. Vul de rest van de tabel helemaal in. Het is niet moeilijk, maar vereist wel de nodige aandacht. Als je twijfels hebt betreffende de werking van bepaalde instrukties, kun je hoofdstuk 4 raadplegen, waar de instrukties beschreven staan.

Het uiteindelijke resultaat van de vermenigvuldiging moet binair "15" zijn, opgeslagen in de registers RES laag en RES hoog. RES hoog moet "00000000" bevatten en RES laag moet "00001111" bevatten. Als dit je uitkomst is, heb je gewonnen. Probeer het anders nog eens. De meest voorkomende fout is een foutief gebruik van het carrybit. Verzeker je ervan, dat het carrybit na iedere rekenkundige bewerking verandert. Vergeet niet, dat de RLE na iedere optelbewerking het carrybit zet.

Programmeeralternatieven

Het programma dat we net ontwikkeld hebben, is maar een van de vele manieren waarop het geschreven had kunnen worden. Iedere programmeur kan een manier vinden om het programma te veranderen, en soms om het te verbeteren. Zo hebben we bijvoorbeeld het vermenigvuldigtal naar links geschoven voor de optelling. Wiskundig gezien is het hetzelfde als we het resultaat een positie naar rechts zouden schuiven voor de optelling bij het vermenigvuldigtal. Het voordeel is, dat we register TIJD niet nodig zouden hebben en dus een geheugenplaats zouden uitsparen. Deze methode zou de voorkeur verdienen bij een microprocessor die met voldoende interne registers uitgerust is om VMR, VMT en RES in de microprocessor zelf op te slaan. Omdat we nu het geheugen wel moeten gebruiken om dit te doen, is het niet inte-

ressant om een geheugenplaats uit te sparen. De vraag is dus, of de tweede methode een snellere vermenigvuldiging tot gevolg heeft. Dit is een interessante oefening:

Oefening 3.13: Schrijf nu een 8 × 8 vermenigvuldiging, gebruik makend van hetzelfde algoritme, maar schuif het resultaat een positie naar rechts in plaats van het vermenigvuldigtal een positie naar links. Vergelijk dit met het vorige programma en bepaal of deze aanpak sneller of langzamer is dan de voorgaande.

Hierbij kan zich een probleem voordoen: Om de snelheid van een programma vast te stellen, zijn de tabellen in de Appendix nodig, waarin vermeld staat hoeveel cycli iedere instruktie nodig heeft. Dit aantal hangt voor sommige instrukties af van hun plaats. De 6502 kent een speciale adresseermode, de *direkte adresseringsmode*, waarbij de eerste pagina (geheugenplaatsen 0 tot 255) gereserveerd is voor snelle uitvoering. Dit wordt in hoofdstuk 5 uitgelegd bij de adresseertechnieken. In het kort: Alle programma's, die snel uitgevoerd moeten worden, zullen variabelen hebben die in pagina 0 zijn geplaatst, zodat de instrukties maar twee bytes nodig hebben om geheugenplaatsen te adresseren (256 plaatsen vereisen slechts een byte), terwijl instrukties op andere plaatsen in het geheugen, in de meeste gevallen 3 bytes lang zullen zijn. Maar laten we deze analyse uitstellen tot hoofdstuk 5.

Een verbeterd vermenigvuldigingsprogramma

Het zojuist ontwikkelde programma is een rechtstreekse vertaling van het algoritme in kode. Om echter doelmatig te programmeren, moet nauwkeuriger op details gelet worden, om het programma korter en sneller te maken. We zullen nu een verbeterde uitvoering van hetzelfde algoritme geven.

Een van de opdrachten, die veel instrukties en tijd kost, is het schuiven van het resultaat en de vermenigvuldiger. Een standaard "truuk" die in het vermenigvuldigings-algoritme gebruikt wordt is gebaseerd op de volgende waarneming: iedere keer als de vermenigvuldiger een positie naar rechts geschoven wordt, komt er links een bitpositie vrij. Tegelijkertijd kunnen we waarnemen, dat het eerste resultaat (of deelprodukt) maximaal 9 bits nodig heeft. Na het schuiven voor de volgende vermenigvuldiging is de lengte van het deelprodukt met een bit toegenomen. Met andere woorden, we hoeven in eerste instantie maar een geheugenplaats te reserveren voor het deelprodukt, en kunnen

dan de bitposities gebruiken die vrijkomen als de vermenigvuldiger geschoven wordt.

We schuiven nu de vermenigvuldiger naar links. Hierbij komt rechts een bitpositie vrij. We plaatsen nu het meest rechtse bit van het deelprodukt in deze vrijgekomen bitpositie. Laten we nu het programma eens beschouwen.

Laten we ook een optimaal gebruik van de registers in aanmerking nemen. De interne registers van de 6502 zijn getekend in fig. 3-16. X kan het beste als teller gebruikt worden. We zullen hem gebruiken om het verschoven aantal bits te tellen. De accumulator is (helaas) het enige interne register waarmee geschoven kan worden. Om de doelmatigheid te verhogen, moeten we hierin de vermenigvuldiger of het resultaat opslaan.

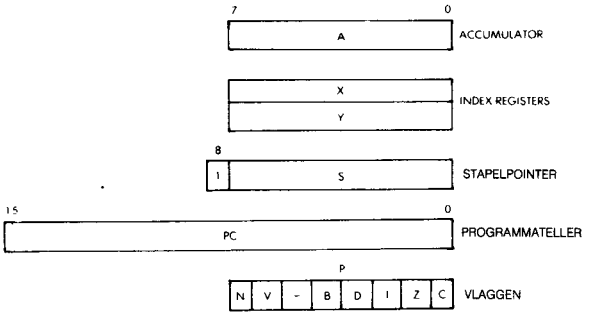


Fig. 3-16: 6502 Registers

Welke moeten we in de accumulator opslaan? Iedere keer als er een 1 uitgeschoven wordt, moet het resultaat bij het vermenigvuldigtal opgeteld worden. Aangezien de 6502 ook alleen maar iets bij de accumulator op kan tellen, moet het resultaat in de accumulator komen.

De andere getallen moeten in het geheugen opgeslagen worden. (Zie fig. 3-17.)

In A en B komt het resultaat. In A komt het hoge deel en in B het lage deel van het resultaat. A is de accumulator en B is een geheugenplaats, bij voorkeur in pagina 0. In C (een geheugenplaats) komt het verme-

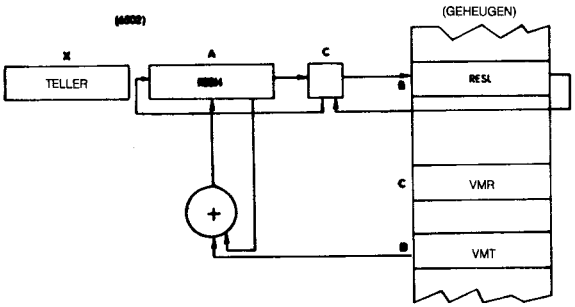


Fig.3-17: Toewijzing van registers (verbeterde vermenigvuldiging)

nigvuldigtal. Het programma is als volgt:

```
MULT    LDA  #0      INITIALIZE RESULT TO ZERO (HIGH)
        STA  B       INITIALIZE RESULT (LOW)
        LDX  #8      X = TELLER
LOOP    LSR  C       SCHUIF VMR
        BCC  NIETOP
        CLC          CLEAR CARRY
        ADC  D       A = A + MPD
NIETOP  ROR  A       SCHUIF RES
        ROR  B       VANG BIT IN B
        DEX          VERLAAG TELLER
        BNE  LOOP    LAATSTE VERSCHUIVING
```

Fig. 3-18: Verbeterde vermenigvuldiging

Laten we dit proramma eens bekijken. Omdat A en B het resultaat zullen bevatten, moeten ze geinitialiseerd worden met de waarde nul. Dit doen we als volgt:

```
VERM  LDA #0
      STA B
```

We gebruiken register X als teller en geven die de beginwaarde 8:

```
      LDX  #8
```

We zijn nu klaar om de hoofdlus in te gaan. We schuiven eerst de vermenigvuldiger en testen dan het carrybit, waarin het uitgeschoven,

meest rechtse, bit van de vermenigvuldiger zit:

```
LOOP  LSR C
      BCC NIETOP
```

Hier schuiven we de vermenigvuldiger naar links (in plaats van naar rechts zoals eerst). Dit is gelijkwaardig met het voorgaande algoritme, omdat de optelbewerking omwisselbaar is.

Er zijn nu twee mogelijkheden: als de carry 0 was, springen we naar NIETOP. Laten we aannemen, dat de carry 1 was. We gaan verder:

```
      CLC
      ADC D
```

Omdat de carry 1 was, moet hij op nul gezet worden. Daarna tellen we het vermenigvuldigtal bij de accumulator op. (De accumulator bevat het resultaat, tot nu toe 0.)
We schuiven nu het deelprodukt:

```
NIETOP  ROR A
        ROR B
```

Het deelprodukt in A wordt een bit naar rechts geschoven. Het meest rechtse bit komt in het carrybit. Het carrybit wordt gevangen en in register B geroteerd, waarin het lage deel van het resultaat zit.

We hoeven nu alleen nog maar te testen of we klaar zijn:

```
      DEX
      BNE LOOP
```

Als we dit programma bekijken, blijkt dat we in vergelijking met het vorige programma maar half zoveel instrukties nodig hebben. Het wordt dus ook sneller uitgevoerd. Hiermee is aangetoond, hoe belangrijk het is om de juiste registers te selekteren voor informatieopslag.

Een rechtstreeks ontwerp heeft een werkend programma tot gevolg. Maar het heeft geen optimaal programma tot gevolg. Het is daarom van het uiterste belang om de beschikbare registers en geheugenplaatsen op de best mogelijke manier te gebruiken. Dit voorbeeld illustreert een rationele aanpak van registerselektie voor een maximale doeltreffendheid.

Oefening 3.14: Bereken de snelheid van een vermenigvuldiging, gebruik makend van dit laatste programma. Neem aan, dat in 50% van de gevallen een sprong gemaakt wordt. Zoek het aantal cycli voor iedere instruktie in de tabel achterin dit boek op. Ga uit van een kloksnelheid van een cyclus per 1 microseconde.

Binaire deling

Het algoritme voor een binaire deling is analoog aan dat voor een vermenigvuldiging. De deler wordt achtereenvolgens afgetrokken van de hoge orde bits van het deeltal. Na iedere aftrekking wordt het resultaat gebruikt, in plaats van het oorspronkelijke deeltal. De waarde van het quotient wordt iedere keer gelijktijdig met 1 verhoogd. Uiteindelijk wordt het resultaat van de aftrekking negatief. Dit wordt een "overdraw" genoemd. Het deelresultaat moet dan hersteld worden door de deler er weer bij op te tellen. Natuurlijk moet dan het quotient gelijktijdig met 1 verlaagd worden. Quotient en deeltal worden dan een bitpositie naar links geschoven, en het algoritme wordt herhaald.

De zojuist beschreven methode wordt de *herstellende methode* genoemd. Een variatie op deze methode, die sneller uitgevoerd wordt, is de *niet-herstellende* methode.

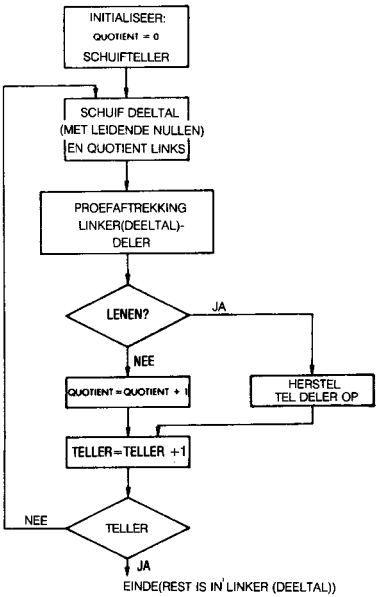


Fig. 3-19: Stroomdiagram voor een 8 bits binaire deling

De 16-bits deling

We zullen nu de niet-herstellende deling voor een 16-bits deeltal en een 8-bits deler beschrijven. Het resultaat telt 8 bits. De toewijzing van de registers en het geheugen voor dit programma zijn te zien in fig. 3-20. Het deeltal is opgeslagen in de accumulator (hoge deel) en geheugenplaats 0, hier B genaamd. Het resultaat staat in Q (geheugenplaats 1). De deler is opgeslagen in D (geheugenplaats 2). Het resultaat zal in Q en A (in A komt de rest) komen.

Het programma is gegeven in fig. 3-21, het stroomdiagram in fig. 3-22.

Oefening 3.15: Verifieer de juiste werking van dit programma door met de hand een deling uit te voeren en het programma te doorlopen als in oefening 3.12. Deel 33 door 3. Het resultaat moet natuurlijk 11 zijn, met rest 0.

Logische bewerkingen

De ander klasse instrukties, die de RLE in de microprocessor kan uitvoeren,is de verzameling logische instrukties. Dit zijn onder andere: AND, OR en exclusive OR (EOR). Verder kunnen hiertoe gerekend worden de schuifbewerkingen die we al gebruikt hebben, en de vergelijk instruktie, die bij de 6502 CMP genoemd wordt. Het gebruik van AND, ORA en EOR wordt beschreven in Hoofdstuk 4 bij de instruktieset. Laten we nu een kort programma schrijven, dat kontroleert of een gegeven geheugenplaats LOC de waarde "0", "1", of iets anders bevat. Het programma is als volgt:

```
        LDA     LOC     INHOUD VAN LOC NAAR DE
                        ACCUMULATOR
        CMP     #$00    VERGELIJK MET 0
        BEQ     NUL     IS HET 0?
        CMP     #$01    1?
        BEQ     EEN
NIET    ...
        ...
NUL     ...
        ...
EEN     ...
        ...
```

De eerste instruktie: LDA LOC leest de inhoud van geheugenplaats LOC. Dit is het karakter dat we willen testen.
CMP #$00

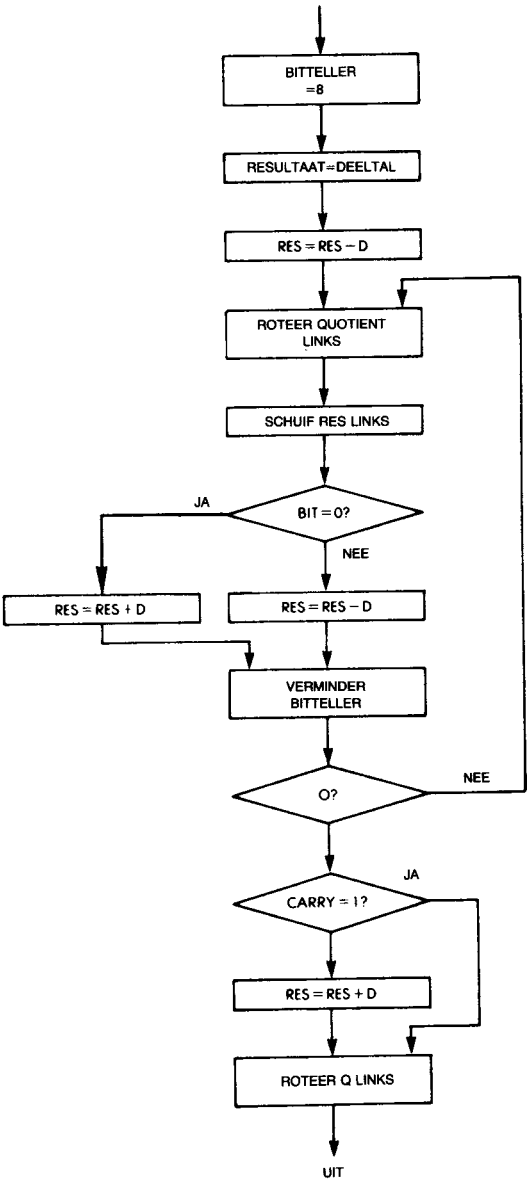


Fig. 3-20: Stroomdiagram 16 door 8 deling.

```
LINE  #  LOC     CODE        LINE
0002     0000                    * = $0
0003     0000                B   * = * + 1
0004     0001                Q   * = * + 1
0005     0002                D   * = * + 1
0006     0003                    * = $200
0007     0200     A0 08      DIV  LDY #8
0008     0202     38              SEC
0009     0203     E5 02           SBC D
0010     0205     08         LOOP PHP
0011     0206     26 01           ROL Q
0012     0208     06 00           ASL B
0013     020A     2A              ROL A
0014     020B     28              PLP
0015     020C     90 05           BCC ADD
0016     020E     E5 02           SBC D
0017     0210     4C 15 02        JMP NEXT
0018     0213     65 02      ADD  ADC D
0019     0215     88         NEXT DEY
0020     0216     D0 ED           BNE LOOP
0021     0218     B0 03           BCS LAST
0022     021A     65 02           ADC D
0023     021C     18              CLC
0024     021D     26 01      LAST ROL Q
0025     021F     00              BRK
0026     0220                     END
```

Fig. 3-21: Programma.

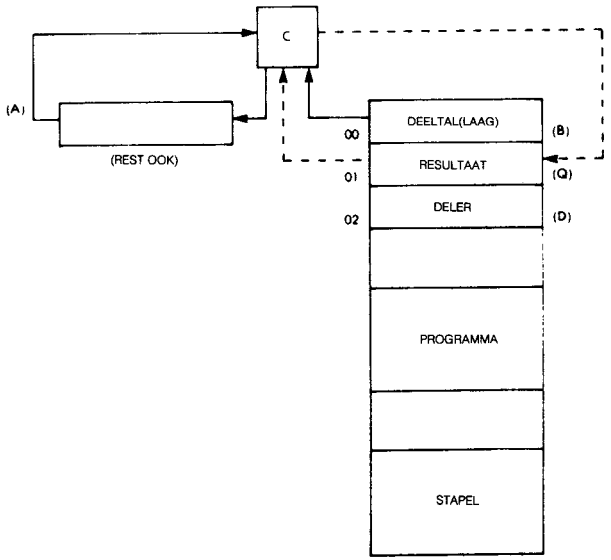


Fig. 3.22: Stroomdiagram 16 door 8 deling (niet herstellend)

Deze instruktie vergelijkt de inhoud van de accumulator met de literal "00" hexadecimaal, dus het bitpatroon "0000 0000". Als de inhoud van de accumulator overeenstemt met het gegeven bitpatroon, hier "0000 0000", wordt de Z bit in het statusregister op 1 gezet. Indien er geen overeenstemming is wordt deze Z bit op nul gezet. De Z bit kan dan getest worden met volgende instruktie:

```
BEQ NUL
```

De BEQ instruktie betekent "Branch if EQual"(Spring als gelijk). De spronginstruktie bepaald of de test succesvol is door het Z bit te kontroleren. Als dit gezet is, springt het programma naar NUL. Anders wordt de volgende instruktie uitgevoerd:

```
CMP #$01
```

Het proces wordt herhaald met het nieuwe patroon. Als de test slaagt, heeft de volgende instruktie een sprong naar plaats een tot gevolg. Anders wordt de volgende instruktie uitgevoerd.

Oefening 3.16: Schrijf een programma, dat de inhoud van geheugenplaats "24" leest en naar adres STER springt als geheugenplats 24 een "*" bevatte. Het bitpatroon voor een "*" in assembleertaal-notatie is "0010 1010".

SAMENVATTING

We hebben nu de belangrijkste instrukties van de 6502 bestudeerd door ze te gebruiken. We hebben waarden overgedragen tussen het geheugen en de registers. We hebben rekenkundige en logische bewerkingen op gegevens uitgevoerd. We hebben getest en afhankelijk van het resultaat hiervan verschillende delen van het programma uitgevoerd. We hebben ook een lusstruktuur bij het vermenigvuldigingsprogramma geintroduceerd. We zullen nu een belangrijke programmeerstruktuur introduceren: de subroutine.

SUBROUTINES

In principe is een subroutine een blok instrukties, waaraan door de programmeur een naam gegeven is. Om praktisch redenen moet een subroutine beginnen met een speciale instruktie, de subroutine-deklaratie genaamd, waarmee deze als zodanig voor de assembler gedefinieerd wordt. Hij moet ook afgesloten worden door een speciale in-

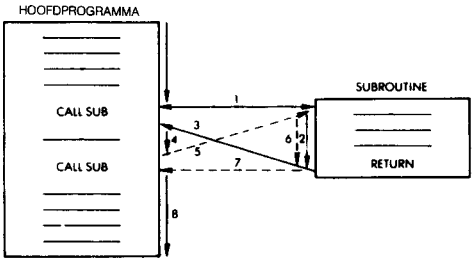


Fig. 3-23: Aanroep van subroutines

struktie, de *return* (terugkeer) genaamd. Laten we eerst het gebruik van subroutines in een programma illustreren, om zo hun waarde aan te tonen. Daarna zullen we onderzoeken hoe ze eigenlijk uigevoerd zijn.

Het gebruik van een subroutine is geillustreerd in fig. 3-23. Het hoofdprogramma is links getekend. De subroutine is symbolisch rechts weergegeven. Laten we het subroutine-mechanisme eens onderzoeken. De regels van het hoofdprogramma worden na elkaar uitgevoerd, tot een nieuwe instruktie genaamd "CALL SUB". Deze speciale instruktie is de *subroutine-oproep* (call), en heeft een overdracht naar de subroutine tot gevolg. Dit betekent, dat de volgende instruktie die uitgevoerd wordt na de CALL SUB, de eerste instruktie van de subroutine is. Dit is aangegeven met pijl 1 in de illustratie.

Daarna wordt het subprogramma in de subroutine net als een gewoon programma uitgevoerd. We nemen aan, dat de subroutine geen andere oproepen bevat. De laatste instruktie van deze subroutine is een RETURN. Dit is een speciale instruktie, waarmee naar het hoofdprogramma teruggekeerd wordt. De volgende instruktie die uitgevoerd wordt is degene die op de CALL SUB volgt. Dit is aangegeven met pijl 3 in de illustratie. De uitvoering van het programma gaat dan verder als aangegeven door pijl 4.

In het hoofdprogramma staat nog een tweede CALL SUB. Er vindt een nieuwe overdracht plaats, aangegeven door pijl 5. Dit betekent, dat de subroutine na de CALL SUB instruktie nogmaals uitgevoerd wordt.

Als de RETURN instuktie in de subroutine gevonden wordt, vindt terugkeer plaats naar de instruktie na de betrokken CALL SUB. Dit is aangegeven door pijl 7. Na de terugkeer naar het hoofdprogramma, gaat de uitvoering van het programma gewoon door, zoals geillustreerd door pijl 8.

De funktie van de twee speciale instrukties CALL SUB en RETURN zou nu duidelijk moeten zijn. Wat is de waarde van een subroutine?

De essentiele waarde van een subroutine is, dat hij vanuit ieder aantal punten in het programma aangeroepen kan worden, en herhaald gebruikt kan worden, zonder dat hij herschreven hoeft te worden. Het eerste voordeel is, dat deze aanpak geheugenruimte bespaart, en de subroutine niet steeds herschreven hoeft te worden. Een tweede voordeel is, dat de programmeur een specifieke subroutine maar eenmaal hoeft te ontwerpen, en hem dan herhaald kan gebruiken. Dit is een aanzienlijke verbetering in het ontwerp van programma's.

Oefening 3.17: Wat is het voornaamste nadeel van een subroutine?

Het nadeel van de subroutine zou duidelijk moeten worden door de stroom van de uitvoering van afwisselend het hoofdprogramma en de subroutine te bekijken. Een subroutine wordt langzamer uitgevoerd, omdat er twee extra instrukties uitgevoerd moeten worden: de CALL SUB en de RETURN.

De uitvoering van het subroutine-mechanisme

We zullen hier onderzoeken hoe de twee speciale instrukties, CALL SUB en RETURN, intern in de processor uitgevoerd zijn. Het gevolg van de CALL SUB instruktie is, dat de volgende instruktie van een nieuw adres gehaald wordt. Je zult je herinneren, dat het adres van de volgende uit te voeren instruktie in een computer in de programmateller opgeslagen is (zie hoofdstuk 1). Dit betekent, dat het gevolg van de CALL SUB het vervangen van de inhoud van de programmateller (PC) is. Het effekt is, dat het startadres van de subroutine in de programmateller geladen wordt. *Maar is dit voldoende?*

Laten we om deze vraag te beantwoorden de andere instruktie die uitgevoerd moet worden eens beschouwen: de RETURN. De RETURN moet, zoals zijn naam al aangeeft, een terugkeer bewerkstelligen naar de instruktie die op de CALL SUB volgt. Dit is alleen mogelijk als het adres van deze instruktie ergens bewaard wordt. Dit adres is nu net de inhoud van de programmateller op het moment dat de CALL SUB gevonden werd. Dit is zo, omdat de programmateller, iedere keer als hij gebruikt wordt, automatisch verhoogd wordt (lees hoofdstuk 1 nog eens!). Dit is precies het adres dat we willen bewaren, zodat we later een RETURN kunnen uitvoeren.

Het volgende probleem is: waar kunnen we dit terugkeeradres bewaren? Het adres moet op een redelijke plaats opgeborgen worden, waar gegarandeerd is, dat het niet uitgewist wordt. Laten we nu echter de situatie, geillustreerd in figuur 3-22 eens beschouwen: in dit voorbeeld roept subroutine 1 subroutine 2 aan. Ons mechanisme moet in dit geval ook werken. Er zouden natuurlijk meer dan twee subroutines kunnen zijn, zeg N "geneste" aanroepen. Steeds als een nieuwe CALL gevonden wordt, moet het mechanisme de programmateller opbergen. Dit impliceert, dat we tenminste 2N geheugenplaatsen nodig hebben voor dit mechanisme. Verder moeten we eerst van SUB2 terugkeren en dan van SUB1. Met andere woorden, we hebben een struktuur nodig, die de tijdsvolgorde bewaart waarin de gegevens opgeslagen zijn.

Deze struktuur heeft al een naam. We hebben hem al geintroduceerd. Het is de stapel. Figuur 3-26 laat de inhoud van de stapel zien tijdens opeenvolgende subroutineaanroepen. Laten we eerst naar het hoofdprogramma kijken. Op adres 100 komen we de eerste aanroep tegen: CALL SUB1. We zullen ervan uitgaan, dat de aanroep van de subroutine in deze microprocessor drie bytes lang is. Het volgende adres is dus niet "101", maar "103".

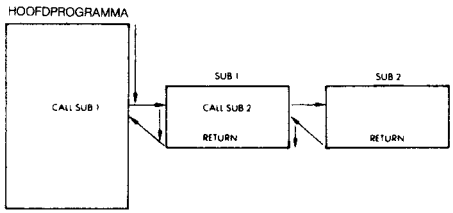


Fig. 3-24: Geneste aanroepen

De CALL instruktie gebruikt adressen "100", "101" en "102". Omdat de kontrole-eenheid van de 6502 "weet" dat het een 3-byte instruktie is, zal de waarde van de programmateller, nadat de aanroep helemaal gedekodeerd is, "103" zijn. Het gevolg van de aanroep is, dat de waarde "280" in de progammateller geladen wordt. "280" is het startadres van SUB1.

Het gevolg van de CALL is, dat de waarde "103" van de programmateller op de stapel gezet wordt. Dit is links onderin de tekening geillustreerd, waar te zien is, dat op tijdstip 1 de waarde "103" in de stapel bewaard wordt. Laten we nu eens rechts onderin de tekening

kijken. Op plaats 300 komen we een nieuwe CALL tegen. Net als in het voorgaande geval wordt nu "900" in de programmateller geladen. Dit is het startadres van SUB2. Gelijktijdig wordt de waarde "303" op de stapel gezet. Dit is links onderin de tekening aangegeven, waar op tijdstip 2 de waarde "303" binnenkomt. Uitvoering gaat dan door rechts in de figuur, binnen SUB2.

We zijn nu klaar om het effekt van de RETURN instruktie te demonstreren en de juiste werking van ons stapelmechanisme. De uitvoering binnen SUB2 gaat door, tot op tijdstip 3 de RETURN instruktie gevonden wordt. Het effekt van de RETURN instruktie is eenvoudig de top van de stapel naar de programmateller te transfereren. Met andere woorden, de programmateller krijgt de waarde weer die hij had, voordat de subroutine ingegaan werd. De top van de stapel in ons voorbeeld is "303". Figuur 3-26 laat zien dat op tijdstip 3 de waarde "303" van de stapel gehaald is en weer terug in de programmateller geladen is. Het resultaat is, dat de uitvoering van instrukties verder gaat bij adres "303". Op tijdstip 4 komen we de RETURN van SUB1 tegen. De waarde bovenin de stapel is "103". Deze wordt opnieuw naar de programmateller getransfereerd. Het resultaat is, dat het programma verder gaat vanaf plaats "103" in het hoofdprogramma. Dit is

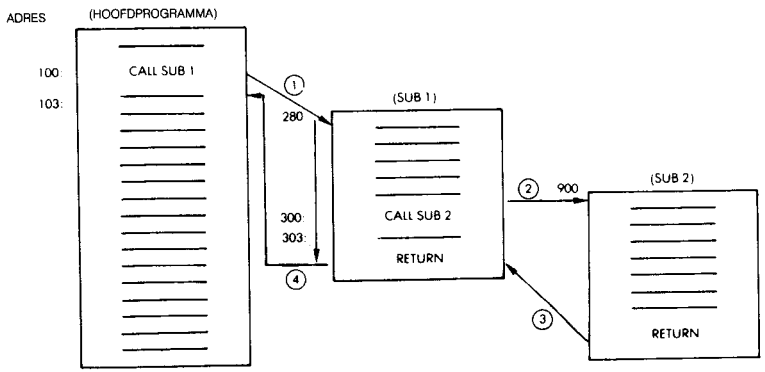


Fig. 3-25: Subroutine aanroepen

precies het effekt dat we willen hebben. In figuur 3-26 is te zien, dat op tijdstip 4 de stapel leeg is. Het mechanisme werkt.

Het subroutine mechanisme werkt door tot de maximale diepte van de stapel. Om deze reden waren de oudere microprocessoren, met een

stapel van 4 of 8 registers, beperkt tot 4 of 8 niveau's van subroutine-aanroepen. In theorie kan de 6502, wiens stapel beperkt is tot 256 plaatsen, dus tot 128 subroutine aanroepen verzorgen. Dit is alleen waar, als er geen interrupts zijn, als de stapel niet voor andere doeleinden gebruikt wordt, en als er geen registers in de stapel opgeslagen hoeven te worden. In de praktijk zullen er minder niveau's gebruikt worden.

Merk op, dat in fig. 3-24 en 3-25 de subroutines rechts van het hoofdprogramma getekend zijn. Dit is alleen voor de duidelijkheid van het diagram. In werkelijkheid worden de instrukties door de gebruiker als gewone instrukties van het programma ingevoerd.

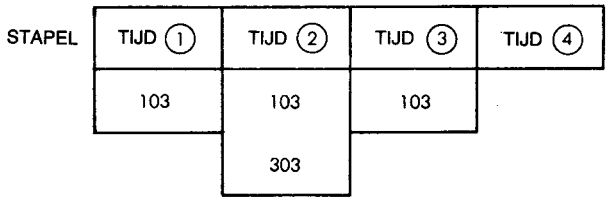


Fig. 3-26: Stapelgedrag in de tijd

Op een vel papier, de listing van het volledige programma, kunnen de subroutines aan het begin, in het midden, of aan het einde van de tekst staan. Om deze reden moeten ze voorafgegaan worden door een subroutine-deklaratie: ze moeten geidentificeerd worden. De speciale instruktie maakt de assembler duidelijk, dat wat volgt als een subroutine behandeld moet worden. Dergelijke aanwijzingen voor de assembler worden besproken in hoofdstuk 9.

6502 Subroutines

We hebben zojuist het subroutine-mechanisme besproken, en hoe de stapel gebruikt wordt om dit uit te voeren. De subroutine-aanroep instruktie voor de 6502 wordt JSR genoemd (jump to subroutine = spring naar subroutine). Het is inderdaad een 3-byte instruktie. Het is helaas een onvoorwaardelijke sprong: hij test geen bits. Als er testen uitgevoerd moeten worden, moeten voor de JSR expliciete sprongen tussengevoegd worden.

De terugkeer van een subroutine is de RTS instruktie (return van subroutine). Het is een 1-byte instruktie.

Oefening 3.18: Waarom is de terugkeer van een subroutine net zo snel als de CALL? (Hint: als het antwoord niet voor de hand ligt, bekijk dan nog eens hoe het subroutine-mechanisme uitgevoerd is met behulp van de stapel, en analyseer de interne bewerkingen die verricht moeten worden.)

Subroutine voorbeelden

De meeste programma's die we ontwikkeld hebben, en nog zullen ontwikkelen, zouden in de meeste gevallen als subroutines geschreven zijn. Zo is het waarschijnlijk, dat het vermenigvuldigingsprogramma bijvoorbeeld in veel programma's gebruikt zal worden. Om nu het ontwikkelen van programma's te vereenvoudigen en te verduidelijken, is het handig om een subroutine te definieren, waarvan de naam bijvoorbeeld VERMEN is. Aan het eind van deze subroutine voegen we eenvoudig de instruktie RTS toe.

Oefening 3.19: Als VERMEN als een subroutine gebruikt wordt, beschadigt hij dan interne vlaggen of registers?

Recursie

Het woord recursie wordt gebruikt om aan te geven, dat een subroutine zichzelf aanroept. Als je begrepen hebt, hoe het mechanisme uitgevoerd is, zou je nu de volgende vraag moeten kunnen beantwoorden:

Oefening 3.20: Mag een subroutine zichzelf aanroepen? (Met andere woorden, werkt alles zelfs als een subroutine zichzelf aanroept?). Als je hier niet zeker van bent, teken dan de stapel en vul hem met opeenvolgende adressen. Je kunt nu nagaan of het werkt of niet. Hiermee is de vraag beantwoord. Bekijk vervolgens het geheugen en de registers en bepaal of er een probleem ontstaat (zie oefening 3.19).

Subroutine parameters

Als een subroutine aangeroepen wordt, is in de meeste gevallen te verwachten dat hij met bepaalde gegevens werkt. Bij de vermenigvuldiging bijvoorbeeld, wil je twee getallen doorgeven aan de subroutine die de vermenigvuldiging uitvoert.
We hebben bij de vermenigvuldigingsroutine gezien dat deze routine

de vermenigvuldiger en het vermenigvuldigtal op gegeven plaatsen verwacht. Dit illustreert de eerste methode om parameters door te geven: via het geheugen. Er worden nog twee andere technieken toegepast en parameters kunnen dus op drie manieren doorgegeven worden:

1 - Via registers
2 - Via het geheugen
3 - Via de stapel

– *Registers* kunnen gebruikt worden om parameters door te geven. Dit is een voordelige oplossing, vooropgesteld dat registers beschikbaar zijn, omdat je geen vaste geheugenplaats nodig hebt. De subroutine blijft onafhankelijk van het geheugen. Als een vaste geheugenplaats wordt gebruikt, moet iedere andere gebruiker van de routine erg voorzichtig zijn en opletten of de geheugenplaats ook inderdaad vrij is (zie oefening 3-20). Om deze reden wordt in veel gevallen een deel van het geheugen gereserveerd om parameters tussen de verschillende subroutines door te geven.
– Het gebruik van het *geheugen* heeft het voordeel van een grotere flexibiliteit (meer gegevens), maar heeft een geringere prestatie tot gevolg, en legt de subroutine ook vast op een bepaald geheugengebied.
– Het deponeren van gegevens in de *stapel* heeft hetzelfde voordeel als het gebruik van registers: het is onafhankelijk van het geheugen. De subroutine weet dat hij bijvoorbeeld twee parameters moet ophalen die bovenop de stapel staan. Er is natuurlijk een nadeel: de stapel staat vol met gegevens en daarmee is het aantal subroutine-niveau's verminderd.

De keus is aan de programmeur. In het algemeen is het wenselijk om zolang mogelijk onafhankelijk van vaste geheugenplaatsen te blijven.

Als er geen registers beschikbaar zijn, is de beste oplossing meestal de stapel. Als er echter veel gegevens aan de subroutine doorgegeven moeten worden, dan moet deze informatie in het geheugen geplaatst worden. Een elegante oplossing van het probleem om een blok gegevens door te geven is het doorgeven van een *pointer* (wijzer) naar de informatie. Een pointer is het beginadres van een blok. Een pointer kan via een register doorgegeven worden (bij de 6502 is de pointer dan beperkt tot 8 bits), of anders via de stapel (twee stapelplaatsen kunnen gebruikt worden om een 16 bits adres op te slaan)

Tenslotte kan, als geen van deze twee methoden beschikbaar is, met de subroutine afgesproken worden dat de gegevens op een vaste geheugenplaats zullen staan (de "postbus").

Oefening 3.21: Welke van de twee genoemde methoden leent zich het best voor recursie?

Subroutine bibliotheek

Het structureren van delen van een programma in aparte subroutines heeft een groot voordeel: ze kunnen onafhankelijk van elkaar verbeterd worden en kunnen een mnemonische naam hebben. Subroutines kunnen dan gegroepeerd worden tot een ware bibliotheek. Echter het systematisch gebruikmaken van subroutines voor een verzameling van instrukties die door hun funktie kunnen worden gegroepeerd, kan leiden tot een slechte efficientie. Het is de taak van de programmeur om de voor- en nadelen af te wegen.

Samenvatting

Steeds ingewikkelder algoritmes zijn geintroduceerd en vervolgens omgezet in programma's. De voornaamste typen instrukties zijn gebruikt.

We hebben belangrijke strukturen als lussen, stapels en subroutines gedefinieerd.

Je zou nu de grondbeginselen van het programmeren moeten begrijpen, evenals de voornaamste technieken die in standaard toepassingen gebruikt worden. We gaan nu de beschikbare instrukties bestuderen.

4
DE INSTRUKTIESET VAN DE 6502

DEEL 1 - ALGEMENE BESCHRIJVING

INLEIDING

In dit hoofdstuk worden de diverse klassen van instrukties geanalyseerd, die in het algemeen in een computer beschikbaar zouden moeten zijn. Vervolgens worden de op de 6502 beschikbare instrukties stuk voor stuk geanalyseerd en wordt in detail beschreven hoe ze gebruikt kunnen worden en wat hun effekt op de vlaggen is. Verder gaan we in op de samenhang met de verschillende adresseermodes. Een gedetailleerde bespreking van de adresseertechnieken wordt gegeven in hoofdstuk 5.

INSTRUKTIEKLASSEN

Instrukties kunnen op veel manieren geklassificeerd worden, en er is geen standaard. We zullen hier vijf kategorieen instrukties onderscheiden:

1 - gegevensoverdracht
2 - gegevensverwerking
3 - test en spring
4 - invoer/uitvoer
5 - kontrole

We zullen nu achtereenvolgens ieder van deze klassen onderzoeken.

Gegevensoverdracht

Gegevensoverdracht-instrukties dragen 8-bits gegevens over van register naar register, van register naar geheugen of terug, of van een register naar een uitvoerrandapparaat, of van een invoerrandapparaat naar een register. Voor registers die een specifieke rol spelen, kunnen er speciale instrukties zijn. Bijvoorbeeld om transferten van gevens van en naar de stapel efficient uit te voeren. Ze verplaatsen een gegeven tussen de top van de stapel en een register in een enkele bewerking, terwijl automatisch de stapelwijzer bijgewerkt wordt.

Gegevensverwerking

Instrukties om gegevens te verwerken vallen uiteen in vier algemene kategorieen:

- rekenkundige bewerkingen (zoals optellen)
- logische bewerkingen (zoals AND, OR, exclusive OR)
- schuif- en verplaatsbewerkingen (zoals schuif, roteer)
- vermeerderen en verminderen (zoals DEX)

Opgemerkt moet worden, dat het voor een effektieve verwerking van gegevens wenselijk is om krachtige rekenkundige instrukties te hebben als bijvoorbeeld vermenigvuldigen en delen. Helaas kennen de meeste microprocessoren deze niet. Ook is het wenselijk om krachtige schuif-en roteer-instrukties te hebben zoals: verschuif N bits, of een nibble (= 4 bits) verwisseling, waarbij de linker en de rechterhelft en een byte verwisseld worden. Ook deze instrukties hebben de meeste microprocessoren niet.
Voordat we de 6502 instrukties onderzoeken, willen we nog even het verschil tussen *schuiven* en *roteren* benadrukken. Bij schuiven wordt de inhoud van een register of een geheugenplaats 1 bit naar links of naar rechts geschoven. Er wordt een "0" ingeschoven. Het bit dat uitgeschoven wordt komt in de carry. Bij een rotatie komt het uitgeschoven bit ook in de carry. Het ingeschoven bit is nu echter de vorige waarde van de carry. Dit komt overeen met een 9-bits rotatie. Het zou vaak wenselijk zijn om een echte 8-bits rotatie te hebben, waarbij het bit dat uitgeschoven wordt aan de ene kant, aan de andere kant weer ingeschoven wordt. Hierin is in de meeste microprocessoren niet voorzien. Ook is het tenslotte bij het naar rechts schuiven van een woord handig om een ander type verschuiving te hebben, en wel de tekenverlenging

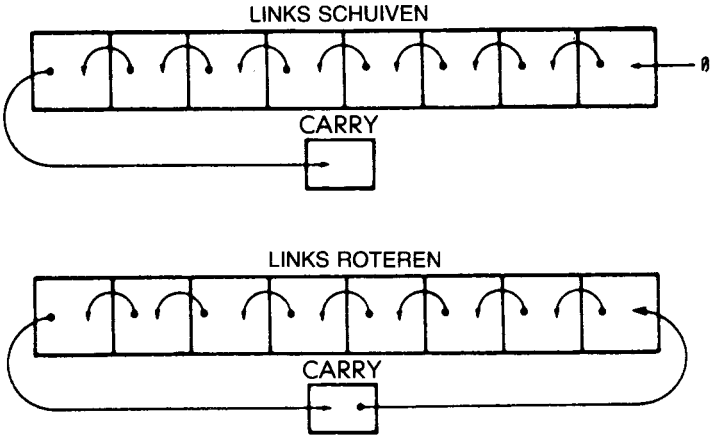


Fig. 4-1: Schuiven en roteren

of de "rekenkundige verschuiving naar rechts". Als er rekenkundige bewerkingen op 2-komplement getallen gepleegd worden, in het bijzonder bij het uitvoeren van drijvende-komma getallen, moet het links ingeschoven bit een "1" zijn (het tekenbit moet zovaak herhaald worden als door de opeenvolgende verschuivingen nodig is). Helaas kent de 6502 dit soort verschuiving niet. Andere microprocessoren kennen hem wel.

Test en spring

De test instrukties testen alle bits van het statusregister op "0" of "1", of een kombinatie hiervan. Het is daarom wenselijk om zoveel mogelijk vlaggen in dit register te hebben. Verder is het gemakkelijk om met een instruktie een kombinatie van bits te testen. Tenslotte is het wenselijk om *ieder bit in ieder register* te kunnen testen, en om de waarde van een register met ieder ander register te kunnen vergelijken (groter, kleiner, gelijk). De microprocessor testinstrukties zijn meestal beperkt tot het testen van enkele bits van het statusregister.

De spronginstrukties vallen meestal uiteen in drie kategorieen:

- de absolute sprong zelf, die een volledig 16-bits adres specificeert,
- de relatieve sprong (branch), die meestal beperkt is tot een 8-bits verplaatsingsveld,
- de call, die bij subroutines gebruikt wordt

Het is handig om twee- of zelfs drie-wegs vertakkingen te hebben, afhankelijk van bijvoorbeeld het resultaat van een vergelijking, dat groter, kleiner of gelijk kan zijn. Het is ook handig om instrukties te hebben, waarmee over een klein aantal instrukties vooruit of achteruit gesprongen kan worden. Tenslotte is er aan het einde van de meeste lussen vaak een vermindering of vermeerdering van een registerinhoud. De beschikbaarheid van een gekombineerde instruktie vermeerder/verminder plus test en spring is een aanzienlijk voordeel bij het uitvoeren van lussen. Deze is op de meeste microprocessoren niet beschikbaar. Er zijn slechts simpele sprongen, gekombineerd met eenvoudige testen beschikbaar. Dit is natuurlijk lastig bij het programmeren en vermindert de efficientie.

Invoer/uitvoer

Invoer/Uitvoer (*I*nput/*O*utput) instrukties zijn gespecialiseerde instrukties voor de behandeling van invoer/uitvoer komponenten. In de praktijk maken bijna alle microprocessoren gebruik van *memory-mapped I/O*. Dit betekent, dat de invoer/uitvoer komponenten net als de geheugenchips met de adresbus verbonden zijn, en ook als zodanig geadresseerd worden. Ze verschijnen aan de programmeur als geheugenplaatsen. Alle geheugenbewerkingen kunnen dan op de gewenste komponenten toegepast worden. Het voordeel is dat een grote varieteit van instrukties toegepast kan worden. Het nadeel is dat geheugenoperaties doorgaans drie bytes lang zijn en dus traag. Voor een efficiente verwerking van invoer/uitvoer in een dergelijke omgeving is het vaak wenselijk om een kort adresseringsmechanisme te hebben, zodat de I/O komponenten, waarvan de verwerkingssnelheid kritisch is, in pagina 0 geplaatst kunnen zijn. Als echter pagina 0 adressering beschikbaar is, wordt dit meestal gebruikt voor RAM (Random Access Memory) geheugen, en kan dus niet effektief voor invoer/uitvoer komponenten gebruikt worden.

Kontrole-instrukties

Kontrole-instrukties genereren synchronisatiesignalen en kunnen een programma laten wachten of onderbreken. Ze kunnen ook een programma afbreken of dienen als een gesimuleerde onderbreking (interrupt). (Interrupts worden besproken in hoofdstuk 6 bij de invoer/uitvoer technieken.)

INSTRUKTIES DIE OP DE 6502 BESCHIKBAAR ZIJN

Gegevensoverdracht instrukties

De 6502 heeft een volledige verzameling instrukties om gegevens over te dragen, behalve voor het laden van de stapelwijzer, die minder flexibel is. De inhoud van de accumulator kan verwisseld worden met een geheugenplaats met de instruktie LDA (load = laden) en STA (store = opbergen). Ditzelfde geldt voor de registers X en Y. Hiervoor zijn de instrukties respektievelijk LDX, LDY en STX, STY. Register S kan niet direkt geladen worden. Er is natuurlijk voorzien in inter-register overdrachten: de instrukties zijn: TAX (transfereer A naar X), TAY, TSX, TXA, TXS en TYA. Er is een kleine asymmetrie, omdat de inhoud van S wel met X verwisseld mag worden, maar niet met Y.

Er is geen 2-adres geheugen-naar-geheugen bewerking, zoals "tel de inhoud van LOC1 bij LOC2 op".

Stapelbewerkingen

Er zijn twee "push" en "pop" bewerkingen beschikbaar. Ze brengen de inhoud van de register A of het statusregister P over naar de top van de stapel, terwijl gelijktijdig de stapelwijzer aangepast wordt. Deze instrukties zijn PLA en PLP (Pull A en Pull P), waarmee de inhoud van de top van de stapel naar resp. A of P overgebracht wordt, en PHA en PHP (Push A en Push B), waarmee de omgekeerde bewerking verricht wordt.

Gegevensverwerking

Rekenkundig

De gebruikelijke (beperkte) verzameling van rekenkundige, logische en schuifinstrukties is aanwezig. Rekenkundige bewerkingen zijn: ADC, SBC. ADC is een optelling met carry. Er is geen optelling zonder carry. Dit is enigszins ongemakkelijk, omdat nu voor de optelling een CLC instruktie nodig is. Aftrekkken geschiedt met SBC. Er is een speciale decimale mode waarmee getallen die in BCD genoteerd staan meteen opgeteld en afgetrokken kunnen worden. In de meeste microprocessoren is slechts een van deze instrukties beschikbaar als een aparte instruktiekode. Door de aanwezigheid van deze decimale vlag verdubbelt het effektieve aantal beschikbare rekenkundige bewerkingen.

Vermeerderen/verminderen

Instrukties om te verminderen of te vermeerderen zijn beschikbaar voor het geheugen en de indexregisters X en Y, maar niet voor de accumulator. Dit zijn respectievelijk: INC en DEC, die op het geheugen werken, INX, INY en DEX, DEY voor de registers X en Y.

Logische bewerkingen

De logische bewerkingen zijn de klassieke: AND, ORA en EOR. We zullen nu de rol van ieder van deze instrukties duidelijk maken.

AND

Iedere logische bewerking wordt gekarakteriseerd door een waarheidstabel, waarin de logische waarde van de uitgang als funktie van de ingangen uitgedrukt wordt.
De waarheidstabel voor een AND is als volgt:

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

De AND bewerking wordt gekarakteriseerd door het feit, dat de uitgang alleen "1" is als de ingangen "1" zijn. Met andere woorden: als een van de ingangen "0" is, is de uitgang gegarandeerd "0". Dit kenmerk wordt gebruikt om bepaalde bitposities in een woord nul te maken. Dit noemen we "maskeren".

Stel bijvoorbeeld, dat we de vier meest rechtse bitposities in een woord nul willen maken. Dit kan met het volgende programma:

```
LDA   WOORD      WOORD BEVAT "10101010"
AND   #%11110000 "11110000" IS HET MASKER
```

Laten we aannemen, dat WOORD gelijk is aan "10101010". Het resultaat van dit programma is, dat in de accumulator de waarde "10100000" achterblijft. "%" wordt gebruikt om een binair getal aan te geven.

Oefening 4.1: Schrijf een programma van twee regels, dat bits 1 en 6 van WOORD nul maakt.

Oefening 4.2: Wat gebeurt er met een masker: MASKER = "11111111"?

ORA

Deze instruktie is de inclusive OR bewerking. Hij wordt gekarakteriseerd door de volgende waarheidstabel:

```
0 OR  0 = 0
0 OR  1 = 1
1 OR  0 = 1
1 OR  1 = 1
```

De logische OR wordt gekarakteriseerd door het feit, dat als een van de ingangen "1" is, de uitgang altijd "1" is. OR wordt dus gebruikt om een bit in een woord op "1" te zetten. Laten we de vier meest rechtse posities in WOORD op "1" zetten:

```
LDA   #WOORD
ORA   #%00001111
```

Laten we aannemen, dat WOORD "10101010" bevatte. De waarde van de accumulator wordt dan "10101111".

Oefening 4.3: Wat zou er gebeuren als we de instruktie ORA #%10101111 zouden gebruiken?

Oefening 4.4: Wat is het effekt van het ORen met "FF" hexadecimaal?

EOR

EOR staat voor "exclusive OR". Het verschil tussen inclusive OR en exclusive OR is, dat bij de EOR de uitgang "0" is, dan en alleen dan als slechts een van de ingangen "1" is. Als beide ingangen "1" zijn , zou de gewone OR een "1" als uitgang geven. De exclusive OR geeft een "0" resultaat. De waarheidstabel is:

```
0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0
```

De exclusive OR wordt voor vergelijkingen gebruikt. Zodra er een bit verschillend is, wordt de exclusive OR van twee woorden ongelijk aan nul. Verder wordt de exclusive OR bij de 6502 gebruikt om een woord te komplementeren, aangezien er geen specifieke komplement instruktie is. Dit kan gebeuren door de EOR te nemen met een woord

van allemaal enen. Het programma is dan als volgt:

```
LDA   #WOORD
EOR   #%11111111
```

Laten we aannemen, dat WOORD "10101010" bevatte. De uiteindelijke waarde van de accumulator wordt dan "01010101". We kunnen nagaan, dat dit het komplement is van de oorspronkelijke waarde.

Oefening 4.5: Wat is het effekt van EOR #$00?

Schuifbewerkingen

De standaard 6502 is uitgerust met een instruktie om naar links te schuiven, genaamd ASL (arithmetic shift left = rekenkundig schuiven naar links), en een instruktie om naar rechts te schuiven, genaamd LSR (logical shift right = logisch schuiven naar rechts). Ze worden hierna beschreven.

De 6502 kent echter maar een roteerinstruktie, en wel die naar links (ROL).

Waarschuwing: nieuwere versies van de 6502 hebben een extra roteerinstruktie. Kontroleer de gegevens van de fabrikant om dit na te gaan (ROR = roteer rechts).

Vergelijkingen

Registers X, Y en A kunnen met het geheugen vergeleken worden met de instrukties CPX, CPY en CMP.

Test en Spring

Omdat testen bijna uitsluitend met behulp van het vlaggenregister gebeurt, zullen we nu de vlaggen die de 6502 heeft onderzoeken. De inhoud van het vlaggenregister is getekend in inderstaande figuur 4-2.

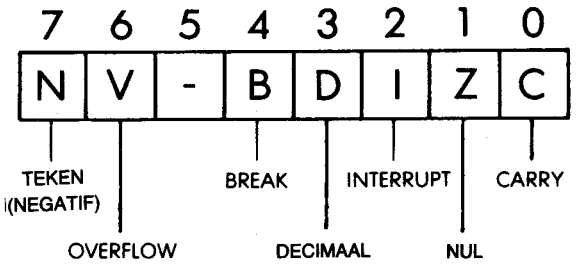


Fig. 4-2: Het vlaggenregister

We zullen nu de funktie van de vlaggen van links naar rechts bespreken.

Teken

Het meest linkse bit is het tekenbit of negatiefbit. Als N gelijk is aan "1", geeft het aan, dat de waarde van een resultaat negatief is in de 2-komplement notatie. In de praktijk is vlag N gelijk aan bit 7 van het resultaat. Het wordt gezet door alle instrukties voor gegevensverwerking en gegevensoverdracht.

De N vlag is in de meeste gevallen gelijk aan bit 7 van de accumulator. Het gevolg is, dat bit 7 het enige bit van de accumulator is dat eenvoudig met een enkele instruktie getest kan worden. Om een ander bit te testen van de accumulator, moet de inhoud ervan verschoven worden. In alle gevallen waarin men dus snel de inhoud van een woord wil testen, zal de voorkeur uitgaan naar bitpositie 7. Om deze reden zijn I/O-statusbits meestal verbonden met bit 7 van de databus. Om de status van een I/O komponent te kontroleren, hoeft alleen maar de inhoud van het externe statusregister in de accumulator gelezen te worden, waarna bit N getest kan worden.

Instrukties die N zetten zijn: ADC, AND, ASZ, BIT, CMP, CPY, CPX, DEC, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, TAX, TAY, TXS, TXA, TYA.

Overflow

De funktie van de overflow (overstroming) is reeds besproken in Hoofdstuk 3 bij het deel over de rekenkundige bewerkingen. Het wordt gebruikt om aan te geven, dat het resultaat van een optelling of aftrekking met 2-komplement getallen foutief kan zijn wegens een overflow van bit 6 naar bit 7, dus in het tekenbit. Als dit bit gezet is, moet een speciale korrektieroutine gebruikt worden. Als niet de 2-komplement notatie, maar de direkt binaire gebruikt wordt, is het overflowbit identiek aan een carry van bit 6 naar bit 7.

De BIT instruktie maakt een bijzonder gebruik van dit bit.
Het resultaat van deze instruktie is dat bit V gelijk gemaakt wordt aan bit 6 van de gegevens die getest worden.

De V vlag wordt beinvloed door ADC, BIT, CLV, PLP, RTI, SBC.

Break

Deze vlag wordt automatisch gezet door de processor als er door het BRK commando een interrupt veroorzaakt wordt. Hij maakt onderscheid tussen een geprogrammeerde break en een hardware interrupt. Hij wordt niet veranderd door de andere instrukties.

Decimaal

Het gebruik van deze vlag is al besproken in hoofdstuk 3 bij het deel over rekenkundige programma's. Als het D-bit "1" is, werkt de processor in *BCD* mode, en als het "0" is in de *binaire mode*. Deze vlag wordt beinvloed door vier instrukties: CLD, PLP, RTI, SED.

Interrupt

Dit interruptmasker kan expliciet door de programmeur gezet worden met de instrukties CLI of PLP, of door de microprocessor tijdens de reset of een interrupt.
Het gevolg is, dat volgende interrupts afgeschermd worden.
Instrukties die dit bit beinvloeden zijn: BRK, CLI, PLP, RTI, SEI.

Zero (nul)

Als de Z-vlag "1" is, geeft dit aan dat het resultaat van een overdracht of een bewerking nul is. Deze vlag wordt ook gezet door de vergelijkinstruktie. Er is geen specifieke instruktie waarmee het Z-bit op nul gezet kan worden. Hetzelfde resultaat kan echter eenvoudig verkregen worden. Het op nul zetten van het Z-bit kan bijvoorbeeld met de instruktie:

```
LDA  #0
```

Het Z-bit wordt beinvloed door de volgende instrukties: ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, RTS, ROL, ROR, RTS, SBC, TAX, TAY, TXA, TYA.

Carry

We hebben al gezien dat het carrybit voor twee doelen gebuikt

wordt. Het eerste doel is om een rekenkundige carry of borrow tijdens rekenkundige bewerkingen aan te geven. Het tweede doel is om het bit te bewaren dat tijdens schuif of roteerbewerkingen uit een register "valt". Deze twee funkties hoeven niet verward te worden en grotere computers kennen ze niet. Bij een microprocessor bespaart deze aanpak tijd, in het bijzonder bij het uitvoeren van een vermenigvuldiging of een deling. Het carrybit kan expliciet gezet en op nul gezet worden.

Instrukties die het carrybit beinvloeden zijn: ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP, ROL, ROR, RTI, SBC, SEC.

Test en sprong instrukties

In de 6502 kan ieder bit van het vlaggenregister afzonderlijk op "0" of "1" getest worden. Er zijn 6 bits, en dus zijn er twaalf verschillende spronginstrukties. Dit zijn:

- BMI (branch on minus = spring als negatief), BPL (branch on plus = spring als positief). Deze instrukties testen natuurlijk het N-bit.
- BCC (branch on carry clear = spring als carry clear is) en BCS (branch on carry set = spring als carry gezet is): ze testen bit C.
- BEQ (branch equal = spring als gelijk (aan nul)) en BNE (spring als resultaat ongelijk aan nul). Ze testen Z.
- BVS (branch on overflow set = spring als overflow gezet is) en BVC (branch on overflow clear = spring als overflow clear is). Ze testen V.

Deze instrukties testen en springen binnen dezelfde instruktie. Alle sprongen specificeren een relatieve verplaatsing vanaf de huidige instruktie. Omdat het verplaatsingsveld 8 bits breed is, kan hiermee een verplaatsing van -128 tot $+127$ gespecificeerd worden (2-komplement). De verplaatsing wordt opgeteld bij het adres van de instruktie die volgt op de spronginstruktie.

Omdat alle sprongen twee bytes lang zijn, resulteert dit in een effektieve verplaatsing van $-128 + 2 = -126$ tot $+ 127 + 2 = +129$.

Er zijn nog twee onvoorwaardelijke sprongen beschikbaar: JMP en JSR. JMP is een sprong naar een 16-bits adres. JSR is een subroutineaanroep. Hiermee wordt naar het nieuwe adres gesprongen en automatisch de programmateller op de stapel geplaatst. Omdat deze sprongen onvoorwaardelijk zijn, worden ze meestal voorafgegaan door een test-en-spring instruktie.

Er zijn twee terugkeerinstrukties (returns) beschikbaar: RTI, een terugkeer van een interrupt, die we zullen bespreken in het deel over interrupts, en RTS, een terugkeer van een subroutine, die het terugkeeradres van de stapel haalt (en vermeerdert).

Er zijn twee speciale instrukties om bits te testen en te vergelijken.

De BIT instruktie verricht een AND met een geheugenplaats en de accumulator. Een belangrijk aspekt is dat *de inhoud van de accumulator niet veranderd wordt.* Vlag N wordt gelijk gemaakt aan bit 7 van de geheugenplaats, terwijl de V-vlag gelijk gemaakt wordt aan bit 6 van de geheugenplaats die getest wordt. Tenslotte geeft bit Z het resultaat van de AND bewerking aan. Z wordt "1" als het resultaat "0" is. In de meeste gevallen zal een masker in de accumulator geladen worden, waarna achtereenvolgende geheugenplaatsen met de BIT instruktie getest worden. Als het masker maar een "1" bevat bijvoorbeeld, kan getest worden of een bepaald geheugenwoord op die plaats een "1" bevat. In de praktijk betekent dit dat een masker alleen gebruikt hoeft te worden voor de bitposities "0" tot "5". De lezer zal zich herinneren, dat de bitposities "6" en "7" automatisch opgeslagen worden in de V-vlag en de N-vlag. Ze hoeven niet gemaskeerd te worden.

De CMP instruktie vergelijkt de inhoud van de accumulator met die van een geheugenplaats door deze inhoud van de accumulator af te trekken. Het resultaat van de vergelijking wordt daarom aangegeven door bits Z en N. Hiermee kan getest worden op groter, kleiner of gelijk. De waarde van de accumulator wordt niet veranderd door de vergelijking. CPX en CPY vergelijken respektivelijk met X en met Y.

Invoer/uitvoer instrukties

De 6502 heeft geen speciale invoer/uitvoer instrukties.

Kontrole instrukties

Kontrole instrukties zijn gespecialiseerde instrukties om vlaggen te zetten of op nul te zetten. Dit zijn: CLC, CLD, CLI, CLV, waarmee respektievelijk bits C, D, I en V op nul gezet worden, en SEC, SED, SEI, waarmee respektievelijk bits C, D en I gezet worden.

De BRK insruktie is equivalent met een software interrupt en wordt beschreven in hoofdstuk 7 in het deel over interrupts.

De NOP instruktie is een instruktie die geen effekt heeft en die

meestal gebruikt wordt om de tijdsduur van een lus te verlengen. Tenslotte heeft de 6502 nog twee speciale pennen, waarmee een interruptmechanisme in werking gezet wordt (getriggerd wordt). Dit wordt verklaard in hoofdstuk 6 over invoer/uitvoer technieken. Het is een hardware kontrole faciliteit (IRQ en NMI pennen).

Laten we nu iedere instruktie in detail bekijken.

Om de diverse adresseermodes werkelijk goed te kunnen begrijpen wordt de lezer aangeraden om de volgende sectie eerst vlot door te nemen en er na het bestuderen van Hoofdstuk 5 over adresseertchnieken dieper op in te gaan.

DEEL 2 - DE INSTRUKTIES

AFKORTINGEN

A	Accumulator
M	Gespecificeerd geheugenadres (Memory)
P	Statusregister
S	Stapelwijzer
X	Indexregister
Y	Indexregister
DATA	Gespecificeerde gegevens
HEX	Hexadecimaal
PC	Programmateller (counter)
PCH	Programmateller hoog
PCL	Programmateller laag
STAPEL	Inhoud van de top van de stapel
∨	Logische of (or)
∧	Logische en (and)
∀	Exclusieve of (eor)
•	Veranderen
( )	Inhoud van
(M6)	Bit positie 6 op adres M.

ADC — Optellen met carry

Funktie: A ← (A) + DATA + C

Formaat:

Beschrijving:

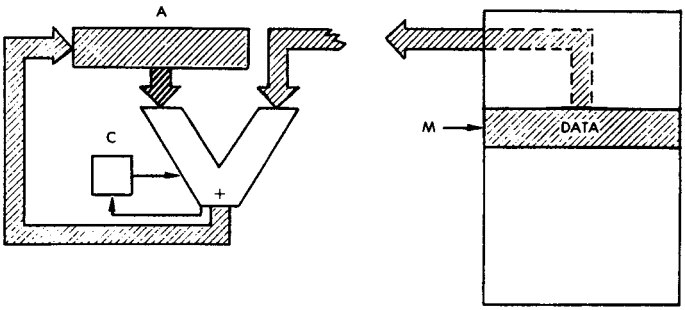
Tel de inhoud van het geheugenadres of de konstante bij de accumulator op, samen met het carrybit. Het resultaat komt in de accumulator.

Opmerkingen:

– ADC kan in binaire of decimale mode werken: de vlaggen moeten op de juiste waarde gezet zijn.

– Om zonder carry op te tellen, moet vlag C op nul gezet worden (CLC).

Datapad:



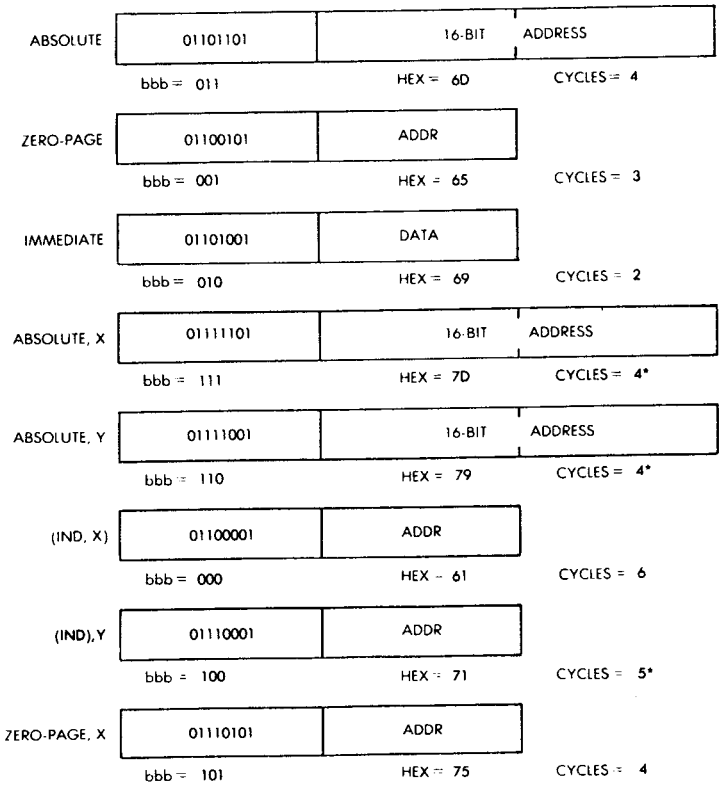
Adresseermodes: PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

	IMPLIED	ACCUMULATOR	ABSOLUTE	0.PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0.PAGE, X	0.PAGE, Y	RELATIVE	INDIRECT
HEX			6D	65	69	7D	79	61	71	75			
BYTES			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

Vlaggen:

N	V	B	D	I	Z	C
●	●				●	●

Instruktiekodes: * PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT



AND — Logische AND

Funktie: A ← (A) Λ DATA

Formaat:

001bbb01	ADDR/DATA	ADDR

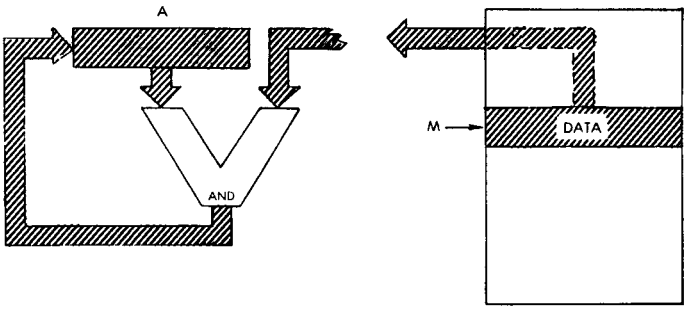
Beschrijving:

Verricht de logische AND van de accumulator en de gespecificeerde data. Het resultaat komt in de accumulator.

De waarheidstabel is:

A\M	0	1
0	0	0
1	0	1

Datapad:



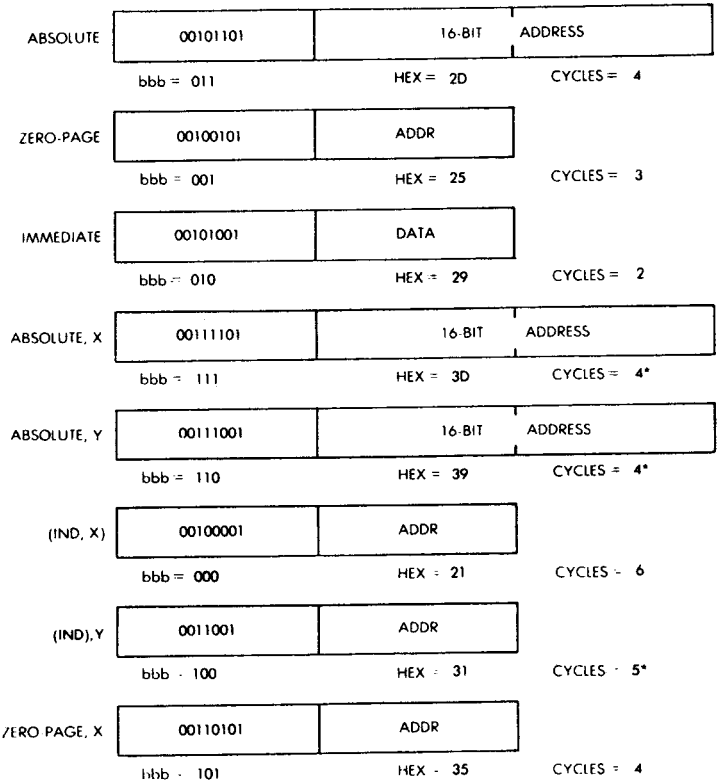
Adresseermodes: PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

	IMPLIED	ACCUMULATOR	ABSOLUTE	0.PAGE	IMMEDIATE	ABS. X	ABS. Y	(IND, X)	(IND), Y	0 PAGE, X	0 PAGE, Y	RELATIVE	INDIRECT
HEX			2D	25	29	3D	39	21	31	35			
BYTES			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

Vlaggen:

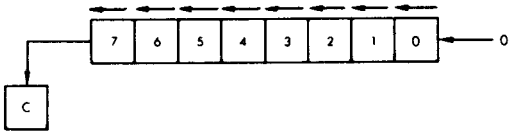
N	V	B	D	I	Z	C
●					●	

Instruktiekodes: PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT



ASL — Rekenkundig naar links schuiven

Funktie:

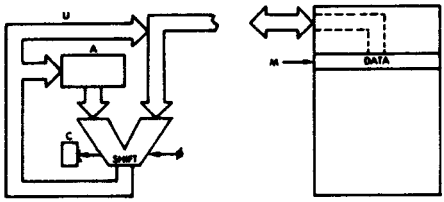


Formaat:

Beschrijving:

Verschuif de inhoud van de accumulator of de geheugenplaats een positie naar links. Rechts wordt een 0 ingeschoven. Bit 7 komt in de carry. Het resultaat komt weer terug in de bron, d.w.z. in de accumulator of het geheugen.

Datapad:



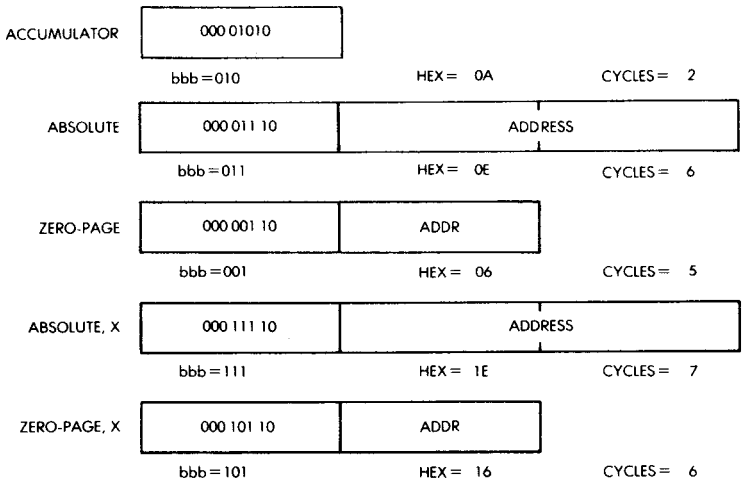
Adresseermodes:

	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIVE	INDIRECT
HEX		0A	0E	06		1E				16			
BYTES		1	3	2		3				2			
CYCLES		2	6	5		7				6			
bbb		010	011	001		111				101			

Vlaggen:

N	V	B	D	I	Z	C
●					●	●

Instruktiecodes:



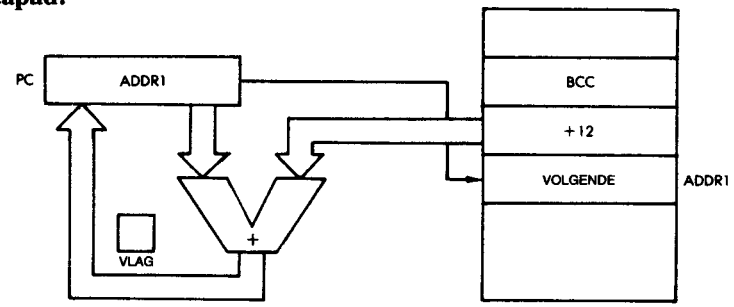
BCC — Spring als carry clear

Funktie:
Ga naar gespecificeerd adres als C=0

Formaat:

Beschrijving:
Test de carryvlag. Spring als C=0 naar het huidige adres, vermeerderd met de verplaatsing (max. +127 of −128). Als C=1 geen aktie ondernemen. De verplaatsing wordt opgeteld bij het adres van de instruktie die volgt op de BCC. Dit heeft een effektieve verplaatsing van +129 tot −126 tot gevolg.

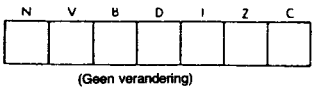
Datapad:



Adresseermode:
Alleen relatief

HEX = 90, bytes = 2, cycles = 2 + 1 als sprong lukt
+ 2 als naar andere pagina gesprongen wordt

Vlaggen:



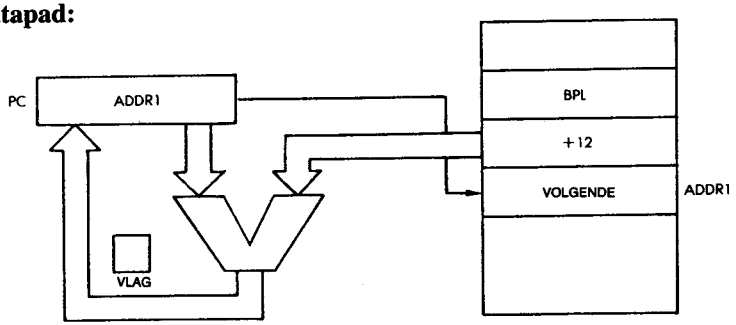
BCS — Spring als carry gezet

Funktie:
Ga naar gespecificeerd adres als C=1

Formaat:

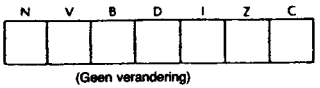
Beschrijving:
Test de carryvlag. Spring als C=1 naar het huidige adres, vermeerderd met de verplaatsing (max. +127 of −128). Als C=0 geen aktie ondernemen. De verplaatsing wordt opgeteld bij het adres van de instruktie die volgt op de BCC. Dit heeft een effektieve verplaatsing van +129 tot −126 tot gevolg.

Datapad:



Adresseermode:
Alleen relatief
HEX = 80, bytes = 2, cycles = 2 + 1 als sprong lukt
+ 2 als naar andere pagina gesprongen wordt

Vlaggen:

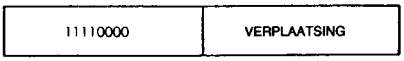


BEQ — Spring als gelijk aan nul

Funktie:

Ga naar gespecificeerd adres als Z=1 (resultaat = 0).

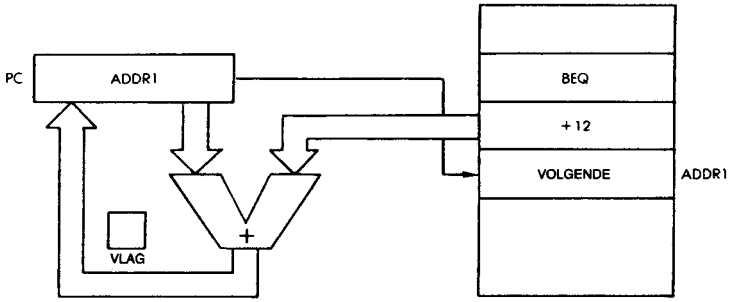
Formaat:



Beschrijving:

Test de vlag. Spring als Z=1 naar het huidige adres, vermeerderd met de verplaatsing (max. +127 of −128). Als Z=0 geen aktie ondernemen. De verplaatsing wordt opgeteld bij het adres van de instruktie die volgt op de BEQ. Dit heeft een effektieve verplaatsing van +129 tot −126 tot gevolg.

Datapad:

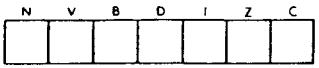


Adresseermode:

Alleen relatief

HEX = FO, bytes = 2, cycles = 2 + 1 als sprong lukt
+ 2 als naar andere pagina gesprongen wordt

Vlaggen:



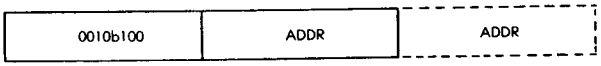
(Geen verandering)

BIT — Vergelijk geheugenbits met accumulator

Funktie:

$Z \leftarrow \overline{(A) \wedge (M)}$, $N \leftarrow (M^7)$, $V \leftarrow (M^6)$

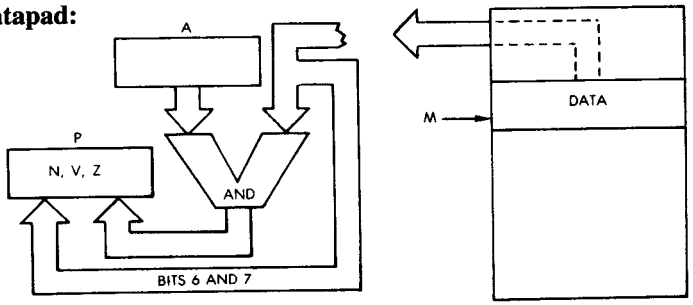
Formaat:



Beschrijving:

De logische AND van A en M wordt bepaald, maar niet opgeborgen. Het resultaat van de vergelijking wordt aangegeven door Z. Z=1 als de vergelijking klopt; anders 0. Verder worden bits 6 en 7 van het geheugen overgedragen in V en N van het statusregister.

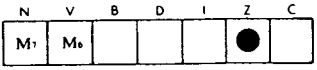
Datapad:



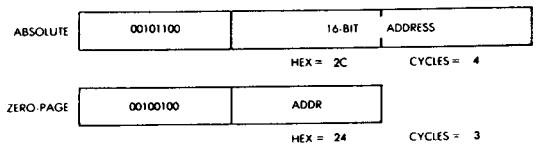
Adresseermodes:

	IMPLIED	ACCUMULATOR	ABSOLUTE	0.PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0.PAGE, X	0.PAGE, Y	RELATIVE	INDIRECT
HEX			2C	24									
BYTES			3	2									
CYCLES			4	3									
bbb			011	001									

Vlaggen:



Instruktiekodes:

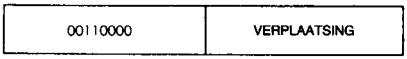


BMI — Spring als negatief

Funktie:

Ga naar gespecificeerde adres als N=1 (resultaat <0).

Formaat:

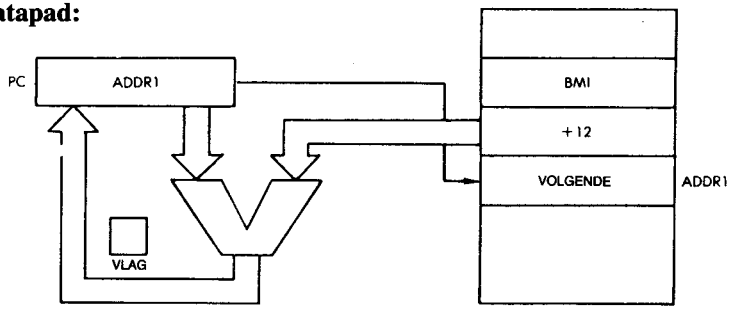


Beschrijving:

Test de N vlag (teken). Als N=1, spring naar het huidige adres, vermeerderd met de verplaatsing (max. + 127 of −128). Als N=0 geen aktie ondernemen.

De verplaatsing wordt opgeteld bij het adres van de eerste instruktie na de BMI. Dit heeft een effektieve verplaatsing van +129 tot −126 tot gevolg.

Datapad:

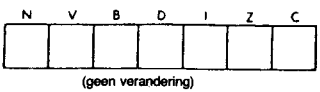


Adresseermode:

Alleen relatief:

HEX = 30, bytes = 2, cycles = 2 + 1 als sprong lukt
+ 2 als naar andere pagina gesprongen wordt

Vlaggen:



BNE — Spring als ongelijk aan nul

Funktie:
Ga naar gespecificeerde adres als Z=0 (resultaat ≠0).

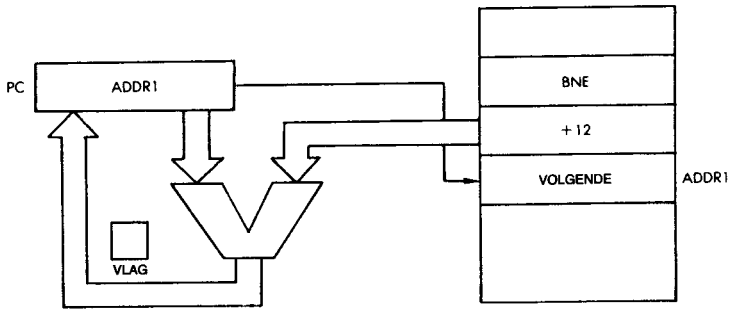
Formaat:

11010000	VERPLAATSING

Beschrijving:
Test het resultaat (Z vlag). Als het resultaat niet gelijk nul is (Z=0), spring naar het huidige adres, vermeerderd met de verplaatsing (max. +127 of −128). Als Z=1 geen aktie ondernemen.

De verplaatsing wordt opgeteld bij het adres van de eerste instruktie na de BNE. Dit heeft een effektieve verplaatsing van +129 tot −126 tot gevolg.

Datapad:



Adresseermode:
Alleen relatief:
HEX = DO, bytes = 2, cyles = 2 + 1 als sprong lukt
+ 2 als naar andere pagina gesprongen wordt

Vlaggen:

N	V	B	D	I	Z	C

(geen verandering)

BPL — Spring als positief

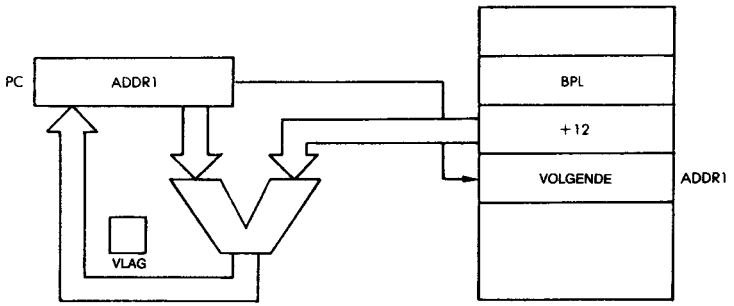
Funktie:
Ga naar gespecificeerde adres als N=0 (resultaat ≥).

Formaat:

Beschrijving:
Test de N vlag (teken). Als N=0 (resultaat positief), spring naar het huidige adres, vermeerderd met de verplaatsing (max. +127 of −128). Als N=1 geen aktie ondernemen.

De verplaatsing wordt opgeteld bij het adres van de eerste instruktie na de BPL. Dit heeft een effektieve verplaatsing van +129 tot −126 tot gevolg.

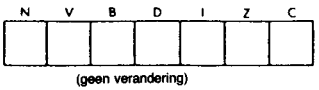
Datapad:



Adresseermode:
Alleen relatief:

HEX = 10, bytes = 2, cycles = 2 + 1 als sprong lukt
+ 2 als naar andere pagina gesprongen wordt

Vlaggen:



BRK — Break

Funktie:
STAPEL (PC) + 2, STAPEL (P), PC ←(FFFE,FFFF))

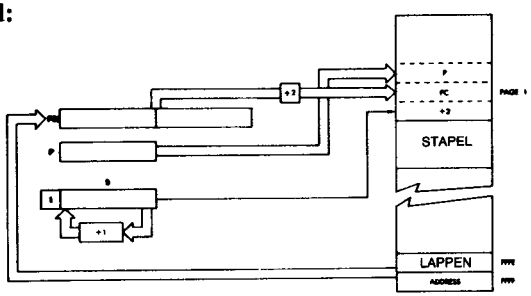
Formaat:

00000000

Beschrijving:
Werkt als een interrupt: eerst wordt de programmateller op de stapel gezet, daarna het statusregister P. De inhoud van geheugenplaatsen FFFE en FFFF worden dan geplaatst in respektievelijk PCL en PCH. De waarde van P in de stapel heeft de B vlag op 1 staan, om een BRK van een IRQ te onderscheiden.

Belangrijk: in tegenstelling met een interrupt, wordt PC = 2 opgeborgen. Dit hoeft niet de volgende instruktie te zijn en een korrektie kan nodig zijn. Dit is wegens het normale gebruik van BRK om bestaande programma's op te lappen, waarbij BRK een 2-byte instruktie vervangt.

Datapad:



Adresseermode:
Alleen impliciet:
HEX = 00 , byte = 1, cycles = 7

Vlaggen:

N	V	B	D	I	Z	C
		★		1		

* B is gezet voor P op de stapel geplaatst wordt

BVC — Spring als overflow clear

Funktie:

Ga naar gespecificeerde adres als V=0.

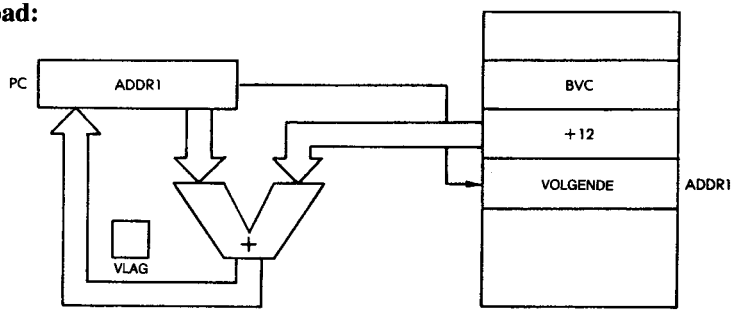
Formaat:

Beschrijving:

Test de overflowvlag (V). Als er geen overflow is (V=0) spring naar het huidige adres, vermeerderd met de verplaatsing (max. +127 of −128). Als V=1 geen aktie ondernemen.

De verplaatsing wordt opgeteld bij het adres van de eerste instruktie na de BVC. Dit heeft een effektieve verplaatsing van +129 tot −126 tot gevolg.

Datapad:

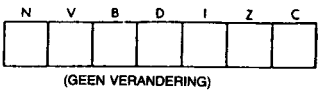


Adresseermode:

Alleen relatief:

HEX = 50, bytes = 2, cycles = 2 + 1 als sprong lukt
+ 2 als naar andere pagina gesprongen wordt

Vlaggen:



BVS — Spring als overflow gezet

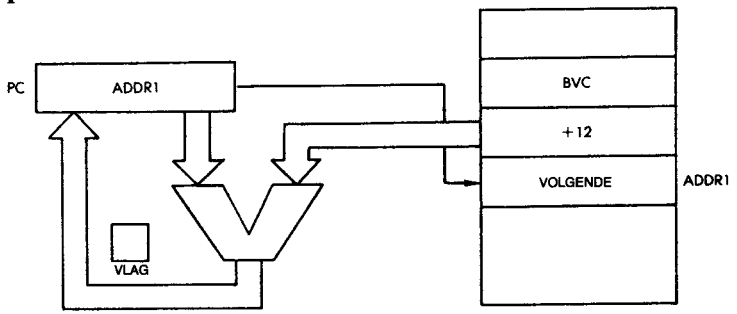
Funktie:
Ga naar gespecificeerde adres als V=1.

Formaat:

Beschrijving:
Test de overflowvlag (V). Als er een overflow is (V=1), spring naar het huidige adres, vermeerderd met de verplaatsing (max. +127 of −128). Als V=0 geen aktie ondernemen.

De verplaatsing wordt opgeteld bij het adres van de eerste instruktie na de BVS. Dit heeft een effektieve verplaatsing van +129 tot −126 tot gevolg.

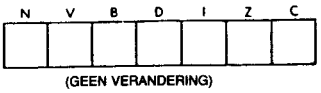
Datapad:



Adresseermode:
Alleen relatief:

HEX = 70, bytes = 2, cycles = 2 + 1 als sprong lukt
+ 2 als naar andere pagina gesprongen wordt

Vlaggen:

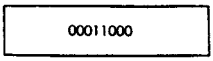


CLC — Clear carry

Funktie:

C ← ∅

Formaat:



Beschrijving:

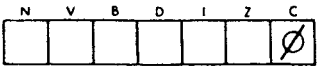
Het carrybit wordt op nul gezet. Dit is vaak nodig voor een ADC.

Adresseermode:

Alleen impliciet:

HEX = 18, byte = 1, cycles= 2

Vlaggen:



CLD — Decimaal vlag op nul zetten

Funktie:
D ← Ø

Formaat:

11011000

Beschrijving:
De decimaal vlag wordt op nul gezet, waarmee de binaire mode ingesteld wordt voor ADC en SBC.

Adresseermode:
Alleen impliciet:
HEX = D8, byte = 1, cycles= 2

Vlaggen:

N	V	B	D	I	Z	C
			Ø			

CLI — Interrupt masker op nul zetten

Funktie:
I ← ∅

Formaat:

01011000

Beschrijving:
Het interrupt maskerbit wordt op 0 gezet. Nu kunnen interrupts doorkomen. Een routine die interrupts afhandelt, moet altijd het I bit op nul zetten, omdat anders interrupts verloren gaan.

Adresseermode:
Alleen impliciet:
HEX = 58, byte= 1, cycles= 2

Vlaggen:

N	V	B	D	I	Z	C
				∅		

CLV — Overflowvlag op nul zetten

Funktie:
$I \leftarrow \emptyset$

Formaat:

01011000

Beschrijving:
De overflowvlag wordt op nul gezet.

Adresseermode:
Alleen impliciet:
HEX = B8, byte = 1, cycles = 2

Vlaggen:

N	V	B	D	I	Z	C
				$\emptyset$		

CMP — Vergelijk met accumulator

Funktie:
(A) − DATA → NZC:

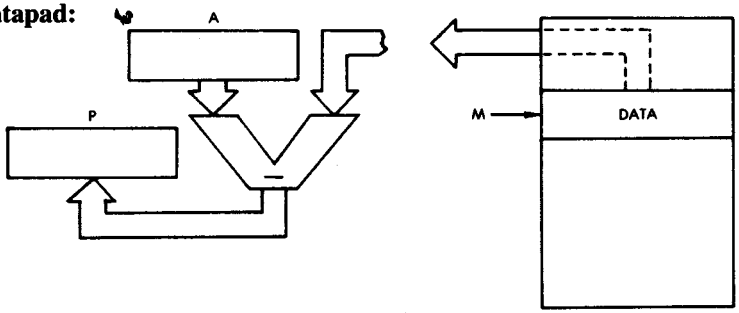
+ (A>DATA)	=	− (A<DATA)
−01	011	−00

Formaat:

110bbb01	ADDR/DATA	ADDR

Beschrijving:
De gespecificeerde inhoud wordt afgetrokken van A. Het resultaat wordt niet opgeborgen, maar de NZC vlaggen worden beinvloed, afhankelijk van of het resultaat positief, nul of negatief is. De waarde van de accumulator wordt niet veranderd. CMP wordt doorgaans gevolgd door een sprong: BCC detekteert A < data, BEQ detekteert A=data, BCS detekteert A ⩾ data, en BEQ gevolgd door BCS detekteert A > data.

Datapad:



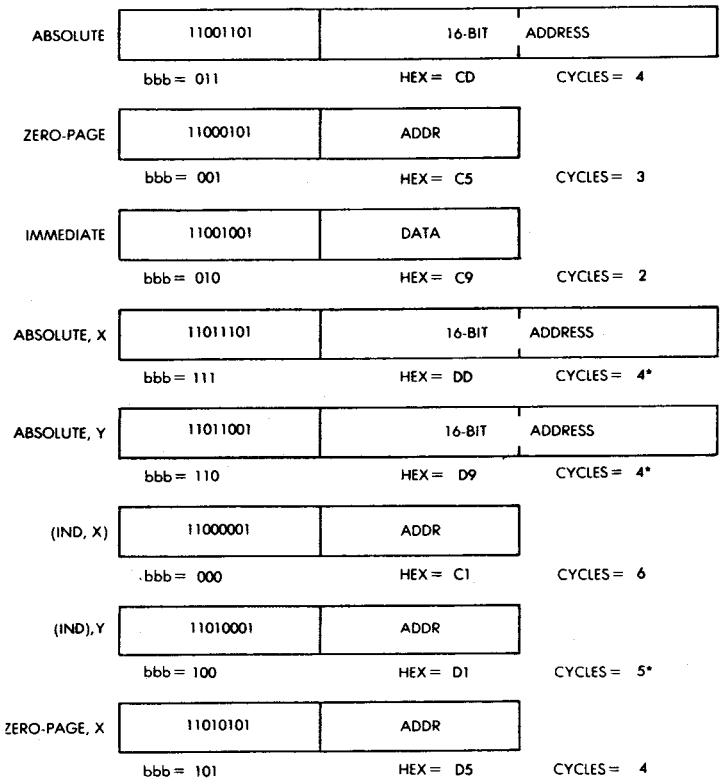
Adresseermodes: PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

	IMPLIED	ACCUMULATOR	ABSOLUTE	0.PAGE	IMMEDIATE	ABS. X	ABS. Y	(IND. X)	(IND). Y	0.PAGE. X	0.PAGE. Y	RELATIVE	INDIRECT
HEX			CD	C5	C9	DD	D9	C1	D1	D5			
BYTES			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

Vlaggen:

N	V	B	D	I	Z	C
●					●	●

Instruktiekodes:



PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

CPX

Vergelijk met register X

Funktie:
X – DATA →NZC:

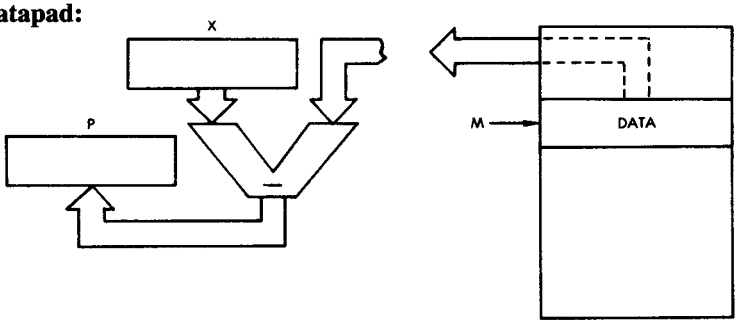
+(X > DATA)	=	–(X <DATA)
–01	011	–00

Formaat:

Beschrijving:

De gespecificeerde inhoud wordt afgetrokken van X. Het resultaat wordt niet opgeborgen, maar de NZC vlaggen worden beinvloed, afhankelijk van of het resultaat positief, nul of negatief is. De waarde van register X wordt niet veranderd. CPX wordt doorgaans gevolgd door een sprong: BCC detekteert X < data, BEQ detekteert X=data en BEQ gevolgd door BCS detekteert X > data. BCS detekteert X ⩾ data.

Datapad:



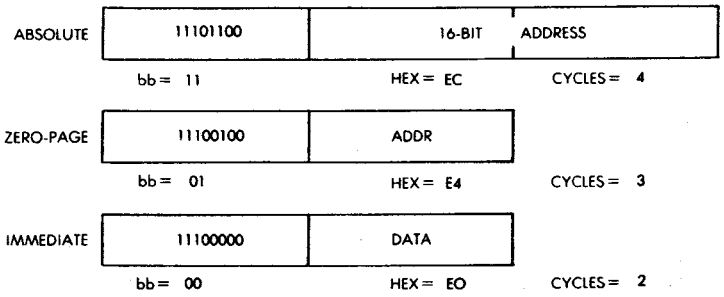
Adresseermodes:

	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS., X	ABS., Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIVE	INDIRECT
HEX			EC	E4	E0								
BYTES			3	2	2								
CYCLES			4	3	2								
bb			11	01	00								

Vlaggen:

N	V	B	D	I	Z	C
●					●	●

Instruktiekodes:



CPY

Vergelijk met register Y

Funktie:
(Y) – DATA →NZC:

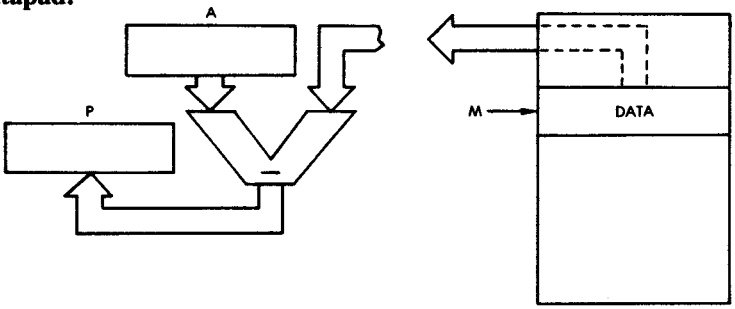
+(Y > DATA)	=	–(Y < DATA)
–01	011	–00

Formaat:

Beschrijving:

De gespecificeerde inhoud wordt afgetrokken van Y. Het resultaat wordt niet opgeborgen, maar de NZC vlaggen worden beinvloed, afhankelijk van of het resultaat positief, nul of negatief is. De waarde van register Y wordt niet veranderd. CPY wordt doorgaans gevolgd door een sprong: BCC detekteert Y < data, BEQ detekteert Y=data en BEQ gevolgd door BCS detekteert Y > data.

Datapad:



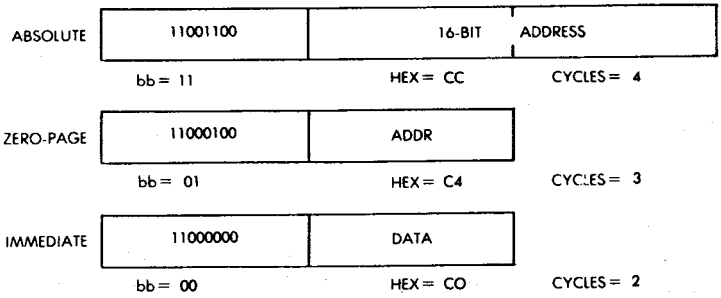
Adresseermodes:

	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIVE	INDIRECT
HEX			CC	C4	C0								
BYTES			3	2	2								
CYCLES			4	3	2								
bb			11	01	00								

Vlaggen:

N	V	B	D	I	Z	C
●					●	●

Instruktiekodes:

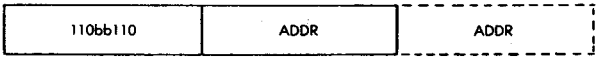


DEC — Verminder

Funktie:

M ← (M) – 1

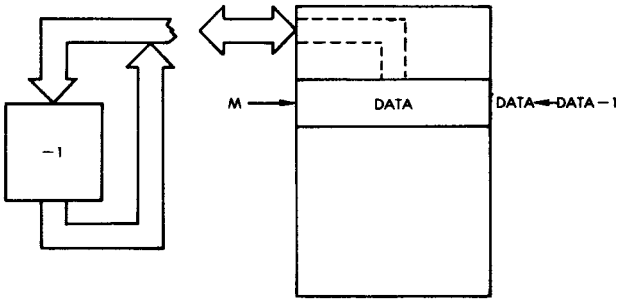
Formaat:



Beschrijving:

De inhoud van de gespecificeerde geheugenplaats wordt met 1 verminderd. Het resultaat wordt weer opgeborgen op het gespecificeerde adres.

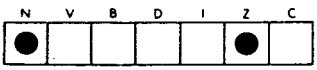
Datapad:



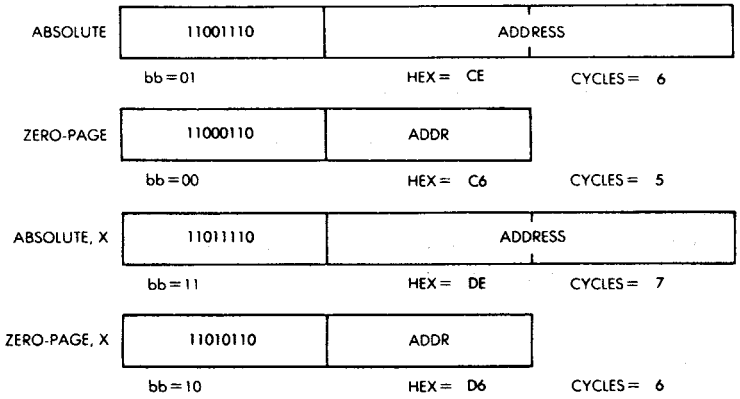
Adresseermodes:

	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIVE	INDIRECT
HEX			CE	C6		DE				D6			
BYTES			3	2		3				2			
CYCLES			6	5		7				6			
bb			01	00		11				10			

Vlaggen:



Instruktiekodes:



DEX — Verminder X

Funktie:

X ← (X) − 1

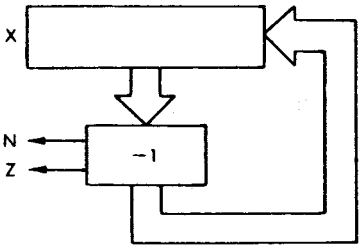
Formaat:

11001010

Beschrijving:

De inhoud van X wordt met 1 verminderd. Hierdoor kan X als teller gebruikt worden.

Datapad:

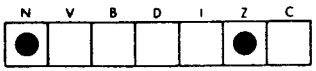


Adresseermode:

Alleen impliciet:

HEX = CA, byte = 1, cycles = 2

Vlaggen:



DEY — Verminder Y

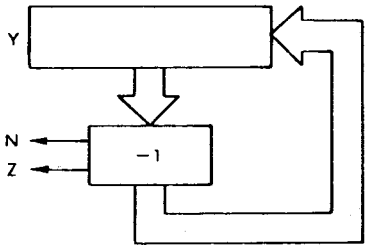
Funktie:
Y ← (Y) − 1

Formaat:

10001000

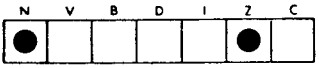
Beschrijving:
De inhoud van Y wordt met 1 verminderd. Hierdoor kan Y als teller gebruikt worden.

Datapad:



Adresseermode:
Alleen impliciet:
HEX = 88, byte = 1, cycles = 2

Vlaggen:



EOR — Exclusive OR met accumulator

Funktie:

A ← (A) ∀ DATA

Formaat:

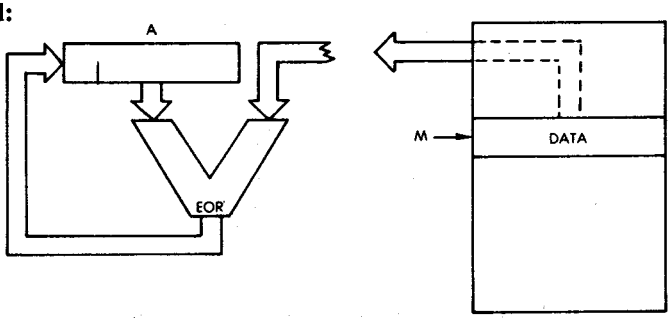
Beschrijving:

De exclusive OR van de accumulator met de gespecificeerde data komt in de accumulator.
De waarheidstabel is:

	0	1
0	0	1
1	1	0

Opmerking: EOR met '−1' kan gebruikt worden om te komplementeren.

Datapad:



Adresseermodes:

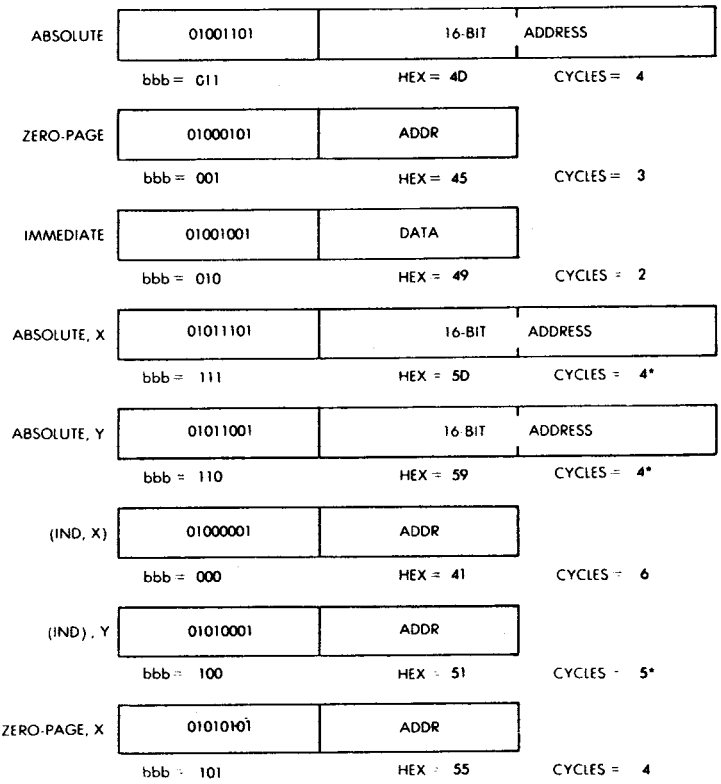
	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIVE	INDIRECT
HEX			4D	45	49	5D	59	41	51	55			
BYTES			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

Vlaggen:

N	V	B	D	I	Z	C
●					●	

Instruktiekodes:



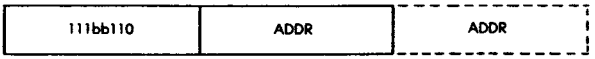
PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

INC — Vermeerder geheugen

Funktie:

M ← (M) +1

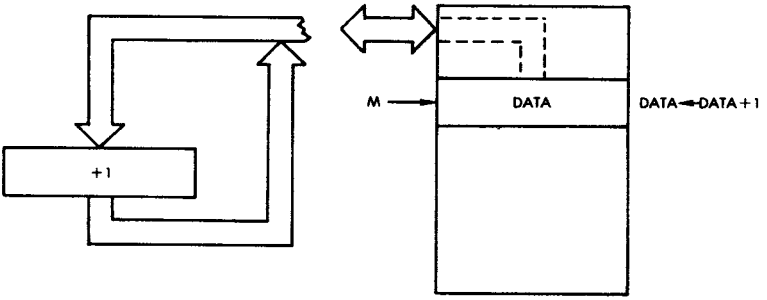
Formaat:



Beschrijving:

De inhoud van de gespecificeerde geheugenplaats wordt met 1 vermeerderd en vervolgens teruggeplaatst.

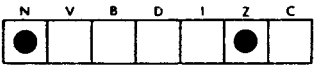
Datapad:



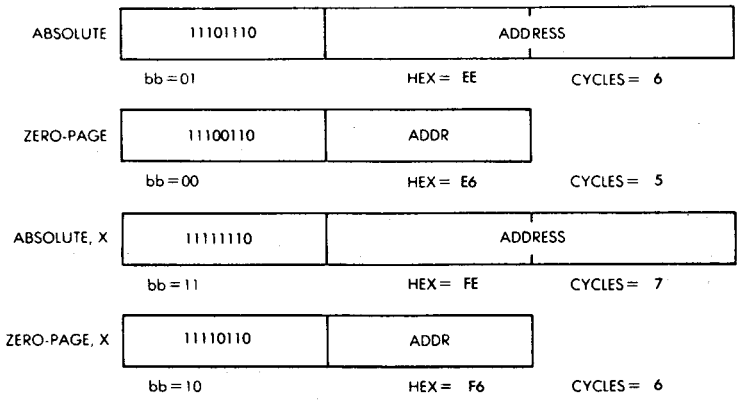
Adresseermodes:

	IMPLIED	ACCUMULATOR	ABSOLUTE	D-PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND, Y)	D-PAGE, X	D-PAGE, Y	RELATIVE	INDIRECT
HEX			EE	E6		FE				F6			
BYTES			3	2		3				2			
CYCLES			6	5		7				6			
bb			01	00		11				10			

Vlaggen:



Instruktiekodes:



INX — Vermeerder X

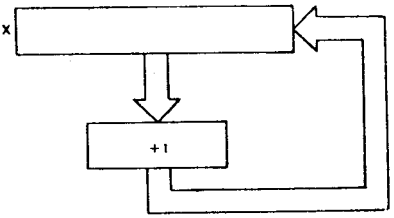
Funktie:
X ← (X) +1

Formaat:

11101000

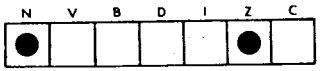
Beschrijving:
De inhoud van X wordt met 1 vermeerderd. Hierdoor kan X als teller gebruikt worden.

Datapad:



Adresseermode:
Alleen impliciet:
HEX = E8, byte = 1, cycles = 2

Vlaggen:



INY — Vermeerder Y

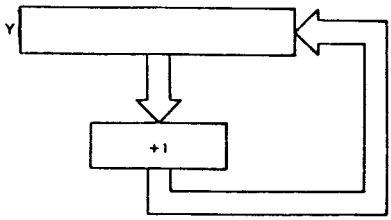
Funktie:
Y ← (Y) + 1

Formaat:

11001000

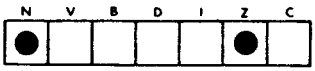
Beschrijving:
De inhoud van Y wordt met 1 vermeerderd. Hierdoor kan Y als teller gebruikt worden.

Datapad:



Adresseermode:
Alleen impliciet:
HEX = C8, byte = 1, cycles = 2

Vlaggen:



JMP — Spring naar adres

Funktie:

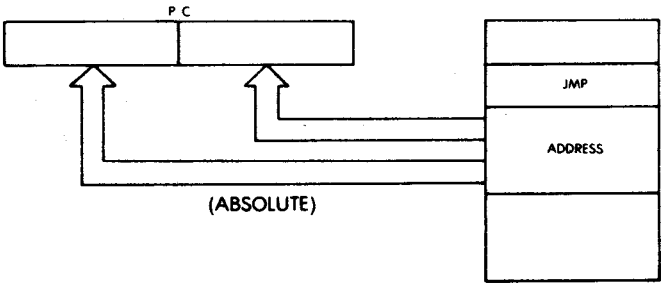
PC ← ADDRESS

Formaat:

Beschrijving:

Er wordt een nieuw adres in de programmateller geladen, waardoor een sprong in de programmavolgorde veroorzaakt wordt. De specifikatie van het adres kan absoluut of indirekt zijn.

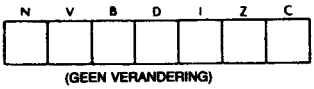
Datapad:



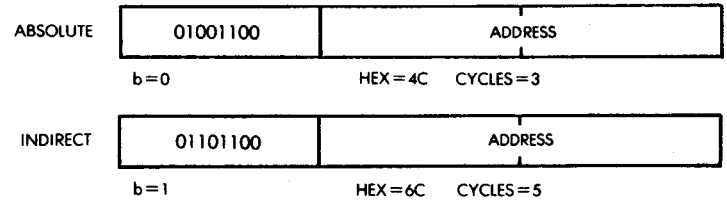
Adresseermodes:

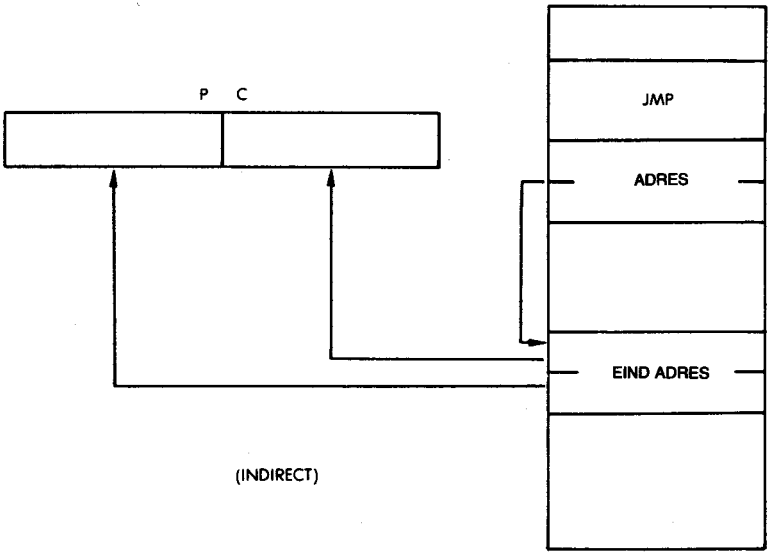
	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIVE	INDIRECT
HEX			4C										6C
BYTES			3										3
CYCLES			3										5
b			0										1

Vlaggen:



Instruktiekodes:





JSR — Spring naar subroutine

Funktie:

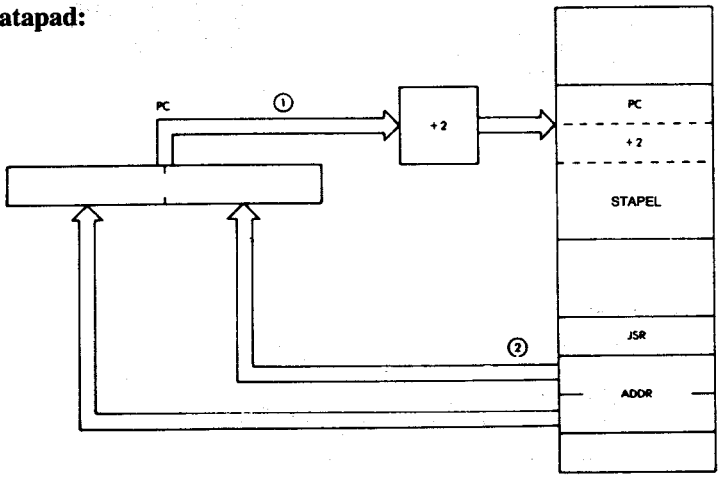
STACK← (PC) +2
PC← ADDRESS

Formaat:

Beschrijving:

De inhoud van de programmateller +2 wordt op de stapel geplaatst. (Dit is het adres van de instruktie die op de JSR volgt.) Het adres van de subroutine wordt dan in PC geladen. Dit wordt ook wel een subroutine'aanroep' genoemd.

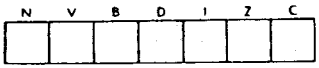
Datapad:



Adresseermode:

Alleen absoluut:
HEX = 20, bytes = 3, cycles = 6

Vlaggen:

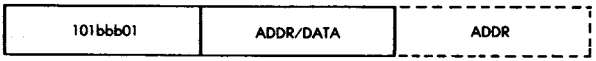


(GEEN VERANDERING)

LDA — Laad accumulator

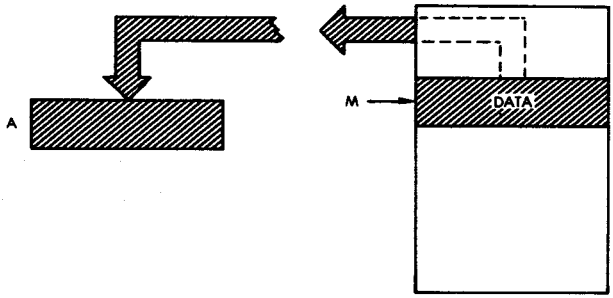
Funktie:
A ← DATA

Formaat:



Beschrijving:
De accumulator wordt met nieuwe data geladen.

Datapad:

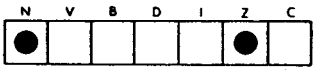


Adresseermodes:

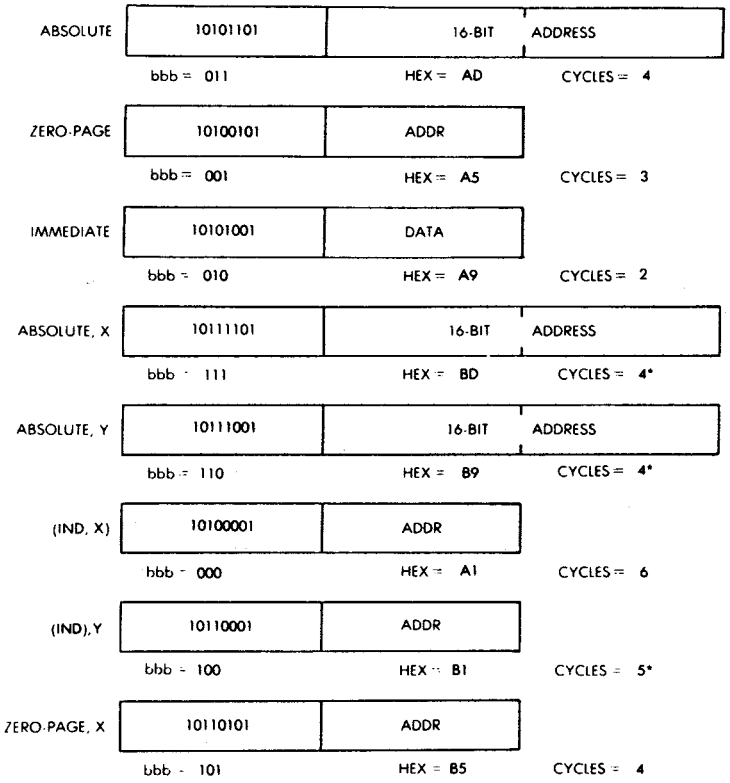
	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIVE	INDIRECT
HEX			AD	A5	A9	BD	B9	A1	B1	B5			
BYTES			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

Vlaggen:



Instruktiekodes:



PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

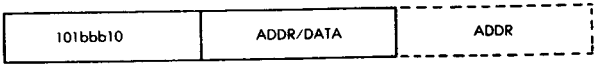
LDX

Laad register X

Funktie:

X ← DATA

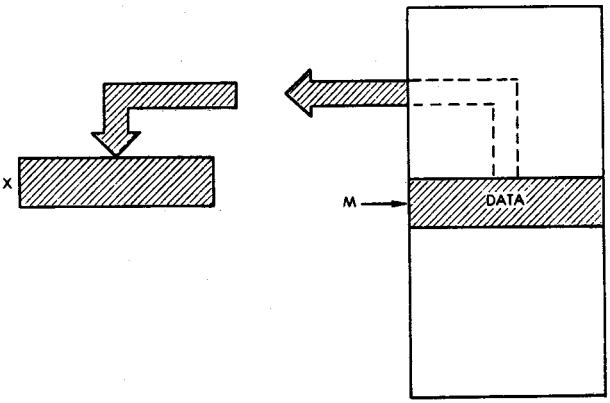
Formaat:



Beschrijving:

Indexregister X wordt geladen met gegevens van het gespecificeerde adres.

Datapad:



Adresseermodes:

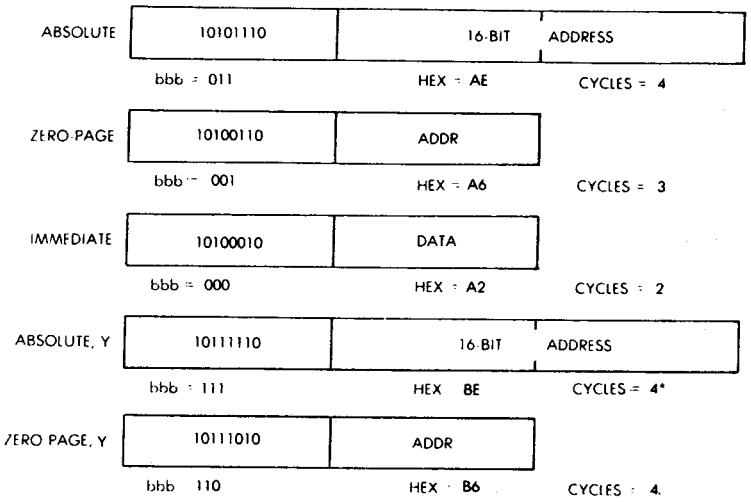
	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS. X	ABS. Y	(IND. X)	(IND). Y	0-PAGE. X	0-PAGE. Y	RELATIVE	INDIRECT
HEX			AE	A6	A2		BE				B6		
BYTES			3	2	2		3				2		
CYCLES			4	3	2		4*				4		
bbb			011	001	000		111				110		

PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

Vlaggen:

N	V	B	D	I	Z	C
●					●	

Instruktiekodes:

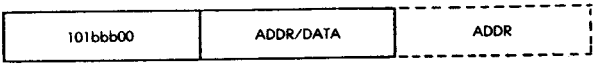


PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

LDY — Laad register Y

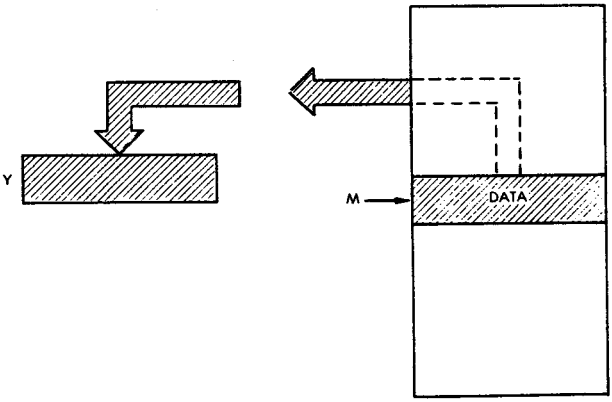
Funktie:
Y← DATA

Formaat:



Beschrijving:
Indexregister Y wordt geladen met gegevens van het gespecificeerde adres.

Datapad:

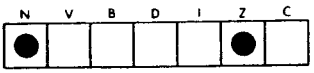


Adresseermodes:

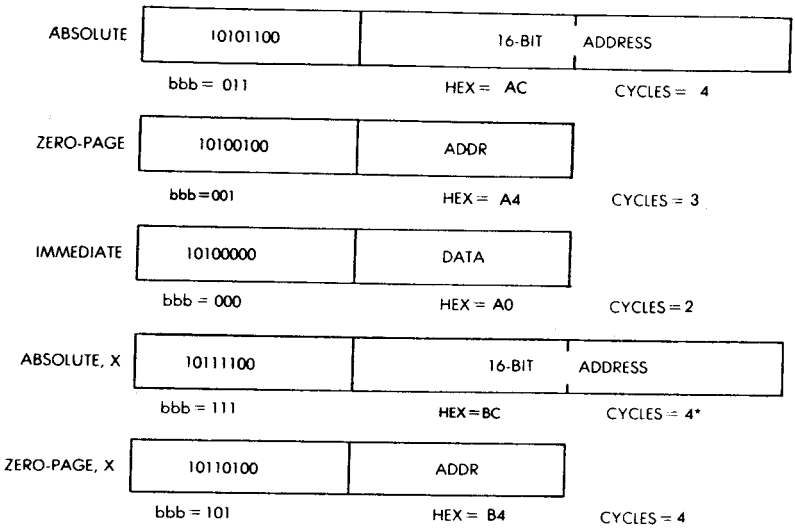
	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIVE	INDIRECT
HEX			AC	A4	A0	BC				B4			
BYTES			3	2	2	3				4			
CYCLES			4	3	2	4*				4			
bbb			011	001	000	111				101			

PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

Vlaggen:



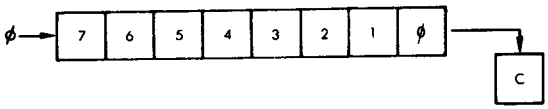
Instruktiekodes:



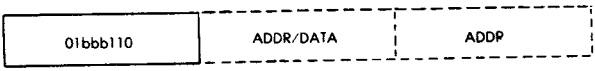
PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

LSR — Logisch schuiven naar rechts

Funktie:



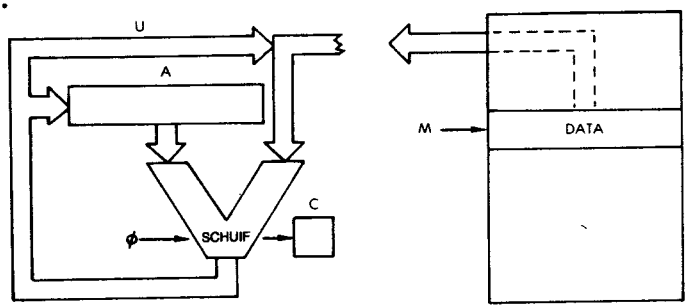
Formaat:



Beschrijving:

Schuif de gespecificeerde inhoud 1 positie naar rechts (accumulator of geheugen). In bit 7 wordt een 0 geschoven. Bit 0 komt in de carry. De verschoven gegevens worden in de bron teruggeplaatst, dus in de accumulator of het geheugen.

Datapad:



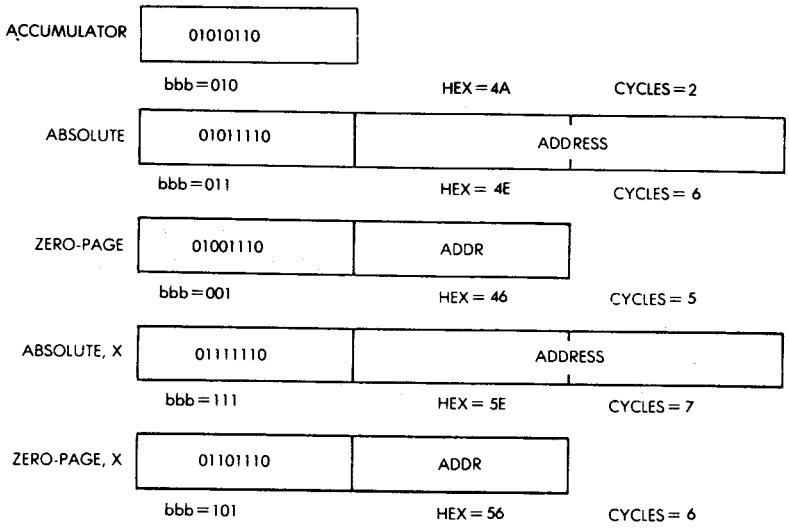
Adresseermodes:

	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS. X	ABS. Y	(IND. X)	(IND). Y	0-PAGE. X	0-PAGE. Y	RELATIVE	INDIRECT
HEX		4A	4E	46		5E				56			
BYTES		1	3	2		3				2			
CYCLES		2	6	5		7				6			
bbb		010	011	001		111				101			

Vlaggen:

N	V	B	D	I	Z	C
∅					●	●

Instruktiekodes:



NOP — Geen operatie

Funktie:
Geen

Formaat:

11101010

Beschrijving:
Doet niets gedurende 2 cycli. Kan gebruikt worden om vertragingslussen te timen of om lege plaatsen in een programma te vullen.

Adresseermode:
Alleen impliciet:
HEX = EA, byte = 1, cycles = 2

Vlaggen:

N	V	B	D	I	Z	C

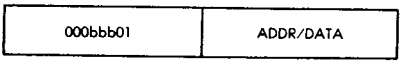
(GEEN VERANDERING)

ORA — Inclusive OR met accumulator

Funktie:

A ← (A) V DATA

Formaat:



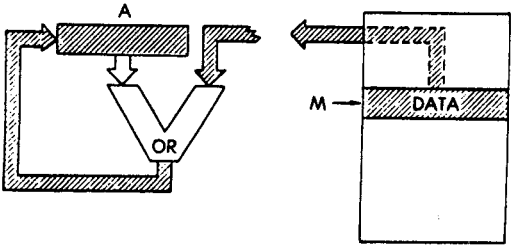
Beschrijving:

De logische (inclusive) OR van de accumulator met de gespecificeerde data komt in A. Kan gebruikt worden om een '1' te forceren op geselekteerde bitposities.

Waarheidstabel:

	0	1
0	0	1
1	1	1

Datapad:



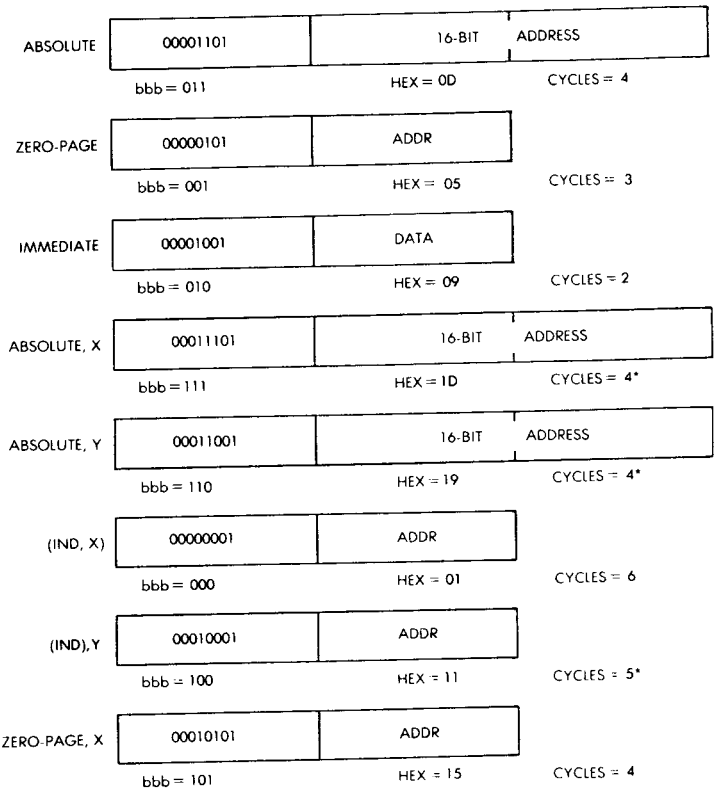
Adresseermodes: PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIVE	INDIRECT
HEX			0D	05	09	1D	19	01	11	15			
BYTES			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

Vlaggen:

N	V	B	D	I	Z	C
●					●	

Instruktiekodes:



PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

PHA

Push A

Funktie:

STAPEL ← (A)
S ← (S) − 1

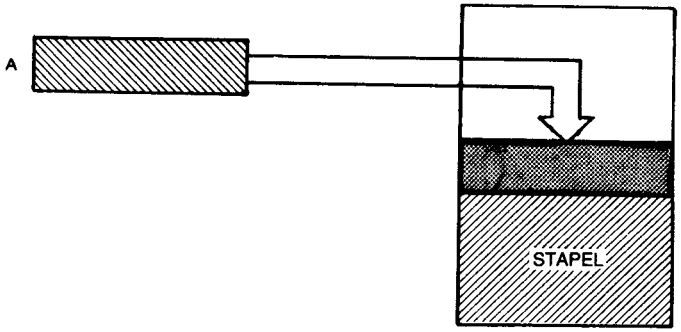
Formaat:

01001000

Beschrijving:

De inhoud van de accumulator wordt op de stapel gezet. De stapelwijzer wordt bijgewerkt. A blijft onveranderd.

Datapad:



Adresseermode:

Alleen impliciet:
HEX = 48, byte = 1, cycles = 3

Vlaggen:

N	V	B	D	I	Z	C

(GEEN VERANDERING)

PHP — Push processor status

Funktie:

STAPEL ← (P)
S ← (S) − 1

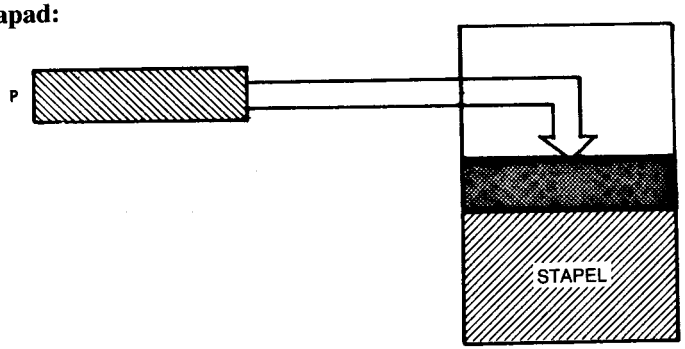
Formaat:

00001000

Beschrijving:

De inhoud van het statusregister P wordt op de stapel gezet. De stapelwijzer wordt bijgewerkt. A blijft onveranderd.

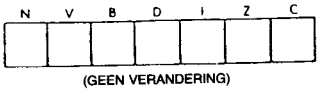
Datapad:



Adresseermode:

Alleen impliciet:
Hex = 08, byte = 1, cycles= 3

Vlaggen:



PLA

Pull accumulator

Funktie:

A ← (STAPEL)
S ← (S) +1

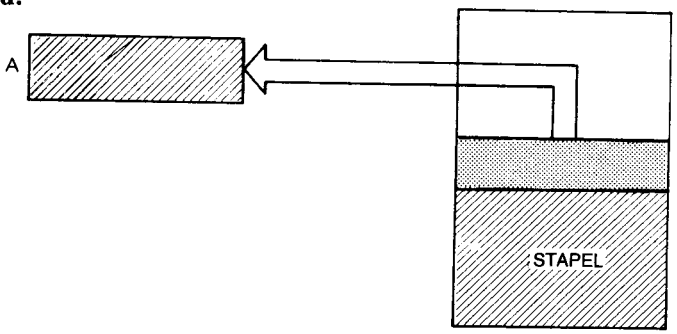
Formaat:

01101000

Beschrijving:

De inhoud van de top van de stapel komt in de accumulator. De stapelwijzer wordt verminderd.

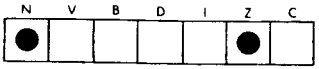
Datapad:



Adresseermode:

Alleen impliciet:
HEX = 68, byte = 1, cycles = 4

Vlaggen:



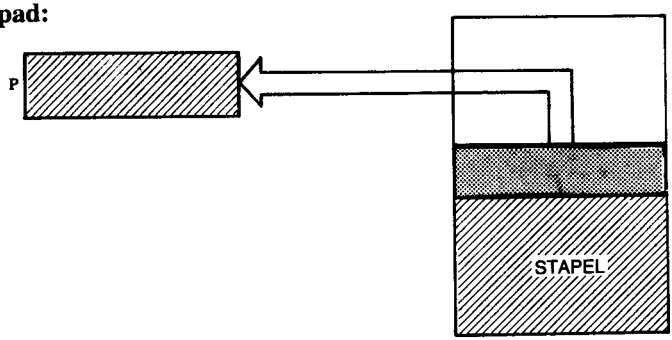
PLP — Pull processorstatus van stapel

Funktie:
P ← (STAPEL)
S ← (S) +1

Formaat: 00101000

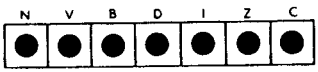
Beschrijving:
Het bovenste woord van de stapel wordt overgedragen naar het statusregister P. De stapelwijzer wordt vermeerderd.

Datapad:



Adresseermode:
Alleen impliciet:
HEX = 28, byte = 1, cycles = 4

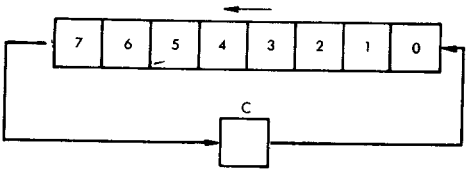
Vlaggen:



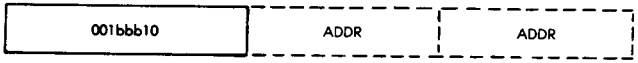
ROL

Roteer een bit naar links

Funktie:



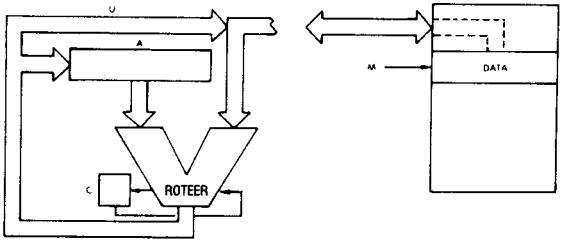
Formaat:



Beschrijving:

De inhoud van het gespecificeerde adres (accumulator of geheugen) wordt een positie naar links geroteerd. De carry komt in bit 0. Bit 7 bepaalt de nieuwe waarde van de carry. Dit is een 9-bits rotatie.

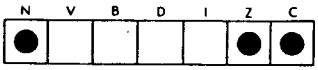
Datapad:



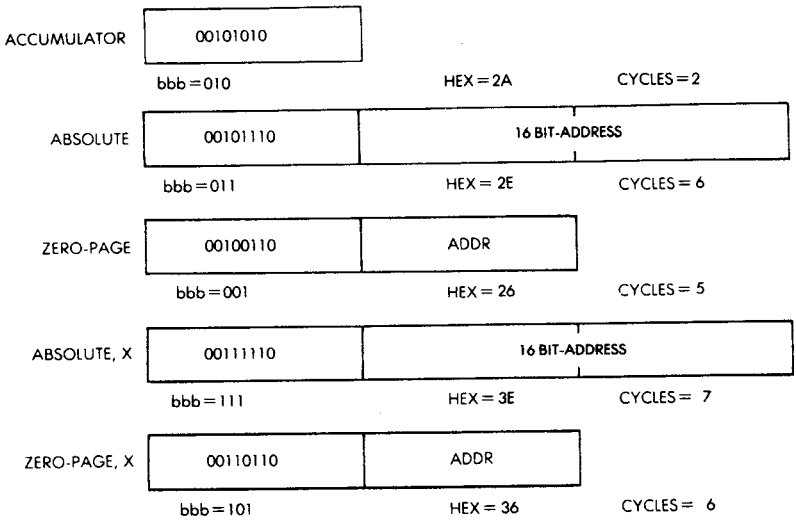
Adresseermodes:

	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIVE	INDIRECT
HEX		2A	2E	26		3E				36			
BYTES		1	3	2		3				2			
CYCLES		2	6	5		7				6			
bbb		010	011	001		111				101			

Vlaggen:



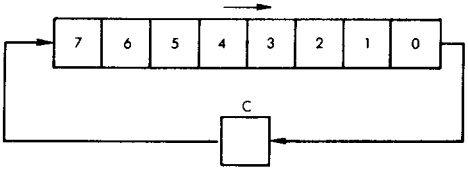
Instruktiekodes:



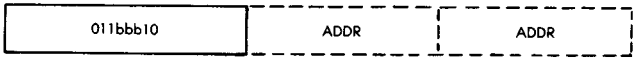
ROR — Roteer een bit naar rechts

Waarschuwing: het kan zijn, dat deze instruktie niet beschikbaar is op oudere 6502's. Ook is het mogelijk dat hij wel beschikbaar is, maar niet gedokumenteerd is.

Funktie:



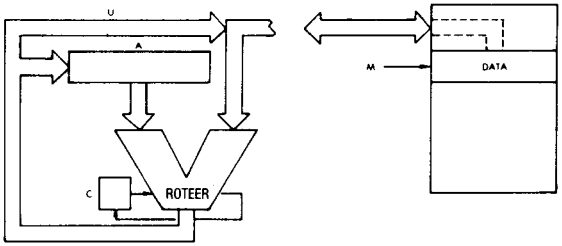
Formaat:



Beschrijving:

De inhoud van het gespecificeerde adres (accumulator of geheugen) wordt een positie naar rechts geroteerd. De carry komt in bit 7. Bit 0 bepaalt de nieuwe waarde van de carry. Dit is een 9-bits rotatie.

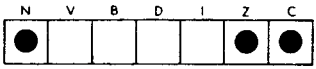
Datapad:



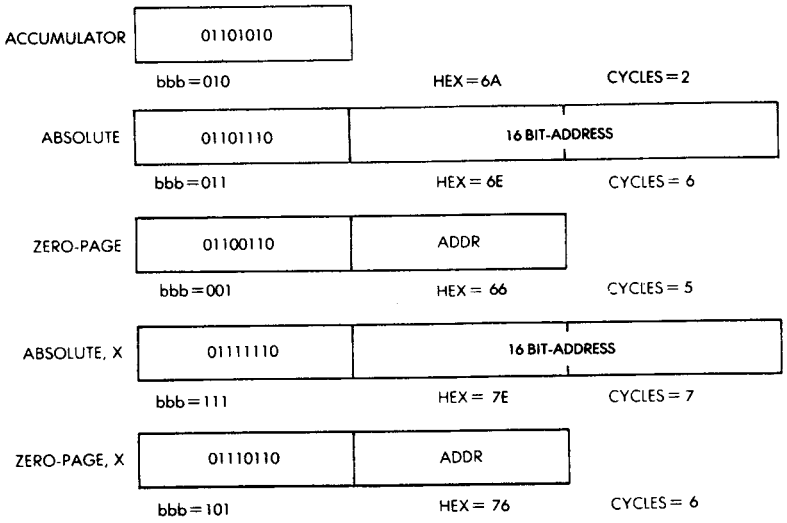
Adresseermodes:

	IMPLIED	ACCUMULATOR	ABSOLUTE	0 PAGE	IMMEDIATE	ABS. X	ABS. Y	(IND. X)	(IND). Y	0 PAGE. X	0 PAGE. Y	RELATIVE	INDIRECT
HEX		6A	6E	66		7E				76			
BYTES		1	3	2		3				2			
CYCLES		2	6	5		7				6			
bbb		010	011	001		111				101			

Vlaggen:



Instruktiekodes:



RTI

Terugkeer van interrupt

Funktie:

P ← (STAPEL)
S ← (S) +1
PCL ← (STAPEL)
S ← (S) +1
PCH ← (STAPEL)
S ← (S) +1

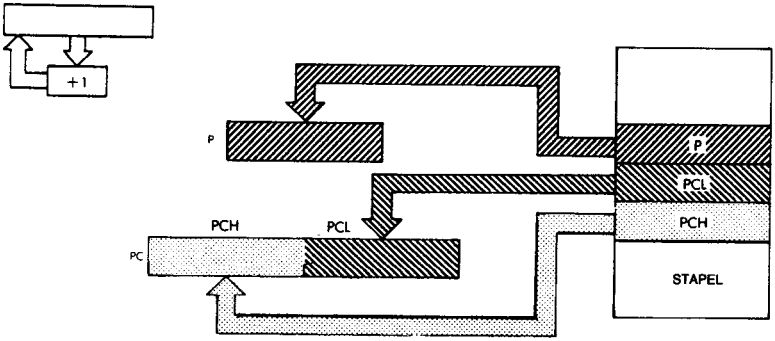
Formaat:

01000000

Beschrijving:

Herstel het statusregister P en de programmateller PC, die op de stapel opgeborgen waren. Werk de stapelwijzer bij.

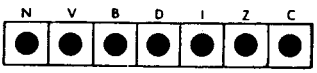
Datapad:



Adresseermode:

Alleen impliciet:
HEX = 40, byte = 1, cycles = 6

Vlaggen:



RTS — Terugkeer van subroutine

Funktie:

PCL ← (STAPEL)
S ← (S)+1
PCH ← (STAPEL)
S ← (S)+1
PC ← (PC + 1)

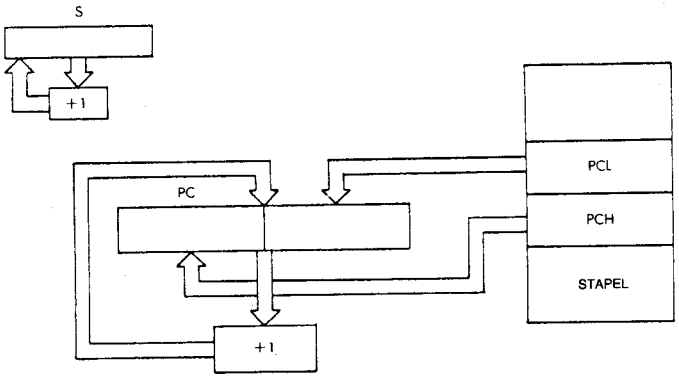
Formaat:

01100000

Beschrijving:

Herstel de programmateller vanuit de stapel en vermeerder hem met 1. Werk de stapelwijzer bij.

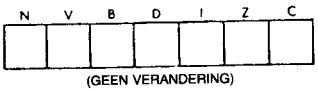
Datapad:



Adresseermode:

Alleen impliciet:
HEX = 60, byte = 1, cycles = 6

Vlaggen:



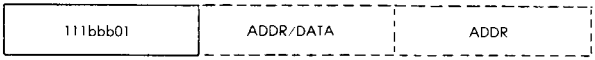
SBC

Trek af met carry

Funktie:

$A \leftarrow (A) - DATA - \bar{C}$ ($\bar{C}$ is borrow)

Formaat:

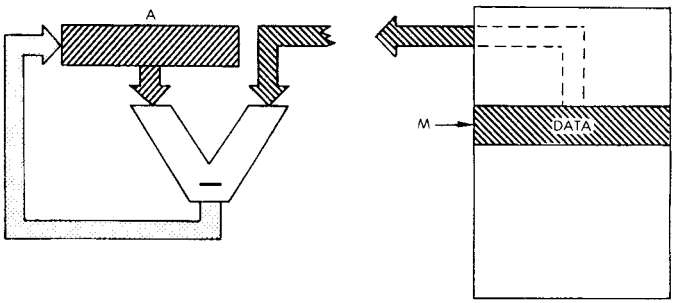


Beschrijving:

Trek de data van het gespecificeerde adres af van de accumulator, met borrow. Het resultaat komt in A. Opmerking: SEC wordt gebruikt voor een aftrekking zonder borrow.

SBC kan in binaire of decimale mode gebruikt worden, afhankelijk van de stand van bit D van het statusregister.

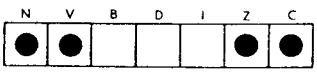
Datapad:



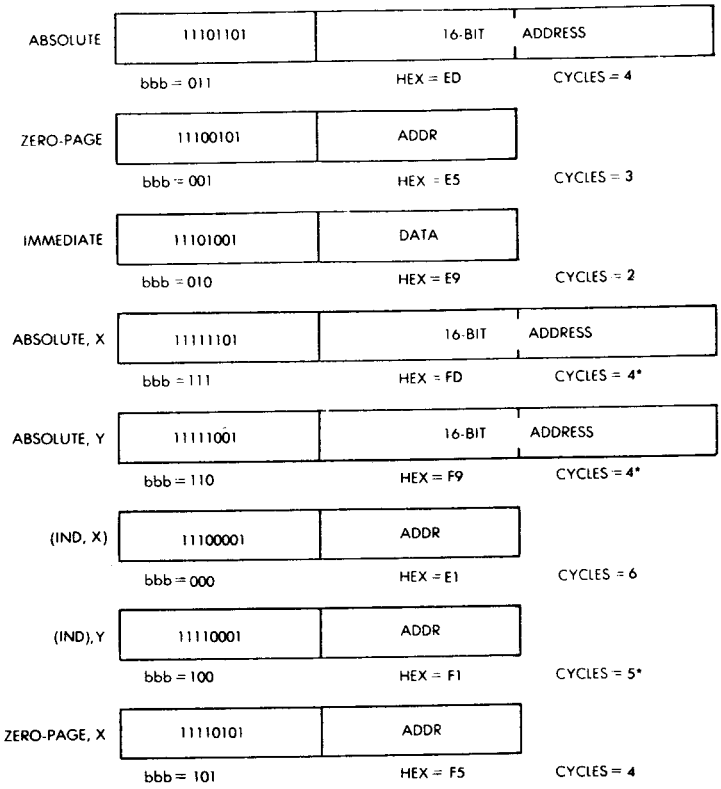
Adresseermodes: PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

	IMPLIED	ACCUMULATOR	ABSOLUTE	0-PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0-PAGE, X	0-PAGE, Y	RELATIVE	INDIRECT
HEX			ED	E5	E9	FD	F9	E1	F1	F5			
BYTES			3	2	2	3	3	2	2	2			
CYCLES			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

Vlaggen:



Instruktiekodes:



PLUS 1 CYCLUS ALS PAGINAGRENS OVERSCHREDEN WORDT

SEC

Zet carry

Funktie:
C ← 1

Formaat:

00111000

Beschrijving:
Het carrybit wordt op 1 gezet. Dit wordt gebruikt voor een SBC om af te trekken zonder carry.

Adresseermode:
Alleen impliciet:
HEX = 38, byte = 1, cycles= 2

Vlaggen:

N	V	B	D	I	Z	C
						1

SED — Zet decimale mode

Funktie:
D ← 1

Formaat:

11111000

Beschrijving:
Het decimaalbit van het statusregister wordt op 1 gezet. Als dit 0 is, is de mode binair. Als het 1 is, is de mode decimaal voor ADC en SBC.

Adresseermode:
Alleen impliciet:
HEX = F8, byte = 1, cycles = 2

Vlaggen:

N	V	B	D	I	Z	C
			1			

SEI — Zet interrupt disable

Funktie:
I ← 1

Formaat:

01111000

Beschrijving:
Het interruptmasker wordt op 1 gezet. Wordt gebruikt tijdens een interrupt of een systeemreset.

Adresseermode:
Alleen impliciet:
HEX = 78, byte = 1, cycles = 2

Vlaggen:

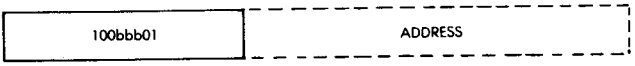
N	V	B	D	I	Z	C
				1		

STA — Berg accumulator op in geheugen

Funktie:

M ← (A)

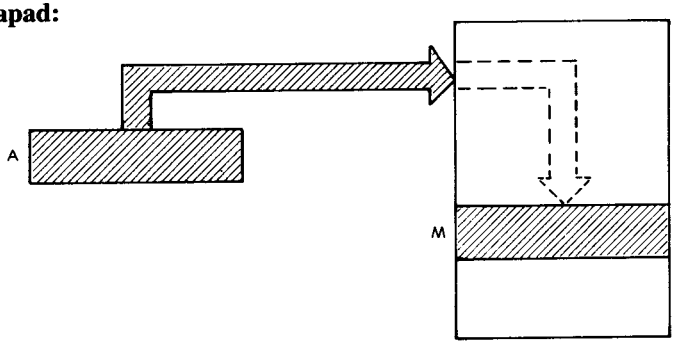
Formaat:



Beschrijving:

De inhoud van A wordt gekopieerd in de gespecificeerde geheugenplaats. De inhoud van A blijft onveranderd.

Datapad:



Adresseermodes:

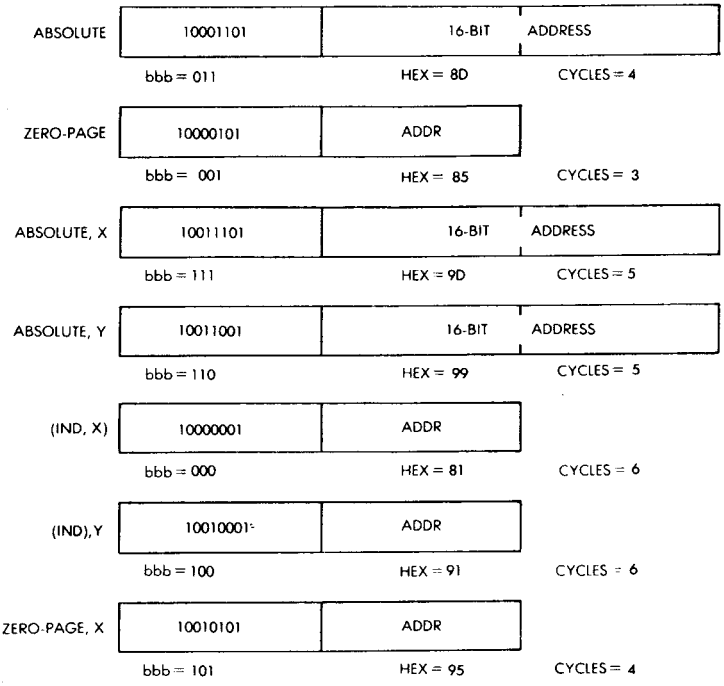
	IMPLIED	ACCUMULATOR	ABSOLUTE	0.PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0.PAGE, X	0.PAGE, Y	RELATIVE	INDIRECT
HEX			8D	85		9D	99	81	91	95			
BYTES			3	2		3	3	2	2	2			
CYCLES			4	3		5	5	6	6	4			
bbb			011	001		111	110	000	100	101			

Vlaggen:

N	V	B	D	I	Z	C

(GEEN VERANDERING)

Instruktiekodes:



STX — Berg X in geheugen op

Funktie:

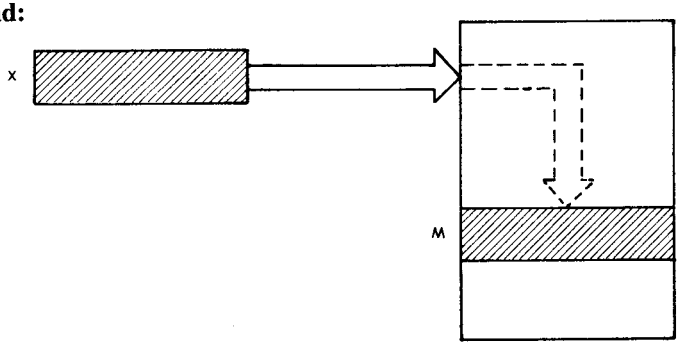
M← (X)

Formaat:

Beschrijving:

Kopieer de inhoud van X in de gespecificeerde geheugenplaats. De inhoud van X blijft onveranderd.

Datapad:



Adresseermodes:

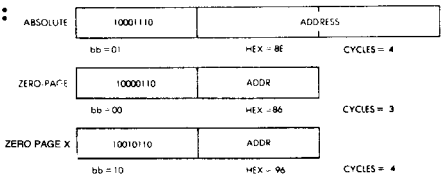
	IMPLIED	ACCUMULATOR	ABSOLUTE	0 PAGE	IMMEDIATE	ABS, X	ABS, Y	(IND, X)	(IND), Y	0 PAGE, X	0 PAGE, Y	RELATIVE	INDIRECT
HEX			8E	86							96		
BYTES			3	2							2		
CYCLES			4	3							4		
bb			01	00							10		

Vlaggen:

N	V	B	D	I	Z	C

(GEEN VERANDERING)

Instruktiekodes:



STY

Berg Y op in geheugen

Funktie:

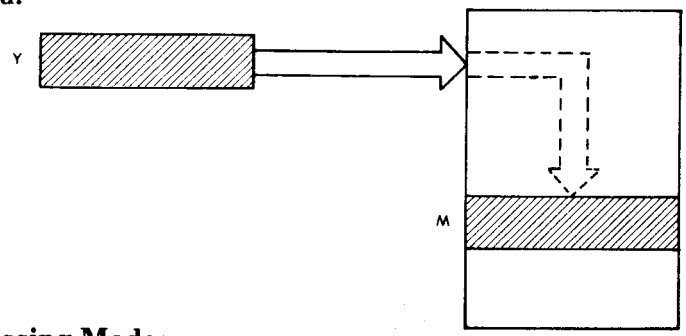
M← (Y)

Formaat:

Beschrijving:

Kopieer de inhoud van indexregister Y in de gespecificeerde geheugenplaats. De inhoud van Y blijft onveranderd.

Datapad:

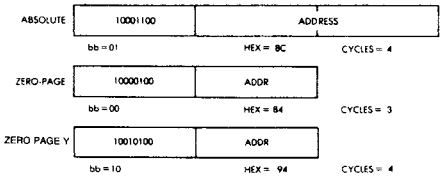


Addressing Modes:

	IMPLIED	ACCUMULATOR	ABSOLUTE	0.PAGE	IMMEDIATE	ABS. X	ABS. Y	(IND. X)	(IND. Y)	0.PAGE. X	0.PAGE. Y	RELATIVE	INDIRECT
HEX			8C	84						94			
BYTES			3	2						4			
CYCLES			4	3						4			
bb			01	00						10			

Vlaggen:

Instruktiekodes:



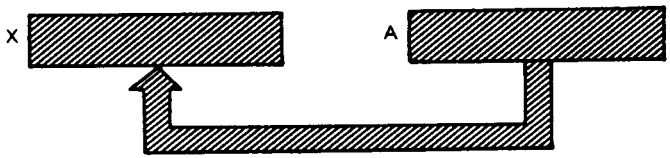
TAX — Transfereer accumulator naar X

Funktie:
X ← (A)

Formaat: 10101010

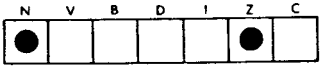
Beschrijving:
Kopieer de inhoud van de accumulator in X. De inhoud van A blijft onveranderd.

Datapad:



Adresseermode:
Alleen impliciet:
HEX = AA, byte = 1, cycles = 2

Vlaggen:



TAY

Transfereer accumulator naar Y

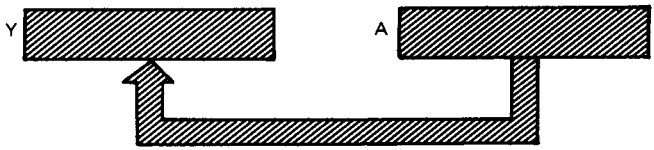
Funktie:
Y← (A)

Formaat:

10101000

Beschrijving:
Kopieer de inhoud van de accumulator in indexregister Y. De inhoud van A blijft onveranderd.

Datapad:



Adresseermode:
Alleen impliciet:
HEX = A8, byte = 1, cycles = 2

Vlaggen:

N	V	B	D	I	Z	C
●					●	

TSX — Transfereer S naar X

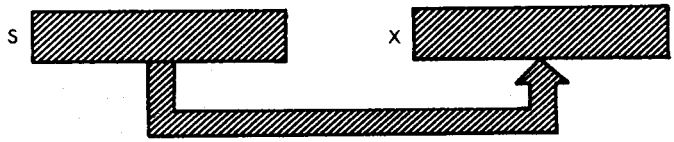
Funktie:
X← (S)

Formaat:

10111010

Beschrijving:
De inhoud van stapelpointer S wordt gekopieerd in indexregister X. De inhoud van S blijft onveranderd.

Datapad:



Adresseermode:
Alleen impliciet:
HEX = BA, byte = 1, cycles = 2

Vlaggen:

N	V	B	D	I	Z	C
●					●	

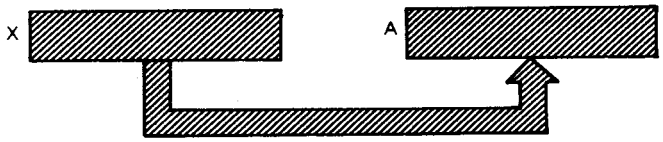
TXA — Transfereer X naar de accumulator

Funktie:
A← (X)

Formaat: 10001010

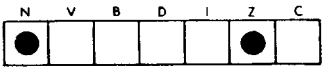
Beschrijving:
De inhoud van indexregister X wordt gekopieerd in de accumulator. De inhoud van X blijft onveranderd.

Datapad:



Adresseermode:
Alleen impliciet:
HEX = 8A, byte = 1, cycles = 2

Vlaggen:



TXS — Transfereer X naar S

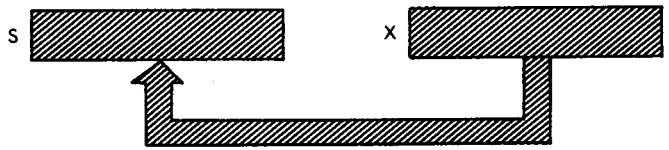
Funktie:
S ← (X)

Formaat:

10011010

Beschrijving:
De inhoud van indexregister X wordt gekopieerd in de accumulator. De inhoud van X blijft onveranderd.

Datapad:



Adresseermode:
Alleen impliciet:
HEX = 9A, byte = 1, cycles = 2

Vlaggen:

N	V	B	D	I	Z	C

(GEEN VERANDERING)

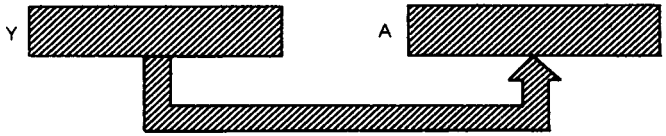
TYA — Transfereer Y naar A

Funktie:
A ← (Y)

Formaat: 10011000

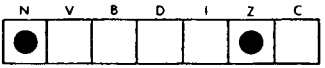
Beschrijving:
De inhoud van indexregister Y wordt gekopieerd in de accumulator. De inhoud van Y blijft onveranderd.

Datapad:



Adresseermode:
Alleen impliciet:
HEX = 98, byte = 1, cycles = 2

Vlaggen:



5
ADRESSERINGSTECHNIEKEN

INLEIDING

In dit hoofdstuk zal de algemene adresseringstheorie besproken worden, samen met de verschillende technieken die ontwikkeld zijn om eenvoudig toegang tot gegevens te verkrijgen. In een tweede deel zullen de specifieke adresseermodes, die beschikbaar zijn op de 6502, bekeken worden samen met hun voordelen en beperkingen, voorzover aanwezig. Tenslotte zullen in een deel met toepassingen de pro's en contra's van de verschillende adresseertechnieken beschouwd worden, door specifieke toepassingsprogramma's te bestuderen, om zo de lezer vertrouwd te maken met de mogelijkheden.

Omdat de 6502 geen ander 16-bits register heeft dan de programmateller, dat gebruikt kan worden om een adres te specificeren, is het nodig dat de 6502 gebruiker de verschillende adresseermodes begrijpt, en in het bijzonder het gebruik van de indexregisters. Complexe toegangsmodes, zoals een kombinatie van indirekt en geindexeerd, kunnen in het begin overgeslagen worden. Alle modes zijn echter nuttig om voor deze processor programma's te ontwikkelen. Laten we nu de verschillende beschikbare alternatieven eens bestuderen.

ADRESSEERMODES

Met adresseren wordt bedoeld de specifikatie, binnen een instruktie, van de plaats van de operand waarop de instruktie zal werken. De voornaamste methoden zullen nu beschouwd worden.

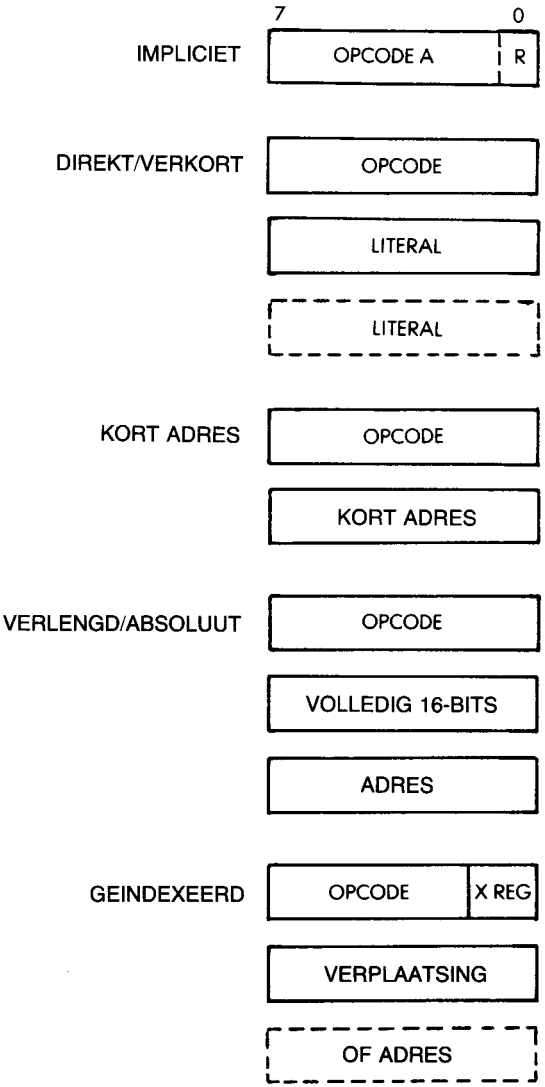


Fig. 5-1: Adressering

Impliciete adressering

Instrukties, die normaal uitsluitend op registers opereren, maken gebruik van *impliciete adressering*. Dit is geillustreerd in fig. 5-1. Een impliciete instruktie ontleent zijn naam aan het feit, dat hij niet specifiek het adres van de operand waarop hij werkt bevat. In plaats daarvan specificeert de opcode een of meer registers (meestal de accumulator, of andere registers). Omdat interne registers meestal beperkt in aantal zijn (zeg maximaal 8), zijn hier maar weinig bits voor nodig. Bijvoorbeeld: drie bits in een instruktie zijn genoeg om 1 tot 8 registers aan te wijzen. Dergelijke instrukties kunnen dus normaal in 8 bits gekodeerd worden. Dit is een belangrijk voordeel, omdat 1-byte instrukties sneller uitgevoerd worden dan een twee-of drie-byte instruktie.

Een voorbeeld van een impliciete 6502 instruktie is: TAX, die specificeert: "transfereer de inhoud van A naar X".

Immediate adressering

Immediate adressering (immediate = meteen) is geillustreerd in figuur 5-1. De 8-bit opcode wordt gevolgd door een 8-of 16-bit konstante (literal). Dit type instruktie is nodig om bijvoorbeeld een 8-bit konstante in een 8-bit register te laden. Als de microprocessor uitgerust is met 16-bits registers, kan het nodig zijn om 16-bit konstantes te laden. Dit is afhankelijk van de interne architektuur van de processor. Een voorbeeld van een immediate instruktie is: ADC #0.

Het tweede woord van deze instruktie bevat de konstante "0", die bij de accumulator opgeteld wordt.

Absolute adressering

Absolute adressering heeft betrekking op de manier, waarop gegevens meestal uit het geheugen gehaald worden, waarbij de opcode gevolgd wordt door een 16-bits adres. Voor absolute adressering zijn dus drie-byte instrukties nodig. Een voorbeeld van absolute adressering is: STA #$1234.

Deze specificeert, dat de inhoud van de accumulator opgeslagen moet worden op geheugenplaats "1234" hexadecimaal.

Het nadeel van absolute adressering is, dat een drie-byte instruktie nodig is. Om de efficientie van de microprocessor te verhogen kan een andere adresseringsmode beschikbaar zijn, waar maar een byte voor het adres gebruikt wordt: direkte adressering.

Direkte adressering

In deze adresseringsmode wordt de opcode gevolgd door een 8-bits adres. Dit is geillustreerd in fig. 5-1. Het voordeel van deze benadering is, dat slechts twee bytes nodig zijn in plaats van de drie bij absolute adressering. Het nadeel is, dat de adressering in deze mode beperkt is tot de adressen 0 tot 255. Dit is pagina 0. Dit wordt ook wel verkorte adressering genoemd, of pagina-nul adressering. Als verkorte adressering beschikbaar is, wordt absolute adressering in tegenstelling hiermee wel verlengde adressering genoemd.

Relatieve adressering

Normale spronginstrukties vereisen 8 bits voor de opcode plus het 16-bits adres, wat het adres is waar het programma heen moet springen. Net als in het voorgaande voorbeeld heeft dit het nadeel, dat er drie bytes nodig zijn, d.w.z. drie geheugencycli. Om efficienter te springen gebruikt relatieve adressering slechts een twee-byte formaat. Het eerste woord is de specificatie van de sprong, meestal samen met de test die uitgevoerd moet worden. Het tweede woord is een verplaatsing. Omdat een sprong zowel positief als negatief moet kunnen zijn, kan met een relatieve sprong 128 plaatsen vooruit of 128 plaatsen achteruit (plus of minus 1, afhankelijk van de afspraken) gesprongen worden. Omdat de meeste lussen vrij kort zijn, kunnen meestal relatieve sprongen gebruikt worden, met als gevolg een aanzienlijk verbeterde prestatie van dergelijke korte routines. Als voorbeeld hebben we de instruktie BCC al gebruikt, die specificeert: "spring als carry op nul staat" naar een plaats binnen 127 woorden van de spronginstruktie.

Geindexeerde adressering

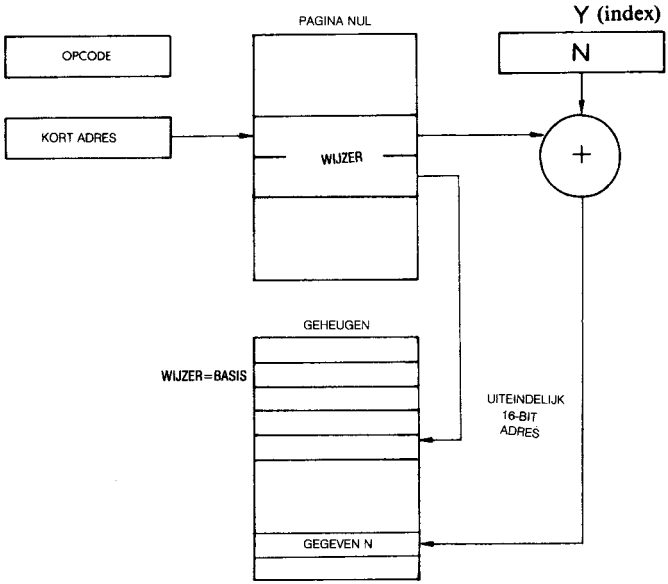
Geindexeerde adressering is een techniek, die bijzonder handig is om achtereenvolgens toegang te krijgen tot de elementen van een blok of een tabel. Dit zal verderop in dit hoofdstuk met voorbeelden geillustreerd worden. Het principe van geindexeerde adressering is, dat de instruktie zowel een indexregister als een adres specificeert. In het algemeen wordt de inhoud van het register bij het adres opgeteld om zo het uiteindelijke adres te krijgen. Op deze manier zou het adres het begin van een tabel in het geheugen kunnen zijn. Het indexregister zou dan gebruikt worden om op een efficiente manier toegang tot achtereenvolgende elementen van de tabel. In de praktijk zijn er vaak beper-

kingen wat betreft de grootte van het indexregister of de grootte van het adres- of verplaatsingsveld.

Pre-indexering en Post-indexering

Er kunnen twee indexeringsmodes onderscheiden worden. Pre-indexering is de meest gebruikte indexeringsmode, waar het uiteindelijke adres de som is van een verplaatsing of een adres en de inhoud van het indexregister.

Post-indexering beschouwt de inhoud van het verplaatsingsveld als het *adres* van de eigenlijke verplaatsing, in plaats van als de verplaatsing zelf. Dit is geillustreerd in fig. 5-2. Bij post-indexering is het uiteindelijke adres de som van de inhoud van het indexregister en de inhoud van de geheugenplaats *aangewezen door het verplaatsingsveld*. Deze methode gebruikt in feite een kombinatie van indirekte adressering en pre-indexering. Maar we hebben indirekte adressering nog niet gedefinieerd. Laten we dat nu doen.



Figuur 5-2: Indirekt geindexeerde adressering

Indirekte adressering

We hebben het geval al gezien waarin twee subroutines een grote hoeveelheid in het geheugen opgeslagen gegevens willen uitwisselen. In het algemeen kan het zijn, dat meerdere programma's of meerdere subroutines toegang nodig hebben tot een gemeenschappelijk blok informatie. Om een programma algemeen te houden, is het niet wenselijk dat een dergelijk blok op een vaste geheugenplaats staat. In het bijzonder kan de grootte van dit blok dynamisch groter of kleiner worden, en kan het op verschillende plaatsen in het geheugen staan, afhankelijk van de grootte. Het zou daarom onpraktisch zijn om toegang tot dit blok te krijgen door absolute adressen te gebruiken.

De oplossing voor dit probleem is het plaatsen van het beginadres van dit blok op een vaste geheugenplaats. Dit is analoog aan de situatie waarin verschillende personen een huis binnen moeten, terwijl er maar een sleutel is. Volgens afspraak wordt de sleutel onder de mat bewaard. Iedere gebruiker weet dan waar hij kijken moet (onder de mat) om de sleutel te vinden (of misschien om het adres van een afgesproken

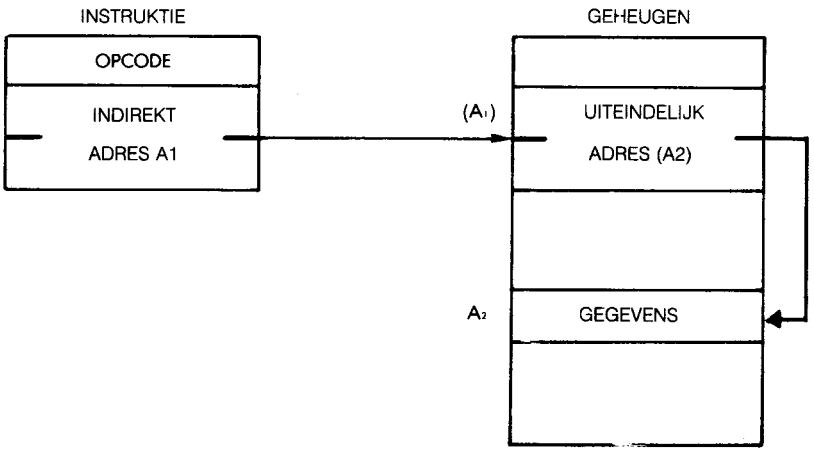


Fig. 5-3: Indirekte adressering

ontmoeting te vinden, wat een betere analogie zou zijn). Indirekte adressering gebruikt daarom een 8-bits opcode, gevolgd door een 16-bits adres. En dit adres wordt gewoon gebruikt om een woord uit het geheugen te halen. Normaal zal dit een 16-bits woord zijn (in ons geval twee bytes) in het geheugen. Dit is geillustreerd in fig. 5-3. De twee bytes op het gespecificeerde adres A1, bevatten A2. A2 wordt dan geinterpreteerd als het adres van de gegevens waar men toegang tot wil krijgen.

Indirekte adressering is bijzonder handig als er wijzers gebruikt worden. Verschillende delen van een programma kunnen dan refereren aan deze wijzers om eenvoudig en elegant toegang te krijgen tot een woord of een blok gegevens.

Kombinaties van modes

De bovengenoemde adresseringsmodes kunnen gekombineerd worden. In het bijzonder moet het in een volledig algemeen adresseringsschema mogelijk zijn om meerdere niveau's van indirektie te gebruiken. Adres A2 zou weer als een indirekt adres geinterpreteerd kunnen worden enzovoort.

Geindexeerde adressering kan ook gekombineerd worden met indirekte toegang. Hiermee kan efficient toegang tot woord n van een blok gegevens verkregen worden, vooropgesteld dat bekend is waar de wijzer naar het startadres is.

We zijn nu vertrouwd met alle gebruikelijke adresseringsmethoden waarin in een systeem voorzien kan zijn. De meeste microprocessorsystemen hebben niet alle modes, maar een kleine deelverzameling ervan vanwege de beperkte uitgebreidheid van een MPE, die op een enkele chip gerealiseerd moet worden. De 6502 kent een ongewoon uitgebreide deelverzameling van mogelijkheden. Laten we die nu bestuderen.

ADRESSERINGSMODES VAN DE 6502

Impliciete adressering (6502)

Impliciete adressering wordt gebruikt door een één-byte instruktie die werkt met interne registers. Als impliciete instrukties alleen met interne registers werken, hebben ze slechts twee klokcycli nodig om uitgevoerd te worden. Als ze toegang tot het geheugen vereisen, zijn er drie of meer nodig.

Instrukties die alleen met interne registers werken zijn: CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, SEC, SED, SEI, SEV, TAX, TAY, TSX, TXA, TXS, TYA.

Instrukties met geheugentoegang zijn: BRK, PHA, PHP, PLA, PLP, RTI, RTS.

Deze instrukties zijn in het vorige hoofdstuk besproken, en hun werking zou nu duidelijk moeten zijn.

Immediate adressering

Omdat de 6502 alleen 8-bits werkregisters heeft (de PC is geen werkregister), is immediate adressering bij de 6502 beperkt tot 8-bit konstantes. Alle instrukties in de immediate adresseringsmode zijn dus twee bytes lang. Het eerste byte bevat de opcode, en het tweede byte bevat de konstante of literal, die in een register geladen moet worden, of gebruikt gaat worden samen met een register in een logische of rekenkundige bewerking.

Instrukties die deze adresseringsmode gebruiken zijn: ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, ORA, SBC.

Absolute adressering

Absolute adressering heeft per definitie drie bytes nodig. Het eerste byte is de opcode en de volgende twee bytes zijn het 16-bits adres, waarmee het adres van de operand gespecificeerd wordt. Behalve in het geval van een absolute sprong, zijn voor deze mode vier cycli nodig.

Instrukties die absolute adressering kunnen gebruiken zijn: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, (ROR), SBC, STA ,STX, STY.

Pagina-nul adressering

Pagina-nul adressering heeft per definitie twee bytes nodig: de eerste is voor de opcode, de tweede is voor het 8-bits of verkort adres.

Voor pagina-nul adressering zijn drie cycli nodig. Omdat pagina-nul adressering naast een kortere kode aanzienlijke snelheidsvoordelen biedt, moet het als het enigszins mogelijk is gebruikt worden. Dit vereist zorgvuldig geheugenbeheer van de programmeur. In het algemeen gesproken kunnen de eerste 256 geheugenplaatsen beschouwd worden als de verzameling werkregisters van de 6502. Elke instruktie zal in slechts drie cycli op deze 256 "registers" werken (behalve instrukties

zoals ROL (ROR), ASL, LSR, die werken in vier cycli). Deze ruimte moet dus zorgvuldig gereserveerd worden voor gegevens waar snel toegang tot verkregen moet worden.

Instrukties, die pagina-nul adressering kunnen gebruiken, zijn dezelfde als die, welke absolute adressering kunnen gebruiken, behalve JMP en JSR (die een 16-bits adres nodig hebben), en SEC (die hier vervangen is door SBC).

De lijst van geldige instrukties is: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX, STY.

Relatieve adressering

Relatieve adressering heeft per definitie twee bytes nodig. Het eerste is een spronginstruktie, terwijl de tweede een verplaatsing en het teken specificeert. Om deze mode van de absolute sprong (jump) te onderscheiden, worden ze *branches* (vertakking) genoemd. Branches gebruiken bij de 6502 altijd de relatieve mode. Jumps gebruiken altijd de absolute mode (plus natuurlijk de verschillende submodes die hiermee gekombineerd kunnen worden, zoals geindexeerd en indirekt). Wat de timing betreft moet deze instruktie nauwkeurig bekeken worden. Als de test faalt, d.w.z. als er niet gesprongen wordt, duurt deze instruktie slechts twee cycli. Dit is, omdat de programmateller al naar de volgende uit te voeren instruktie wijst. Als de test echter slaagt, dus als er gesprongen wordt, duurt deze instruktie drie cycli: er moet een nieuw adres berekend worden. Voor het bijwerken van de programmateller is een extra cyclus nodig. Als er echter over een paginagrens gesprongen wordt, moet de programmteller nogmaals bijgewerkt worden en wordt de effektieve lengte van de instruktie vier cycli.

De gebruiker hoeft zich wat het overschrijden van een paginagrens betreft geen logische zorgen te maken. Daar zorgt de hardware voor. Omdat er echter bij het overschrijden van een paginagrens een carry of een borrow gegenereerd wordt, verandert de tijd waarin de sprong uitgevoerd wordt. Als deze sprong een onderdeel van een exacte timinglus is, is voorzichtigheid geboden.

Een goede assembler zal, als het programma geassembleerd wordt, doorgaans de programmeur waarschuwen dat er over een paginagrens gesprongen wordt en de timing kritiek kan zijn.

Als niet zeker is of de sprong lukt of niet, moet in overweging genomen worden dat de sprong soms twee, soms drie cycli duurt. Vaak wordt een gemiddelde tijd berekend.

De enige instrukties die relatieve adressen gebruiken, zijn de branch-instrukties. Er zijn 8 branch-instrukties, die vlaggen van het statusregister testen op "0" of "1", plus de BIT instruktie. De lijst is: BCC, BEQ, BMI, BNE, BPL, BSC, BVC, BVS.

Geindexeerde adressering (6502)

De 6502 heeft geen geheel algemene voorzieningen, maar wel beperkte. Hij is uitgerust met twee indexregisters. Deze registers zijn echter beperkt tot 8 bits. De inhoud van een indexregister wordt opgeteld bij het adresveld van een instruktie. Meestal wordt een indexregister gebruikt als teller om toegang te krijgen tot opeenvolgende elementen van een blok of tabel. Daarom zijn er speciale instrukties om de inhoud van ieder van deze registers apart te vermeerderen of te verminderen. Verder zijn er twee gespecialiseerde instrukties om de inhoud van de registers te vergelijken met een geheugenplaats, een belangrijke voorziening om effektief gebruik te maken van indexregisters bij het testen tegen eindwaarden.

In de praktijk is de beperking van de indexregisters tot 8 bits geen belangrijke beperking, omdat de meeste gebruikerstabellen meestal korter zijn dan 256 woorden.
De geindexeerde adresseringsmode kan niet alleen bij de normale absolute adressering gebruikt worden, maar ook bij pagina-nul adressering, d.w.z. bij 8 bits adresvelden.

Er is echter een beperking. Register X kan bij beide typen adressering gebruikt worden. Register Y kan echter alleen gebruikt worden bij *absoluut* geindexeerd en *niet* bij *pagina-nul* geindexeerd (behalve voor LDX en STX instrukties, die door register Y gemodificeerd kunnen worden).

Voor absoluut geindexeerde adressering zijn vier cycli nodig, tenzij een paginagrens overschreden wordt, in welk geval er vijf cycli nodig zijn.

Absoluut geindexeerde instrukties kunnen register X en Y gebruiken voor het verplaatsingsveld. De lijst van instrukties die in deze mode gebruikt kunnen worden is:

- met X: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA.
- met Y: ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC, STA (niet ASL, DEC, LSR, ROL, ROR).

In het geval van pagina-nul geindexeerde adressering is register X het enige geldige verplaatsingsregister. Geldige instrukties zijn: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC, STA, STY.

Indirekte adressering (6502)

De 6502 heeft geen volledig algemeen indirekt adresseringsmechanisme. Het adresveld is beperkt tot 8 bits. Met andere woorden, alle indirekte adresseringen gebruiken de submode van pagina-nul adressering. Het effektieve adres, waarmee de opcode werkt, zijn dan de 16 bits gespecificeerd door de inhoud van twee opeenvolgende bytes beginnende bij het pagina-nul adres van de instruktie. Er mag ook geen verdere indirektie voorkomen. Dit betekent, dat een adres wat betrokken is van pagina nul, als zodanig gebruikt moet worden, en niet gebruikt kan worden voor een verdere indirektie.

Tenslotte moeten alle indirekte toegangen geindexeerd zijn, behalve voor JMP.

Om eerlijk te zijn, moet opgemerkt worden, dat slechts weinig microprocessoren in enigerlei vorm van indirekte adressering voorzien. Verder is het mogelijk om een meer algemeen indirekt adresseringsmechanisme uit te voeren met behulp van een macro definitie.

Er zijn twee modes van indirektie adressering mogelijk: geindexeerde indirekte adressering en indirekte geindexeerde adressering (behalve met JMP, die zuiver indirekt gebruikt).

Geindexeerde indirekte adressering

Deze mode telt de inhoud van indexregister X op bij het pagina-nul adres om zo het uiteindelijk 16-bits adres te berekenen. Dit is een efficiente manier om toegang te krijgen tot een van meerdere gegevens die aangewezen worden door een pointer, waarvan het nummer in indexregister X is opgeslagen. Dit is geillustreerd in onderstaande fig. 5-4.

In deze illustratie, bevat pagina-nul en tabel van pointers. De eerste pointer wijst naar adres A, wat een deel van de instruktie is. Als de inhoud van X 2N is, dan zal deze instruktie toegang krijgen tot pointer n van deze tabel en de gegevens waar deze naar wijst ophalen.

Voor geindexeerd indirekte adressering zijn 6 cycli nodig. Het is wat tijd betreft natuurlijk minder efficient dan de direkte adresseringsmode. Het voordeel is de flexibiliteit van het koderen en de gemiddelde snelheidsverbetering.

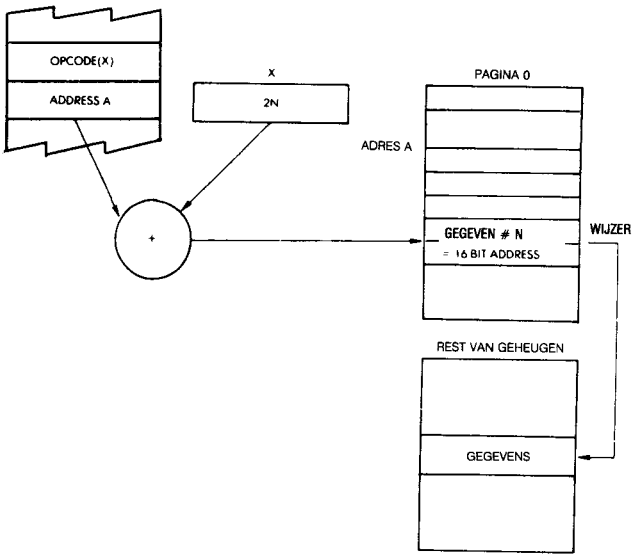


Fig. 5-4: Geindexeerd indirekt

Toegestane instrukties zijn: ADC, AND, CMP, EOR, LDA, ORA, SBC, STA.

Indirekt geindexeerde adressering

Dit komt overeen met het mechanisme van post-indexeren wat besproken is in het voorgaande deel. Daar wordt na de indirektie geindexeerd, in plaats van ervoor. Met andere woorden, het verkorte adres, wat een deel van de instruktie is, wordt gebruikt om toegang te krijgen tot een 16-bit pointer in pagina-nul. De inhoud van indexregister Y wordt hier dan bij opgeteld als een verplaatsing t.o.v. deze pointer. Dan worden de uiteindelijke gegevens opgehaald.

In dit geval wijst de pointer in pagina-nul naar de basis van een tabel in het geheugen. Indexregister Y geeft een verplaatsing. Dit is een echte index in een tabel. Deze instruktie is bijzonder krachtig om aan het n-de element van een tabel te refereren, vooropgesteld, dat het startadres van de tabel in pagina-nul zit. Dit kan in twee bytes gedaan worden.

Geldige instrukties zijn: ADC, CMP, EOR, LDA, ORA, SBC, STA.

Uitzondering: Jump instruktie

De jump instruktie kan indirekt absoluut gebruiken. Het is de enige instruktie die deze mode kan gebruiken.

HET GEBRUIK VAN DE 6502 ADRESSERINGSMODES

Lange en korte adressering

In de verschillende programma's die we ontwikkeld hebben, hebben we al branch instrukties gebruikt. Ze spreken voor zichzelf. Een interessante vraag is: wat kunnen we doen als het toegestane sprongbereik niet voldoende is? Een eenvoudige oplossing is het gebruik van de zogenaamde *lange branch*. Dit is gewoon een branch naar een plaats waar een jump gespecificeerd is:

```
BCC     +3      BRANCH NAAR HUIDIGE
                ADRES +3 ALS CARRY CLEAR
JMP     VER     SPRING ANDERS NAAR VER
                (VOLGENDE INSTRUKTIE)
```

Het programma van twee regels resulteert in een sprong naar VER als carry is nul. Dit lost ons probleem van de verre sprong op. Laten we nu de meer ingewikkelde adresseringsmodes eens beschouwen, te weten indexering en indirektie.

Het gebruik van indexering voor sequentiele bloktoegang

Indexering wordt voornamelijk gebruikt om achtereenvolgende (sequentiele) plaatsen in een tabel te adresseren. De beperking is dat de verplaatsing minder dan 256 moet zijn, zodat deze in een 8-bits indexregister past.

We hebben als geleerd, hoe we op het karakter "*" kunnen kontroleren. Nu zullen we een tabel van 100 karakters onderzoeken op de aanwezigheid van een "*". Het startadres van deze tabel is BASE. De tabel heeft maar 100 elementen. Dit is minder dan 256 en we kunnen dus een indexregister gebruiken. Het programma is als volgt:

```
SEARCH      LDX   #0
NEXT        LDA   BASE,X
            CMP   #"*"
            BEQ   STARFOUND
            INX
            CPX   #100
            BNE   NEXT
NOTFOUND    ...
STARFOUND   ...
```

Het stroomdiagram voor dit programma is gegeven in fig. 5.5. Ga de overeenkomst tussen het programma en het stroomdiagram na. De logika van het programma is erg eenvoudig. Register X wordt gebruikt om naar een element in de tabel te wijzen. De tweede instruktie van het programma:

```
NEXT LDA BASE,X
```

maakt gebruik van geindexeerde adressering. Hij specificeert dat de accumulator geladen moet worden uit adres BASE (16-bits absoluut adres) plus de inhoud van X. In het begin is de inhoud van X "0". Het eerste element dat gehaald wordt is dat van adres BASE. Je kunt nagaan, dat na de eerste iteratie X de waarde "1" heeft, en het volgende element uit de tabel gehaald wordt, van adres BASE + 1.

De derde instruktie van het programma: CMP #"*" vergelijkt de waarde van het karakter dat in de accumulator gelezen is met de kode voor "*". De volgende instruktie test het resultaat van de vergelijking. Als de waarden gelijk zijn, wordt gesprongen naar label STARFOUND:

```
        BEQ STARFOUND
```

Anders wordt gewoon de volgende instruktie uitgevoerd:

```
        INX
```

De indexteller wordt met 1 vermeerderd. Als we nu onderin het stroomdiagram van fig. 5-5 kijken, zien we dat op dit punt de waarde van ons indexregister gekontroleerd moet worden, om te voorkomen dat we de grens van de tabel overschrijden (hier 100 elementen). Dit gebeurt met de volgende instruktie:

```
CPX     #100
```

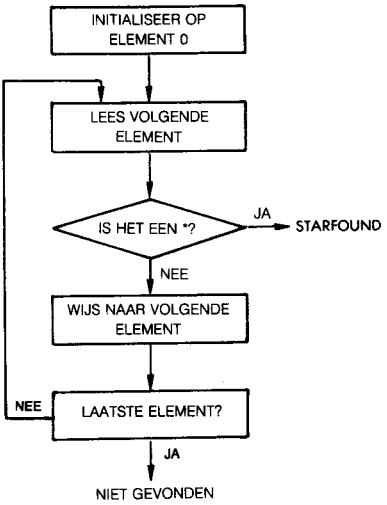


Fig. 5-5: Stroomdiagram voor zoeken van karakter

Deze instruktie vergelijkt register X met de waarde $100. Als de test faalt moeten we het volgende karakter ophalen. Dit gebeurt met:

```
BNE     NEXT
```

Deze instruktie specificeert een sprong naar label NEXT als de test faalt (de tweede instruktie in ons programma). Deze loop wordt uitgevoerd zolang nog geen "*" gevonden is of het indexregister nog niet de waarde "100" heeft. De volgende instruktie die uitgevoerd wordt is

"NOT FOUND". Dit korrespondeert met het geval dat er geen "*" gevonden is.

De akties die genomen worden als er wel of geen "*" gevonden wordt zijn hier niet van belang, en zouden door de programmeur gespecificeerd worden.

We hebben nu geleerd de geindexeerde adresseringsmode te gebruiken om toegang te krijgen tot opeenvolgende elementen in een tabel. Laten we nu deze vaardigheid eens gebruiken en de moeilijkheidsgraad iets opvoeren. We zullen een belangrijk nuttig programma (= utility) ontwikkelen voor het kopieeren van een blok gegevens van een deel van het geheugen naar een ander deel. We zullen in eerste instantie aannemen, dat het aantal elementen in een blok minder dan 256 is, zodat we indexregister X kunnen gebruiken. Daarna zullen we het algemene geval beschouwen, waarin het aantal elementen groter is dan 256.

Een blokoverdrachtroutine voor minder dan 256 elementen

We zullen "NUMBER" het aantal elementen in het blok noemen dat verschoven moet worden. Aangenomen wordt, dat dit aantal kleiner is dan 256. BASE is het beginadres van het blok. DESTINATION is het beginadres van het geheugengebied waar het heengeschoven moet worden. Het algoritme is erg eenvoudig: we verplaatsen een woord per keer en houden bij welk woord we verplaatsen door zijn positie in register X op te slaan. Het programa is als volgt:

```
        LDX   #NUMBER
NEXT    LDA   BASE,X
        STA   DEST,X
        DEX
        BNE   NEXT
```

Laten we het eens onderzoeken:

```
        LDX   # NUMBER
```

Deze regel van het programma laadt het aantal over te dragen woorden (N) in het indexregister. De volgende instruktie laadt het Nde woord van het blok in de accumulator en de derde instruktie deponeert deze in het bestemmingsgebied. Zie fig. 5-6.

Waarschuwing: Dit programma werkt alleen korrekt als aangenomen wordt, dat het basisregister juist onder het blok wijst, net als het be-

stemmingsregister. Anders is een kleine aanpassing van het programma nodig.

Nadat een woord overgedragen is van de bron naar het bestemmingsgebied, moet het indexregister bijgewerkt worden. Dit gebeurt met de instruktie DEX, die hem vermindert. Dan test het programma eenvoudig of de teller al tot 0 veminderd is. Als dit zo is, dan is het programma beeindigd. Anders gaat het nogmals de loop in door terug te gaan naar NEXT.

Je zult opmerken, dat als X = 0, het programma niet in de lus komt. Daarom wordt het woord op plaats BASE niet overgedragen. Het laatste woord dat overgedragen wordt is dat van plaats BASE + 1. Daarom hebben we aangenomen, dat de basis net *onder* het blok wees.

Oefening 5.1: Verander het bovenstaande programma, aannemende dat BASE en DEST naar de eerste ingang van het blok wijzen.

Dit programa illustreert ook het gebruik van lustellers. Je zult opgemerkt hebben, dat X geladen is met de *eindwaarde* en daarna *verminderd* en getest is. Op het eerste gezicht lijkt het eenvoudiger om met "0" in X te beginnen en dan te vermeerderen tot de maximale waarde. Om echter te testen of X zijn maximale waarde bereikt heeft, zou een extra instruktie nodig zijn (de vergelijkinstruktie). Voor de lus zouden dan vijf in plaats van vier instrukties nodig zijn. Omdat het programma meestal voor grote hoeveelheden woorden gebruikt wordt, is het van aanzienlijk belang dat het aantal instrukties in de lus beperkt wordt. Om deze reden wordt, tenminste voor korte lussen, het indexregister meestal *verminderd in plaats van vermeerderd.*

Een blokoverdrachtroutine (meer dan 256 elementen)

Laten we nu eens het algemene geval beschouwen van het verplaatsen van een blok met meer dan 256 elementen. We kunnen nu niet langer het indexregister gebruiken, omdat 8 bits niet genoeg zijn om een getal groter dan 256 op te slaan. De geheugenorganisatie voor dit programma is geillustreerd in figuur 5-7. Voor de lengte van het te verplaatsen geheugenblok zijn 16 bits nodig, en deze wordt dus in het geheugen opgeslagen. Het hoge orde deel representeert het aantal 256-woord blokken: "BLOCKS". De rest wordt "REMAIN" genoemd en is het aantal woorden dat overgedragen moet worden nadat alle blokken overgedragen zijn. Het adres van de bron en de bestemming zijn in geheugenplaatsen FROM en TO. Laten we eerst aannemen, dat RE-

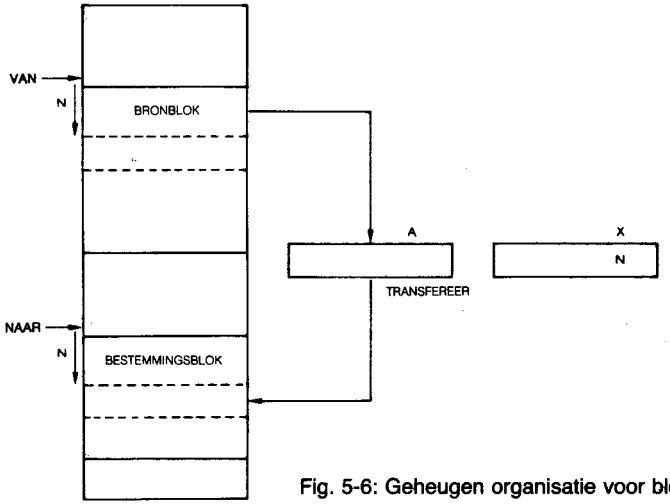


Fig. 5-6: Geheugen organisatie voor blok overdracht

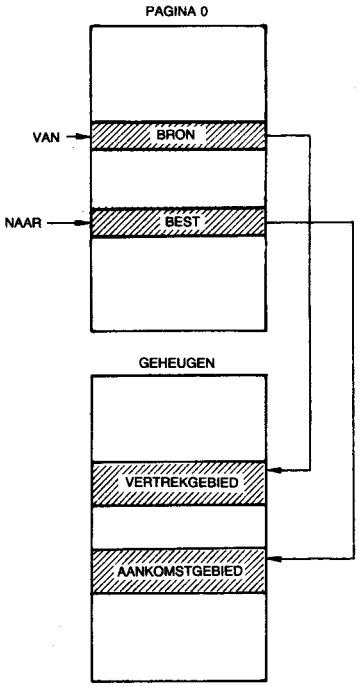


Fig. 5-7: Geheugenplan voor algemene blok overdracht

MAIN nul is, d.w.z. dat we 256-woord blokken overdragen. Het programma is:

```
        LDA  #SOURCELO
        STA  FROM
        LDA  #SOURCEHI
        STA  FROM + 1    BERG BRONADRES OP
        LDA  #DESTLO
        STA  TO
        LDA  #DESTHI
        STA  TO + 1      BERG BESTEM-ADRES OP
        LDX  #BLOCKS     HOEVEEL BLOKKEN
        LDY  #0          BLOK GROOTTE
NEXT    LDA  (FROM), Y   LEES ELEMENT
        STA  (TO), Y     BRENG HET OVER
        DEY              WERK WOORDTELLER BIJ
        BNE  NEXT        KLAAR?
NEXBLK  INC  FROM + 1    VERMEERDER BLOK TELLER
        INC  TO + 1      IDEM
        DEX              BLOKTELLER
        BMI  DONE
        BNE  NEXT
        LDY  #REMAIN
        BNE  NEXT
```

Het 16-bit bronadres wordt door de eerste vier instrukties opgeborgen op adres "FROM". De volgende vier instrukties doen hetzelfde voor de bestemming, die opgeborgen wordt op adres "TO". Omdat we meer dan 256 woorden moet overbrengen, gebruiken we gewoon twee 8-bits indexregisters. De volgende instruktie laadt register X met het aantal over te dragen blokken. Dit is de negende in het programma. De volgende instruktie laadt de waarde nul in indexregister Y om deze te initialiseren voor de overdracht van 256 woorden. We zullen nu geindexeerd indirekte adressering gebruiken. Er moet aan gedacht worden, dat geindexeerd indirekt resulteert in een indirektie via pagina nul en vervolgens een geindexeerde toegang tot het 16-bits adres gespecificeerd door het indexregister. In het programma:

```
NEXT LDA (FROM), Y
```

Deze instruktie laadt de accumulator met de inhoud van de geheugenplaats waarvan het adres is: de bron plus de inhoud van indexregister Y. Zie fig. 5-7 voor de geheugenindeling. Hier is de beginwaarde van register Y "0". "A" wordt dus geladen van geheugenadres "BRON". Merk op, dat we hier in tegenstelling met ons vorige voorbeeld aannemen, dat "BRON" het adres van het eerste woord in het blok is.

Gebruik makend van dezelfde techniek deponeert de volgende instruktie de inhoud van de accumulator (het eerste woord van het blok dat we over willen dragen) op de gewenste plaats van bestemming:

```
STA (TO), Y
```

Net als in het vorige geval verminderen we gewoon het indexregister en gaan 256 maal de loop door. Dit wordt uitgevoerd door de volgende twee instrukties:

```
DEY
BNE NEXT
```

Let op: om kompakt te programmeren is een programmeertruuk gebruikt. De alerte lezer zal opgemerkt hebben, dat indexregister Y *verminderd* wordt. Het eerste woord wat overgedragen wordt zal dus het woord op positie 0 zijn. Het volgende is woord 255. Dit is omdat het verminderen van 0 allemaal enen geeft in het register (of 255). De lezer moet zich er dus van verzekeren, dat er geen fout gemaakt is. Als register Y verminderd is tot 0, vindt er *geen* overdracht plaats. De volgende instruktie die uitgevoerd wordt is: NEXBL. Er worden dus 256 woorden overgebracht. Deze truuk had natuurlijk ook in het vorige programma gebruikt kunnen worden om een korter programma te krijgen.

Als eenmaal een volledig blok overgebracht is, hoeft alleen maar naar de volgende pagina gewezen te worden in ons oorspronkelijke blok en het blok van bestemming. Dit kan gebeuren door "1" op te tellen bij het hoge orde deel van het adres van de bron en de bestemming. Dat kan met de volgende twee instrukties in het programma:

```
NEXBL   INC   FROM+1
        INC   TO+1
```

Nadat de paginawijzer vermeerderd is, kontroleren we eenvoudig of we al dan niet nog een blok moeten overbrengen door de blokteller in X te verminderen. Dit wordt gedaan door:

```
DEX
```

Als alle blokken overgebracht zijn, verlaten we het programma door naar DONE te springen:

```
BMI DONE
```

Anders hebben we twee mogelijkheden: of we hebben nog niet tot 0 verminderd, of we hebben precies tot nul verminderd. Als we nog niet tot 0 verminderd hebben, springen we naar NEXT:

```
BNE NEXT
```

Als we precies tot 0 verminderd hebben, moeten we nog de door REMAIN gespecificeerde woorden overbrengen. Dit is het laatste deel van onze overdracht. Dit gebeurt door:

```
LDY #REMAIN
```

die indexregister Y laadt met de hoeveelheid.
Dan springen we terug naar NEXT:

```
BNE NEXT
```

De lezer verzekere zich ervan, dat tijdens de laatste lus waarin de spronginstruktie naar NEXT uitgevoerd wordt, de volgende keer dat we NEXBL binnenkomen, we inderdaad voorgoed het programma verlaten. Dit is omdat de index X voor binnenkomst in NEXBL de waarde 0 had. De derde instruktie van NEXBL verandert deze in −1, en we gaan naar DONE.

Twee blokken optellen

Dit voorbeeld geeft een eenvoudige illustratie van het gebruik van een indexregister voor de optelling van twee blokken van minder dan 256 elementen. Dan zal het volgende programma gebruik maken van indirekt geindexeerd om blokken te adresseren, waarvan bekend is, dat het adres zich op een bepaalde plaats bevindt, maar waarvan het eigenlijke absolute adres niet bekend is. Het programma is:

```
BLKOP   LDY   #NMR-1---  LAAD TELLER
NEXT    CLC
        LDA   PTR1, Y---  LEES VOLGENDE ELEMENT
        ADC   PTR2, Y     TEL OP
        STA   PTR3, Y     BERG RESULTAAT OP
        DEY               VERLAAG TELLER
        BPL   NEXT        KLAAR?
```

Index Y wordt gebruikt als indexteller en wordt geladen met het aantal elementen min een. We nemen aan, dat pointer PTR1 naar het eerste element van Blok 1 wijst, PTR2 naar het eerste element van Blok 2

en PTR3 wijst naar het bestemmingsgebied, waar de resultaten opgeborgen moeten worden.

Het programma spreekt voor zichzelf. Het laatste element van Blok 1 wordt in de accumulator geladen en dan opgeteld bij het laatste element van Blok 2. Het wordt dan opgeborgen op de juiste plaats in Blok 3. Het daarop volgende element wordt opgeteld, enzovoort.

Dezelfde oefening, gebruik makend van indirekte adressering

We nemen hier aan, dat de adressen PTR1, PTR2 en PTR3 in eerste instantie niet bekend zijn. We weten echter, dat ze opgeborgen zijn in pagina nul op adressen LOC1, LOC2 en LOC3. Dit is een gebruikelijke techniek om informatie door te geven tussen subroutines. Het korresponderende programma is:

```
BLKOP   LDY   #NMR-1
NEXT    CLC
        LDA   (LOC1),Y
        ADC   (LOC2),Y
        STA   (LOC3),Y
        DEY
        BPL   NEXT
```

De overeenkomst tuusen dit programma en het voorgaande zou nu duidelijk moeten zijn. Het illustreert duidelijk het gebruik van het geindexeerd indirekt mechanisme als het absolute adres niet bekend is op het moment dat het programma geschreven is, maar de plaats van de informatie wel bekend is. Opgemerkt kan worden, dat de twee programma's evenveel instrukties hebben. Een interessante oefening is nu te bepalen welke sneller uitgevoerd wordt.

Oefening 5.2: Bepaal het aantal bytes en het aantal cycli van ieder van deze twee programma's, gebruik makend van de tabellen in de Appendix.

SAMENVATTING

Er is een volledige beschrijving van adresseermodes gegeven. Getoond is, dat de 6502 het meeste van de mogelijke mechanismen biedt, en zijn kenmerken zijn geanalyseerd. Tenslotte zijn enkele toepassingsprogramma's gegeven om de waarde van ieder van deze adresseertechnieken te demonstreren. Voor het programmeren van de 6502 moeten deze mechanismen begrepen worden.

OEFENINGEN

5.3: Schrijf een programma dat de eerste 10 bytes optelt van een tabel beginnend op plaats "BASE". Het resultaat past in 16 bits. (Dit is een kontrolesom berekening).

5.4: Kun je hetzelfde probleem oplossen zonder indexering toe te passen?

5.5: Keer de volgorde van de 10 bytes van deze tabel om. Berg het resultaat op vanaf adres "REVER".

5.6: Bepaal het grootste element van deze tabel. Sla dit op in geheugenplaats "GROOT"

5.7: Tel de overeenkomstige elementen van drie tabellen bij elkaar op, waarvan de bases zijn: BASE1, BASE2, BASE3. De lengte van de tabellen is opgeslagen in adres "LENGTE" van pagina nul.

6
INVOER/UITVOER TECHNIEKEN

INLEIDING

We hebben tot nu toe geleerd, hoe we informatie moeten uitwisselen tussen het geheugen en de verschillende registers van de processor. We hebben geleerd, hoe we de registers moeten behandelen en hoe we een varieteit van instrukties kunnen gebruiken om gegevens te manipuleren. We moeten nu leren hoe we met de buitenwereld kunnen kommuniceren. Dit wordt de invoer/uitvoer genoemd.

Invoer heeft betrekking op het halen van gegevens van de randapparatuur (toetsenbord, schijven of sensors). *Uitvoer* heeft betrekking op het verzenden van gegevens van het geheugen of de microprocessor naar externe apparaten als printers, een beeldbuis-terminal, schijven of relais.

We zullen twee stappen onderscheiden. Eerst zullen we leren om invoer/uitvoer bewerkingen te verrichten zoals vereist door gebruikelijke apparaten. Vervolgens zullen we leren hoe we gelijktijdig meerdere invoer/uitvoer apparaten moeten behandelen, d.w.z. in welke volgorde de verschillende in- en uitvoerapparaten afgehandeld worden. Dit tweede deel zal in het bijzonder ingaan op polling vs. interrupts.

INVOER/UITVOER

In dit deel zullen we leren hoe we eenvoudige signalen zoals pulsen kunnen detekteren en genereren. Dan zullen we technieken bestuderen om korrekte timing te realiseren en te meten. Dan zullen we klaar

zijn voor meer complexe soorten I/O, zoals serie-en parallel overdracht met grote snelheid.

Genereren van een signaal

In het eenvoudigste geval wordt een output apparaat afgezet (of aangezet) door de computer. Om de status van een uitvoerapparaat te veranderen, zal de programmeur een niveau van een logische "0" in een logische "1", of een "1" in een "0" veranderen. Laten we aannemen, dat een extern relais verbonden is met bit "0" van een register genaamd "OUT1". Om het aan te schakelen schrijven we gewoon een "1" in de juiste bitpositie van het register. We nemen hier aan, dat OUT1 het adres representeert van dit outputregister in ons systeem. Het programa dat het relais aanschakelt is:

```
TURNON   LDA   #%00000001
         STA   OUT1
```

We hebben aangenomen, dat de toestand van de andere bits van register OUT1 irrelevant zijn. Dit is echter niet altijd het geval. Deze bits zouden met andere relais verbonden kunnen zijn. Laten we daarom dit eenvoudige programma verbeteren. We willen het relais aanschakelen zonder de toestand van de andere bits in het register te veranderen. We zullen aannemen, dat het mogelijk is om zowel te lezen uit als te schrijven in het register. Ons verbeterd programma wordt dan:

```
TURNON LDA  OUT1        LEES INHOUD VAN OUT1
       ORA  #%00000001  FORCEER BIT 0 NAAR "1"
       STA  OUT1
```

Dit programma leest eerst de inhoud van plaats OUT1, en verricht dan een inclusive OR met de inhoud ervan. Dit verandert alleen bitpositie 0 in een "1", en laat de rest intact. (Voor meer details van de ORA bewerking, zie hoofdstuk 4.) Dit is geillustreerd in fig. 6-1.

Pulsen

Een puls wordt op dezelfde manier als een niveau (zie boven) gemaakt. Een uitvoerbit wordt eerst aangezet, en later weer afgezet. Het resultaat is een puls. Dit is geillustreerd in fig. 6-2. Nu moeten we echter nog een ander probleem oplossen: de puls moet een bepaalde tijd duren. Laten we daarom het genereren van een vertraging eens bestuderen.

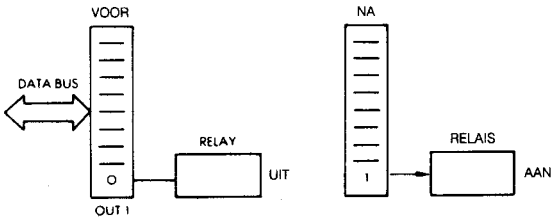


Fig. 6-1: Een relais aanschakelen

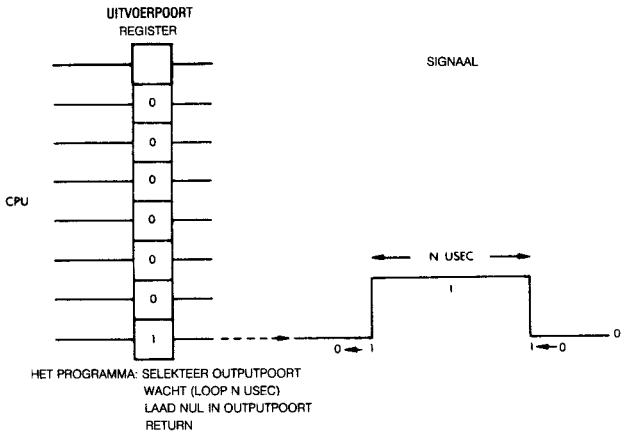


Fig. 6-2: Een geprogrammeerde puls

Veroorzaken en meten van een vertraging

Een vertraging kan met software of met hardware methoden veroorzaakt worden. We zullen hier bekijken hoe we dit met een programma kunnen doen, en later laten zien hoe het met een hardware teller, een programmeerbare interval timer (PIT) genaamd, kan gebeuren.

Geprogrammeerde vertragingen worden verkregen door te tellen. Een teller wordt geladen met een waarde, die verminderd wordt. Het programma blijft in de loop tot de teller de waarde "0" krijgt. De totale tijdsduur die voor dit proces nodig is geeft de gewenste vertraging. Laten we bijvoorbeeld een vertraging van 37 microseconden genereren.

```
DELAY   LDY   #07     Y IS TELLER
NEXT    DEY           VERMINDER
        BNE   NEXT    TEST
```

Dit programma laadt indexregister Y met de waarde 7. De volgende instruktie vermindert Y, en de volgende instruktie veroorzaakt een sprong naar NEXT zolang als Y nog niet "0" geworden is. Als Y uiteindelijk "0" geworden is, komt het programma uit de lus en voert de volgende instruktie uit. De logika van dit programma is eenvoudig en is te zien in het stroomdiagram van fig. 6-3.

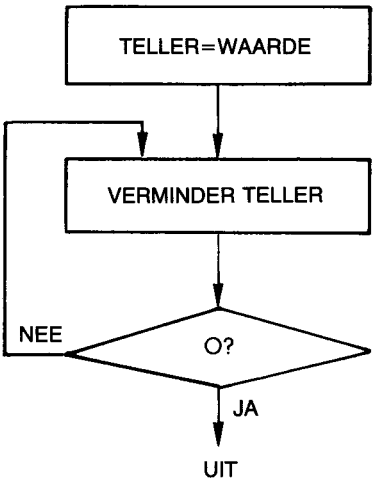


Fig. 6-3: Stroomdiagram van een vertraging

We zullen nu de effektieve vertraging die door dit programma veroorzaakt wordt eens berekenen. In de Appendix zoeken we het aantal cycli dat ieder van deze instrukties nodig heeft op:

LDY heeft in immediate mode 2 cycli nodig. DEY heeft er 2 nodig. Tenslotte heeft BNE er drie nodig. Kontroleer bij het opzoeken van het aantal cycli voor BNE, dat er drie mogelijkheden zijn: als er niet gesprongen wordt heeft BNE maar 2 cycli nodig. Als er wel gesprongen wordt, wat tijdens de lus het geval zal zijn, is een extra cyclus nodig. Tenslotte is nog een extra cyclus nodig als een paginagrens overschreden wordt. We zullen hier aannemen, dat dat niet het geval is.

De timing is dus 2 cycli voor de eerste instruktie, plus vijf cycli voor

de volgende twee, vermenigvuldigd met het aantal keren dat de lus doorlopen wordt.
Vertraging = 2 + 5 × 7 = 37.

Als we een cyclustijd van 1 microseconde aannemen, duurt deze geprogrammeerde vertraging 37 microseconden.

We kunnen uit dit eenvoudige voorbeeld zien, dat de maximale definitie, waarmee we de vertraging kunnen aanpassen, 2 microseconde is. De minimale vertraging is 2 microseconden.

Oefening 6.1: Wat is de maximale vertraging, die met deze drie instrukties uitgevoerd kan worden?

Oefening 6.2: Verander het programma, zodat een vertraging van ongeveer 100 microseconden verkregen wordt.

Als men langere vertragingen wil uitvoeren, is een simpele oplossing het invoegen van extra instrukties tussen DEY en BNE. De eenvoudigste manier om dit te doen is NOP instrukties toe te voegen. (De NOP instruktie doet 2 cycli lang niets.)

Langere vertragingen

Voor het software veroorzaken van langere vertragingen kan een bredere teller gebruikt worden. Twee interne registers, of liever, twee woorden in het geheugen, kunnen een 16-bits teller bevatten. Laten we ter vereenvoudiging aannemen, dat de laagste teller "0" is. Het lage byte wordt met "255" geladen, de maximale telling, en gaat dan door een verminderingsloop. Steeds als het tot "0" verminderd is, wordt het hoge byte van de teller met 1 verminderd. Als het hoge byte verminderd is tot "0", is het programma beeindigd. Als een hogere precisie nodig is, kan de lagere telling een waarde ongelijk aan nul hebben. In dit geval zouden we het programma schrijven zoals uitgelegd is, en aan het einde het vertragingsprogramma van drie regels toevoegen, dat we hierboven beschreven hebben.

Natuurlijk kunnen langere vertragingen veroorzaakt worden door meer dan twee woorden te gebruiken. Dit is analoog aan de werking van de snelheidsmeter in een auto. Als het meest rechtse wiel van "9" naar "0" gaat, wordt het volgende wiel met 1 vermeerderd. Dit is het algemene principe als er met meerdere afzonderlijke eenheden geteld wordt.

Het voornaamste bezwaar is, als men lange vertragingen aftelt, dat

de microprocessor gedurende honderden milliseconden of zelfs seconden niets doet. Als de computer verder niets te doen heeft, is dit volledig aanvaardbaar. In het algemeen moet de computer echter beschikbaar zijn voor andere taken, zodat langere vertragingen doorgaans niet met software uitgevoerd worden. In feite kunnen zelfs korte vertragingen bezwaarlijk zijn in een systeem, als er in bepaalde omstandigheden een gegeven responstijd gegarandeerd moet worden. Dan moet er gebruik gemaakt worden van hardware vertragingen. Verder kan, als er interrupts gebruikt worden, de timing nauwkeurigheid verloren gaan als de tel-lus onderbroken kan worden door een interrupt.

Oefening 6.3: Schrijf een programma om een vertraging van 100 milliseconden uit te voeren (voor een Teletype).

Hardware vertragingen

Hardware vertragingen worden uitgevoerd door gebruik te maken van een programmeerbare interval timer, of kortweg "timer" genaamd. Een register van de timer wordt met een waarde geladen. Het verschil is, dat vanaf dit ogenblik de timer automatisch deze teller periodiek vermindert. Deze periode kan doorgaans door de programmeur aangepast of ingesteld worden. Als de teller verminderd is tot "0", stuurt de timer meestal een interrupt naar de processor. Hij kan ook een statusbit zetten, dat periodiek door de computer afgetast wordt. Het gebruik van interrupts zal verderop in dit hoofdstuk uiteengezet worden.

Andere modes waarin de timer kan werken kunnen zijn het starten bij "0" en vervolgens tellen van de lengte van een signaal, of anders het tellen van het aantal ontvangen pulsen. Als hij werkt als een interval timer, zeggen we dat de timer in *one-shot* mode werkt. Als hij pulsen telt, werkt hij in de *pulse-counting* mode. Sommige timers hebben meerdere registers en enkele optionele faciliteiten, die door het programma te selekteren zijn. Dit is bijvoorbeeld het geval bij de timers in de 6522, een I/O chip die in het volgende hoofdstuk beschreven wordt.

Het detekteren van pulsen

Het detekteren van pulsen is het omgekeerde probleem van het veroorzaken van pulsen, plus een extra moeilijkheid: terwijl een uitvoerpuls onder programmakontrole veroorzaakt wordt, treden inputpulsen *asynchroon* met het programma op. Om een puls te detekteren kunnen

twee methoden toegepast worden: *polling* en *interrupts*. Interrupts worden later in dit hoofdstuk besproken.

Laten we nu de pollingtechniek eens beschouwen. Bij deze techniek leest het programma kontinu de waarde van een gegeven inputregister, en test een bepaalde bitpositie, bijvoorbeeld bit 0. Aangenomen wordt, dat bit 0 oorspronkelijk "0" is. Als een puls ontvangen is, wordt dit bit "1". Het programma houdt voortdurend bit 0 in de gaten, tot het de waarde "1" aanneemt. Zodra een "1" gevondenis, is de puls gedetekteerd. Het programma is als volgt:

```
POLL    LDA   #$01
AGAIN   BIT   INPUT
        BPL   AGAIN
ON      ...
```

Omgekeerd, laten we aannemen, dat de inputlijn normaal "1" is en dat we een "0" willen detekteren. Dit is het normale geval om eem START bit te detekteren, als een lijn die verbonden is met een Teletype gekontroleerd moet worden.
Het programma is:

```
POLL    LDA   #$01
NEXT    BIT   INPUT
        BMI   NEXT
START   ...
```

Het bepalen van de lengte

Het bepalen van de lengte van een puls kan op dezelfde manier gebeuren als het berekenen van de lengte van een outputpuls. Een hardware of een software techniek kan gebruikt worden. Als de lengte van een puls door middel van software bepaald wordt, wordt een teller regelmatig met 1 vermeerderd, waarna de aanwezigheid van een puls gekontroleerd wordt. Als de puls nog aanwezig is, blijft het programma in de loop. Zodra de puls verdwijnt, wordt de telling in het telregister gebruikt om de effektieve duur van de puls te berekenen. Het programma is:

```
DURTN   LDX   #0      CLEAR TELLER
        LDA   #$01    KONTROLEER BIT 0
AGAIN   BIT   INPUT
        BPL   AGAIN
```

```
LONGER INX
       BIT    INPUT
       BMI    LONGER
```

Natuurlijk zorgen we ervoor, dat de maximale duur van de puls niet register X doet overstromen. Als dit het geval zou zijn, zou het programma langer moeten zijn om hier rekening mee te houden (anders zou het een programmeerfout zijn!).

Omdat we nu weten, hoe we pulsen moeten veroorzaken en detekteren, gaan we nu grotere hoeveelheden gegevens ontvangen en verzenden. We zullen twee gevallen onderscheiden: seriele en parallelle gegevens. Daarna zullen we deze kennis toepassen op echte invoer/uitvoer komponenten.

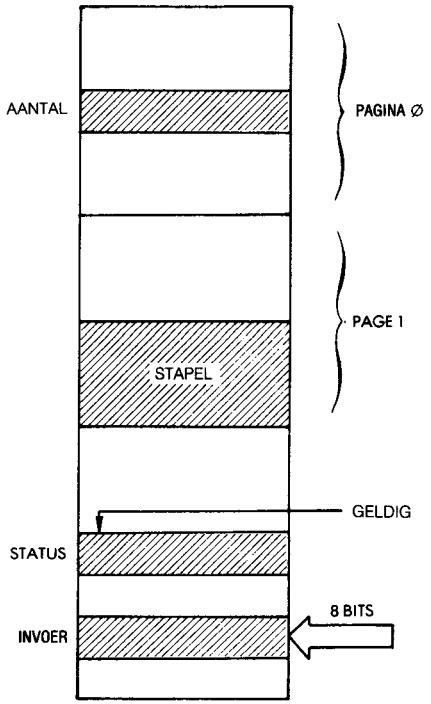


Fig. 6-4: Parallelle woordoverdracht: het geheugen

PARALLELLE WOORDOVERDRACHT

We nemen hier aan, dat de acht bits gegevens die overgedragen moeten worden, parallel beschikbaar zijn op adres "INPUT". De microprocessor moet dit datawoord (gegevenswoord) van deze plaats lezen, zodra een statuswoord aangeeft dat deze geldig is. We nemen aan dat deze statusinformatie vervat is in bit 7 van adres "STATUS". We zullen hier een programma schrijven, dat automatisch ieder datawoord leest zodra het binnenkomt. Ter vereenvoudiging zullen we aannemen, dat het aantal te lezen woorden van te voren bekend is, en opgeslagen is in locatie "COUNT". Als deze informatie niet beschikbaar zou zijn, zouden we testen op een zogenaamd *break karakter* zoals een *rubout* (uitgummen), of misschien het karakter "*". We hebben al geleerd hoe we dit moeten doen.

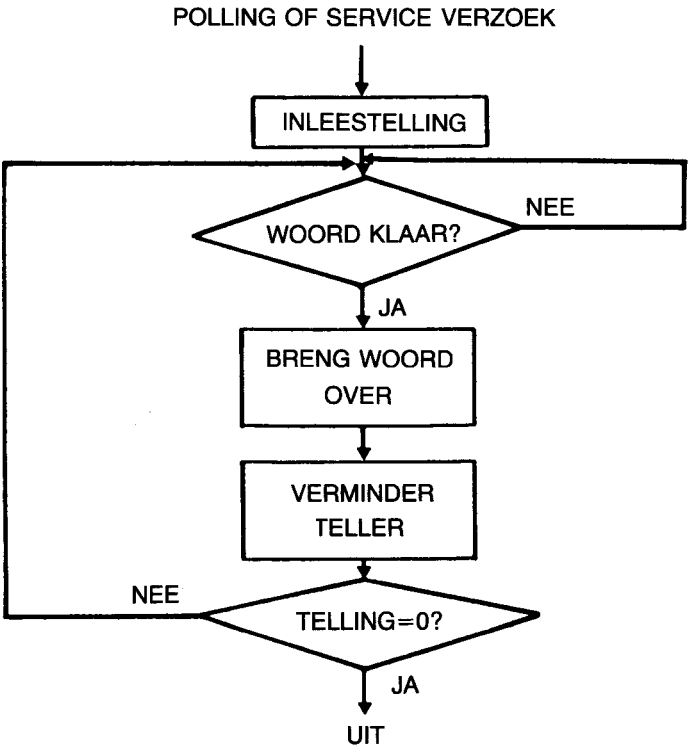


Fig. 6-5: Parallelle woordoverdracht: Stroomdiagram

Het stroomdiagram is te zien in fig. 6-5. Het is erg eenvoudig. We testen de statusinformatie tot deze "1" wordt, aangevende dat het woord klaar is. Als het woord klaar is, lezen we het, en slaan het op in een geschikte geheugenplaats. Dan verminderen we de teller en testen of die al nul geworden is. Zo ja, dan zijn we klaar: zo niet, dan lezen we nog een woord. Het programma dat dit algoritme uitvoert is onderstaand gegeven:

```
PARAL   LDX   TEL      TELLER
KIJK    LDA   STATUS   BIT 7 IS "1" ALS DATA
                       GELDIG
        BPL   KIJK     DATA GELDIG?
        LDA   INPUT    LEES HET
        PHA            BERG OP IN STAPEL
        DEX
        BNE   KIJK
```

De eerste twee instrukties van het programma lezen de statusinformatie en zorgen dat het programma in een loop blijft zolang bit 7 van het statuswoord "0" is (Dit is het tekenbit, bit N).

```
KIJK    LDA   STATUS
        BPL   KIJK
```

Als BPL faalt, is de data geldig en kunnen we die lezen:

```
        LDA   INPUT
```

Het woord is nu gelezen van adres INPUT en moet opgeslagen worden. Aannemende dat het aantal woorden klein genoeg is, gebruiken we

```
        PHA
```

Als de stapel vol kan zijn, of het aantal woorden groot is, zouden we ze niet op de stapel kunnen pushen en zouden we ze moeten overdragen naar een toegewezen geheugengebied, bijvoorbeeld met een geindexeerde instruktie. Dit zou echter een extra instruktie vergen om de indexteller te vermeerderen of te verminderen. PHA is sneller.

Het datawoord is nu gelezen en opgeslagen. We verminderen nu gewoon de woordteller en testen of we klaar zijn:

```
        DEX
        BNE   KIJK
```

We bljven in de loop tot de teller nul wordt. Dit programma van 6 in-

strukties kan een *benchmark* genoemd worden. Een benchmark programma is een zorgvuldig geoptimaliseerd programma, ontworpen om de kapaciteiten van een gegeven processor in een specifieke situatie te testen. Parallelle overdracht is zo'n specifieke situatie. Dit programma is ontworpen voor maximale snelheid en efficiency. Laten we nu de maximale snelheid van dit programma berekenen. We zullen aannemen, dat TEL in pagina nul staat. De duur van iedere instruktie kunnen we vinden door de tabel achterin het boek te inspekteren en is als volgt:

```
                          CYCLI
      LDX  TEL            3
KIJK  LDA  STATUS         4
      BPL  KIJK           2/3 (FALEN/SLAGEN)
      LDA  INPUT          4
      PHA                 3
      DEX                 2
      BNE  KIJK           2/3 (FALEN/SLAGEN)
```

De minimale uitvoertijd wordt verkregen door aan te nemen, dat, iedere keer als we STATUS kontroleren, er data beschikbaar is. Met andere woorden, we nemen aan dat de eerste BPL iedere keer faalt. De timing wordt dan: $3 + (4+2+4+3+3) \times$ TEL.

Als we de eerste drie microseconden, die nodig zijn om het indexregister te initialiseren, verwaarlozen, wordt de tijd om een woord over te dragen 18 microseconden.

De maximale snelheid van gegevensoverdracht is dan:

$$\frac{1}{18(10^{-6})} = 55 \text{ K bytes per seconde}$$

Oefening 6.4: Neem aan dat het aantal over te dragen woorden groter is dan 256. Verander het programma overeenkomstig en bepaal de invloed op de maximale overdrachtssnelheid.

We hebben nu geleerd hoe we op hoge snelheid parallel gegevens over kunnen dragen. We zullen nu een ingewikkelder geval beschouwen.

SERIE BIT OVERDRACHT

Bij een serie ingang komen de informatiebits achter elkaar binnen op een lijn. Deze bits kunnen met regelmatige intervallen binnenko-

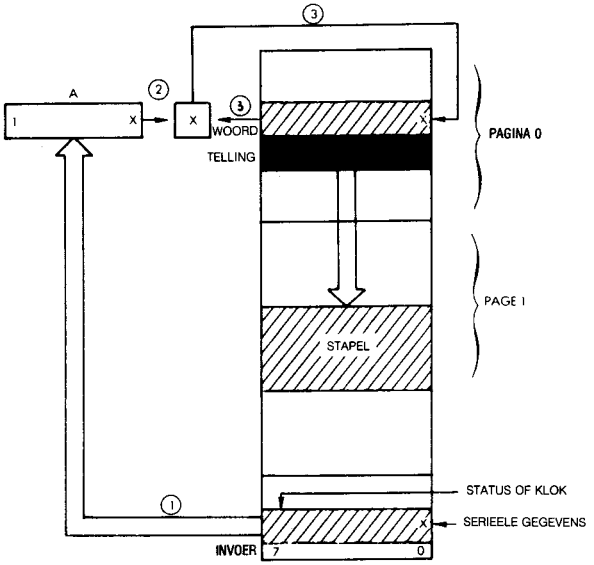


Fig. 6-6: Seriëlle tot parallelle overdracht

men. Dit wordt normaal *synchrone* transmissie genoemd. Of anders kunnen ze met onregelmatige intervallen komen. Dit wordt *asynchrone* transmissie genoemd. We zullen een programma ontwikkelen, dat in beide gevallen werkt. Het principe van het ontvangen van sequentiele data is eenvoudig: we kontroleren een inputlijn, zeg lijn 0. Als een databit op deze lijn gedetekteerd wordt, lezen we het bit in en schuiven het in een bufferregister. Steeds als er 8 bits verzameld zijn, bewaren we het byte data en verzamelen het volgende. Ter vereenvoudiging zullen we aannemen, dat het aantal bytes op voorhand bekend is. Anders moeten we bijvoorbeeld wachten op een speciaal break-karakter en de serieoverdracht op dat punt stoppen. We hebben al geleerd hoe we dit moeten doen. Het stroomdiagram van dit programma is gegeven in figuur 6-7. Het programma is als volgt:

```
SERIAL  LDA   #$00
        STA   WOORD
LOOP    LDA   INPUT   BIT 7 IS STATUS, "0" IS DATA
        BPL   LOOP    BIT ONTVANGEN?
        LSR   A       SCHUIF HET IN C
        ROL   WOORD   SCHUIF BIT IN GEHEUGEN
        BCC   LOOP    GA DOOR ALS CARRY = "0"
        LDA   WOORD
        PHA           BERG SAMENGESTELD BYTE OP
```

```
LDA   #$01      RESET BIT TELLER
STA   WOORD
DEC   TEL       VERLAAG WOORD TELLER
BNE   LOOP      STEL VOLGEND WOORD SAMEN
```

Dit programma is ontworpen om zo efficient mogelijk te zijn, en er zijn nieuwe technieken gebruikt, die we zullen verklaren. (Zie fig. 6-6.)

De afspraken zijn als volgt: geheugenplaats TEL bevat het aantal woorden dat overgedragen moet worden. In geheugenplaats WOORD zullen we de 8 inkomende bits assembleren tot een byte. Adres INPUT refereert aan een inputregister. We nemen aan dat bit 7 van dit register een statusvlag is, of een klokbit. Als het "0" is, is de data niet geldig.

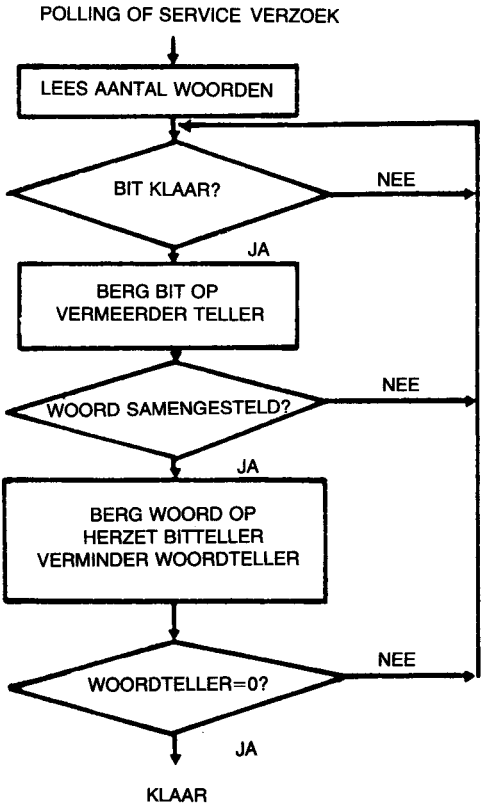


Fig. 6-7: Serie bit overdracht: stroomdiagram

Als het "1" is, is de data geldig. Aangenomen wordt, dat de data zelf binnenkomt op bit 0 van ditzelfde adres. In veel gevallen komt de data op een ander register binnen dan de status.
Het veranderen van dit programma zou een eenvoudige opgave moeten zijn. Verder nemen we aan dat het eerste bit dat dit programma moet ontvangen gegarandeerd een 1 is. Dit geeft aan, dat de echte gegevens binnenkomen. Als dit niet het geval zou zijn, zullen we later een voor de hand liggende verandering leren die hier iets aan doet. Het programma korrespondeert exakt met het stroomdiagram van fig. 6-7. De eerste regels van het programma voeren een wachtlus uit waarin getest wordt of een bit klaar is. Om te bepalen of een bit klaar is, lezen we het inputregister en testen het tekenbit (N). Zo lang dit bit "0" is, slaagt de instruktie BPL en springen we terug naar de loop. Zodra het status (of klok) bit waar wordt ("1"), faalt BPL en wordt de volgende instruktie uitgevoerd.

Denk er aan, dat BPL betekent: "Branch (spring) als Plus", d.w.z. als bit 7 (het tekenbit) "0" is. Deze beginserie komt overeen met pijl 1 in fig. 6-6.

Op dit punt bevat de accumulator een "1" in positie 7 en het eigenlijke databit in positie 0. Het eerste databit dat binnenkomt zal een "1" zijn. De volgende kunnen echter zowel "0" als "1" zijn. Nu willen we het databit bewaren dat in bitpositie 0 opgeslagen is. De instruktie:

```
LSR A
```

schuift de inhoud van de accumulator een positie naar rechts. Hierdoor valt het meest rechtse bit, dat ons databit is, in de carry. We zullen nu dit databit opbergen in geheugenplaats WOORD: (dit is geillustreerd met pijlen 2 en 3 in fig. 6-6)

```
ROL WOORD
```

Het gevolg van deze instruktie is, dat het carrybit in de meest rechtse positie van adres WOORD gelezen wordt. Tegelijkertijd komt het meest linkse bit van WOORD in het carrybit. (Als je twijfels hebt betreffende de roteer operatie, zie hoofdstuk 4!)

Het is belangrijk er aan te denken, dat een roteerbewerking zowel het carrybit in de meest rechtse positie bewaart, als het carrybit een nieuwe waarde geeft met bit 7.

Hier komt er een "0" in de carry. De volgende instruktie:

```
BCC LOOP
```

test de carry en springt terug naar adres LOOP zolang de carry "0" is.

Dit is onze automatische bitteller. Het is eenvoudig te zien, dat als gevolg van de eerst ROL, WOORD "00000001" zal bevatten. Acht verschuivingen later valt de "1" eindelijk in de carry en stopt het springen. Dit is een ingenieuze manier om een automatische loopteller uit te voeren, zonder een instruktie te hoeven verspillen om de inhoud van een indexregister te verminderen. Deze techniek wordt gebruikt om het programma korter te maken en de prestatie te verbeteren.

Steeds als BCC faalt, zijn er 8 bits geassembleerd in WOORD. Deze waarde moet in het geheugen bewaard worden. Dit gebeurt met de volgende instrukties: (pijl 4 in fig. 6-6)

```
LDA WOORD
PHA
```

We bewaren hier het WOORD data (8 bits) in de stapel. Dit kan alleen als er genoeg ruimte in de stapel is. Vooropgesteld, dat aan deze konditie voldaan is, is dit de snelste manier om een woord in het geheugen te bewaren. De stapelwijzer wordt automatisch bijgewerkt. Als we het woord niet in de stapel zouden pushen, zouden we een extra instruktie nodig hebben om een geheugenpointer bij te werken. We zouden ook geindexeerde adressering kunnen gebruiken, maar dan zouden we ook een indexregister moeten vermeerderen of verminderen, wat extra tijd kost.

Nadat het eerst WOORD gegevens opgeborgen is, is er geen garantie meer, dat het eerst binnenkomende databit een "1" zal zijn. Het kan van alles zijn. We moeten daarom de inhoud van WOORD terugzetten naar "00000001" zodat we deze als bitteller kunnen blijven gebruiken. Dit wordt gedaan met de volgende twee instrukties:

```
LDA #$01
STA WOORD
```

Tenslotte zullen we de woordteller verminderen aangezien er een woord geassembleerd is, en testen of we aan het einde van de overdracht zijn. Dit kan gebeuren met de volgende twee instrukties:

```
DEC TEL
BNE LOOP
```

Het bovenstaande programma is ontworpen op snelheid, zodat het een snelle invoerstroom van gegevens kan ontvangen. Zodra het programma beeindigd is, is het natuurlijk aan te bevelen om de woorden van de stapel meteen weg te schrijven naar een ander deel van het ge-

heugen. We hebben al geleerd een dergelijke blokoverdracht uit te voeren in hoofdstuk 5.

Oefening 6.5: Bereken de maximale snelheid waarmee dit programma seriele bits in kan lezen. Ga, om deze snelheid te berekenen, ervan uit dat de adressen WOORD en TEL op pagina nul staan. Je mag ook aannemen dat het hele programma in een pagina staat. Zoek het aantal cycli dat ieder instruktie nodig heeft op in de tabel aan het eind van dit boek, en bereken dan de tijd die verstrijkt tijdens het uitvoeren van dit programma. Om de tijdsduur van een loop te berekenen, moet het aantal keren dat de loop doorlopen wordt vermenigvuldigd worden met de tijdsduur in milliseconden om hem eenmaal te doorlopen. Verder mag je ervan uitgaan, dat iedere keer als de input gelezen wordt, er een databit aanwezig is.

Dit programma is moeilijker te begrijpen dan het voorgaande. Laten we dit nog eens in detail bekijken:

Af en toe komt er een databit binnen in positie "0" van INPUT. Er zouden bijvoorbeeld drie achtereenvolgende enen kunnen zijn. We moeten dus onderscheid maken tussen achtereenvolgens binnenkomende bits. Dit is de funktie van het "klok"-signaal.

Het klok- (of status-)signaal vertelt ons, dat het inputbit nu geldig is.

Voordat we een bit lezen, testen we daarom eerst het statusbit. Als de status "0" is, moeten we wachten. Bij een "1" is de data geldig.

We nemen hier aan, dat het statussignaal verbonden is met bit 7 van het INPUT register.

Oefening 6.6: Kun je uitleggen, waarom bit 7 voor de status gebruikt wordt, en bit 0 voor de data?

Als we eenmaal een databit hebben ontvangen, willen we dit op een veilige plaats opbergen, dan naar links schuiven zodat we het volgende bit kunnen pakken.

Helaas wordt de accumulator in dit programma zowel gebruikt om te lezen als om data en status te testen. Als we de gegevens in de accumulator zouden verzamelen, zou bitpositie 7 uitgewist worden door het statusbit.

Oefening 6.7: Heb je een suggestie om de status te testen,zonder de inhoud van de accumulator te wissen (een speciale instruktie)? Als dit

zou kunnen, zouden we dan de accumulator kunnen gebruiken om de binnenkomende bits te verzamelen?

Oefening 6.8: Herschrijf het programma, gebruik makend van de accumulator om binnenkomende bits op te slaan. Vergelijk de snelheid en het aantal instrukties met het voorgaande.

Laten we nog twee variaties noemen:
We hebben in ons bijzondere voorbeeld aangenomen, dat het eerste binnenkomende bit een speciaal signaal zou zijn, gegarandeerd een "1". In het algemene geval zou het echter van alles kunnen zijn.

Oefening 6.9: Modificeer het bovenstaande programma, ervan uitgaande dat het eerste binnenkomende bit geldige data is (die niet verwaarloosd mag worden), en "0" of "1" kan zijn. Hint: onze "bitteller" zou nog steeds korrekt moeten werken als je hem initialiseert met de juiste waarde.

Tenslotte hebben we het geassembleerde WOORD in de stapel opgeborgen om tijd te winnen. We zouden dit natuurlijk ook in een gespecificeerd geheugengebied kunnen opbergen:

Oefening 6.10: Verander het bovenstaande programma en berg het geassembleerde WOORD op in het geheugengebied dat begint bij BASE.

Oefening 6.11: Verander het bovenstaande programma, zodat de overdracht stopt zodra het karakter "S" in de invoerstroom gedetekteerd wordt.

Het hardware alternatief

Zoals gebruikelijk voor de meeste input/output algoritmen is het mogelijk om deze procedure in hardware uit te voeren. De chip wordt een UART genoemd. Hij verzamelt automatisch bits. Als men echter het aantal komponenten wil beperken zal dit programma of een variatie ervan gebruikt worden.

Oefening 6.12: Verander het programma, aannemende dat de data beschikbaar is op bitpositie 0 van locatie INPUT, terwijl de statusinformatie beschikbaar is op bitpositie 0 van adres INPUT + 1.

BASIS I/O SAMENVATTING

We hebben nu geleerd, hoe we elementaire input/output bewerkingen moeten verrichten en hoe we een stroom van parallelle of serie bits moeten behandelen. We zijn nu klaar om met echte input/output apparaten te kommuniceren.

KOMMUNIKATIE MET INVOER/UITVOER APPARATEN

Om gegevens uit te wisselen met invoer/uitvoer apparaten, moeten we ons er eerst van verzekeren, dat data beschikbaar is als we willen lezen, of dat het apparaat klaar is om data te ontvangen als we willen verzenden. Er kan gebruik gemaakt worden van twee procedures: handshaking en interrupts. Laten we eerst handshaking bestuderen.

Handshaking

Handshaking wordt algemeen toegepast tussen twee asynchrone apparaten, d.w.z. tussen twee apparaten die niet gesynchroniseerd zijn. Als we bijvoorbeeld een woord naar een parallelle printer willen sturen, moeten we ons er eerst van verzekeren, dat de invoerbuffer van de printer beschikbaar is. We vragen de printer dus: Ben je klaar? De printer zegt dan "ja" of "nee". Als hij nog niet klaar is, wachten we. Als hij klaar is, sturen we data. (zie fig. 6-8)

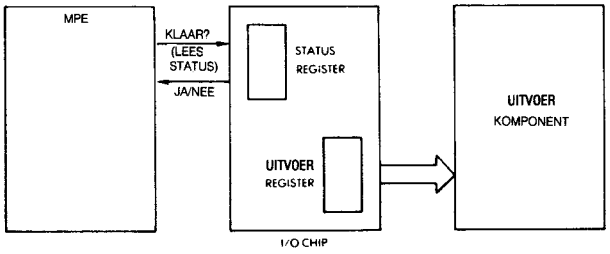


Fig. 6-8: Handshaking (Uitvoer)

Omgekeerd, voordat we data van een invoerapparaat lezen, zullen we verifieren of de data geldig is. We zullen vragen: "Is de data geldig?". En het apparaat zegt ons "ja" of "nee". Dit "ja of nee" kan aangegeven worden door een statusbit, of op een andere manier. (zie fig. 6-9)

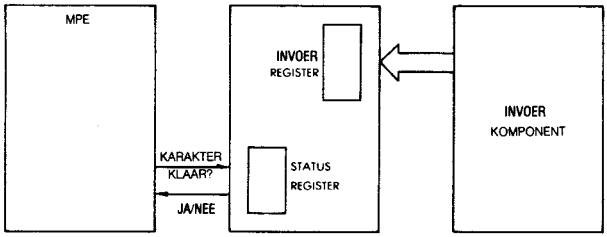


Fig. 6-9: Handshaking (Invoer)

Om kort te gaan: als je informatie uit wilt wisselen met iemand, die onafhankelijk is en op dat moment iets anders zou kunnen doen, moet je je ervan verzekeren dat hij klaar is om met je te kommuniceren. De gebruikelijke beleefdheidsregel is zijn hand te schudden (handshaking). Dan kan de uitwisseling van gegevens beginnen. Van deze procedure wordt gewoonlijk gebruik gemaakt bij de kommunikatie met invoer/uitvoer apparaten.

Laten we nu deze procedure eens illustreren met een eenvoudig voorbeeld:

Het versturen van een karakter naar de printer

We nemen aan, dat het karakter zich in geheugenplaats KAR bevindt. Het programma is als volgt:

```
KARPR   LDX KAR      LEES KARAKTER
WACHT   LDA STATUS   BIT 7 IS "KLAAR"
        BPL  WAIT
        TXA
        STA  PRINTD
```

Register X wordt eerst geladen met het te printen karakter. Dan testen we het statusbit om te kontroleren of de printer klaar is om het karakter te accepteren. Zolang deze echter niet klaar is om te printen, springen we terug naar WACHT en blijven in de loop. Zodra de printer klaar is om te printen, en dit aangeeft door het zetten van zijn readybit (klaarbit; hier bij afspraak bit 7 van adres STATUS), kunnen we het karakter versturen. We dragen het karakter over van register X naar register A:

```
        TXA
```

en we sturen het naar het uitvoerregister van de printer, hier PRINTD genaamd.

```
STA PRINTD
```

Oefening 6.13: Verander het bovenstaande programma om een string van n karakters te versturen, waarbij aangenomen wordt, dat n kleiner is dan 255.

Oefening: 6.14: Verander het bovenstaande programma om een string karakters te printen tot de kode voor een "carriage return" (wagen terug) tegengekomen wordt.

Laten we nu de outputprocedure iets ingewikkelder maken door een kodekonversie te eisen en tegelijkertijd naar meerdere komponenten te zenden:

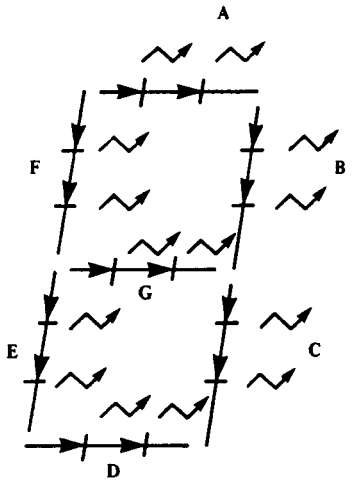


Fig. 6-10: Zeven segment LED

Uitvoer naar een 7-segment LED.

Een traditionele 7-segment licht-emiterende-diode (LED) kan de cijfers "0" to "9", of zelfs soms "0" tot "F" hexadecimaal weergeven door het oplichten van kombinaties van zijn 7 segmenten. Een 7-segment LED is getoond in figuur 6-10. De karakters die met deze LED gegenereerd kunnen worden zij te zien in figuur 6-11.

De segmenten van een LED zijn gelabelled a tot g in figuur 6-10.

Een "0" wordt bijvoorbeeld getoond (displayed) door de segmenten abcdef op te lichten. Laten we nu aannemen, dat bit "0" van een uitvoerpoort verbonden is met segment "a", dat bit "1" verbonden is met segment "b", enzovoort. Bit 7 wordt niet gebruikt. De binaire kode om fedcba op te lichten is dus "0111111". Hexadecimaal is dit "3F". Doe nu de volgende oefening.

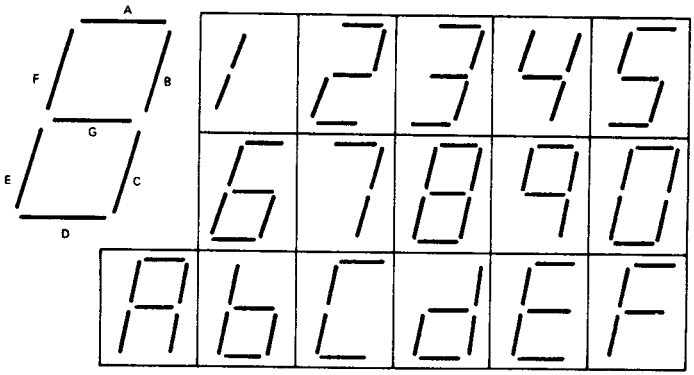


Fig. 6-11: Karakters gegenereerd met een 7-segment LED

Oefening 6.15: Bereken het 7-segment equivalent voor de hexadecimale cijfers "0" to "F". Vul de onderstaande tabel in:

Hex	LED code	Hex	LED code	Hex	LED code	Hex	LED code
0	3F	4		8		C	
1		5		9		D	
2		6		A		E	
3		7		B		F	

Laten we nu hexadecimale waarden tonen op *meerdere* LED's.

Het aansturen van meerdere LED's

Een LED heeft geen geheugen. Hij zal gegevens tonen zolang de segmentlijnen aktief zijn. Om de kosten van een LED-display laag te houden, zal de microprocessor de informatie om beurten naar de verschillende LED's sturen. De rotatie tussen de LED's moet snel genoeg zijn, zodat er geen duidelijke flikkering te zien is. Dit impliceert, dat de

tijd tussen het aansturen van de verschillende LED's minder dan 100 milliseconden moet zijn. Laten we een programma ontwerpen dat dit doet. Register zal gebruikt worden om naar de LED te wijzen die we willen aansturen. We nemen aan, dat de accumulator de hexadecimale waarde bevat die op de LED getoond moet worden. Onze eerste zorg is het omzetten van de hexadecimale waarde naar de 7-segment voorstelling. In het voorgaande deel hebben we de equivalentie tabel opgesteld. Omdat we een tabel gebruiken, zullen we de geindexeerde adresseringsmode gebruiken, waar de verplaatsingsindex gegeven wordt door de hexadecimale waarde. Dit betekent, dat de 7-segment-kode voor hexadecimaal cijfer #3 verkregen wordt door het derde element na de basis op te zoeken in de tabel. Het adres van de basis zullen we SEGBAS noemen. Het programma is als volgt:

```
LEDS    TAX              GEBRUIK HEXWAARDE ALS INDEX
        LDA  SEGBAS,X    LEES CODE IN A
        LDX  #$00
        STX  SEGDAT      ZET SEGMENTDRIVERS UIT
        STA  SEGDAT      GEEF CIJFER WEER
        LDX  #$70        EEN GROOT GETAL
        STY  SEGADR
VERTR   DEX
        BNE  VERTR
        DEY              WIJS NAAR VOLGENDE LED
        BNE  OUT
        LDY  LEDNBR
OUT     RTS
```

Het programma gaat ervan uit, dat register Y de waarde van de volgende aan te sturen LED bevat en dat register X het voor te stellende cijfer bevat.

Met de eerste twee instrukties zoekt het programma eerst de 7-segment kode op van de hexadecimale waarde in de accumulator. De volgende twee instrukties laden "00" als de waarde van de aan te sturen segmenten, d.w.z. schakelen ze uit. De volgende instruktie selekteert dan de juiste LED segmenten om aan te sturen: STA SEGDAT.

Er wordt dan een lus van drie instrukties uitgevoerd voordat naar de volgende LED overgeschakeld wordt. Tenslotte wordt de LED wijzer verminderd. (Had ook vermeerderd kunnen worden.)

Als de LED wijzer tot "0" verminderd is, moet hij weer herladen worden met het hoogste LED nummer. Dit wordt gedaan met de volgende twee instrukties. We nemen aan, dat dit een subroutine is en de laatste instruktie is een RTS: "keer terug van subroutine".

Oefening 6.16: Aannemende dat het bovenstaande programma een subroutine is, zul je opmerken dat registers X en Y intern gebruikt worden, en hun inhoud veranderd wordt. Aannemende, dat de subroutine vrijelijk gebruik mag maken van het geheugengebied aangewezen door de adressen T1, T2, T3, T4, T5, zou je dan aan het begin en het einde van de routine instrukties kunnen toevoegen zodat na het uitvoeren van de subroutine X en Y dezelfde waarde hebben als voor het uitvoeren van de subroutine?

Oefening 6.17: Dezelfde oefening als hierboven, maar ga er nu van uit, dat het geheugengebied T1, etc. niet voor de subroutine beschikbaar is. (Hint: denk eraan, dat in iedere computer een automatisch mechanisme ingebouwd is om informatie in een chronologische volgorde te bewaren.)

Teletype Invoer/Uitvoer

De teletype is een serie-apparaat. Het verzendt en ontvangt informatie in een serie formaat. Ieder karakter is gekodeerd in een 8-bit ASCII formaat (de ASCII tabel is achterin dit boek te vinden).

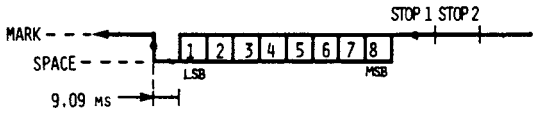


Fig. 6-12: Formaat van een Teletype woord

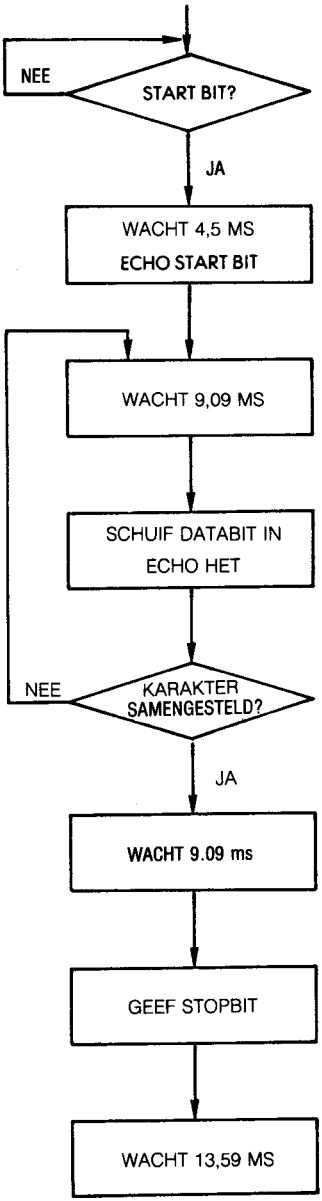


Fig. 6-13: TTY invoer met echo

Verder wordt ieder karakter voorafgegaan door een "startbit", en afgesloten met twee "stopbits". In het zogenaamde 20-milliamp(ere) current lus interface, wat het meest gebruikt wordt, is de toestand van de lijn normaal een "1". Dit is om de processor aan te geven, dat de lijn niet verbroken is. Een start is een "1" naar "0" overgang. Dit is een indicatie voor het ontvangende apparaat, dat de databits volgen. De standaard teletype is en 10 karakters per seconde apparaat. We hebben net vastgesteld, dat voor ieder karakter 11 bits nodig zijn. Dit betekent, dat de teletype 110 bits per seconde verstuurt. We zeggen, dat dit een 110 baud apparaat is. We zullen een programma ontwerpen on met de juiste snelheid bits in serie te verzenden naar de teletype.

Honderdentien bits per seconde impliceert, dat de bits gescheiden zijn door 9,09 seconden. Dit zal de lengte moeten zijn van de lus, die uitgevoerd wordt tussen de opeenvolgende bits. Het formaat van een teletype woord is gegeven in fig. 6-12. Het stroomdiagram voor bitinvoer is gegeven in fig. 6-13. Het programma volgt nu:

```
TTYIN  LDA   STATUS
       BPL   TTYIN    GEBRUIKELIJKE STATUSPOL
       JSR   VERTR    WACHT 9,09 MS
       LDA   TTYBIT   STARTBIT
       STA   TTYBIT   ECHO TERUG
       JSR   VERTR
       LDX   #$08     BITTELLER
NEXT   LDA   TTYBIT   BEWAAR INPUT
       LSR   A        BEWAAR BIT IN CARRY
       ADC   #$00     HERSTEL BIT
       ROL   KAR      BEWAAR BIT IN KAR
       STA   TTYBIT   ECHO TERUG
       JSR   VERTR
       DEX            VOLGENDE BIT
       BNE   NEXT
       LDA   TTYBIT   STOPBIT
       STA   TTYBIT   ECHO HET
       JSR   VERTR
       RTS
```

Fig. 6-14: Invoer van teletype

Merk op, dat het programma enigszins verschilt van het stroomdiagram van fig. 6-13. Het programma moet met aandacht bekeken wor-

den. De logika is erg eenvoudig. Het nieuwe feit is dat iedere keer als een bit van de teletype gelezen is, dit teruggeechoed wordt naar de teletype. Dit is een standaard kenmerk van de teletype. Als een gebruiker een toets indrukt, wordt de informatie naar de processor gestuurd en dan terug naar het printmechanisme van de teletype. Dit bevestigt, dat de transmissielijnen werken, en dat de processor werkt, als het karakter inderdaad korrekt op het papier afgedrukt wordt.

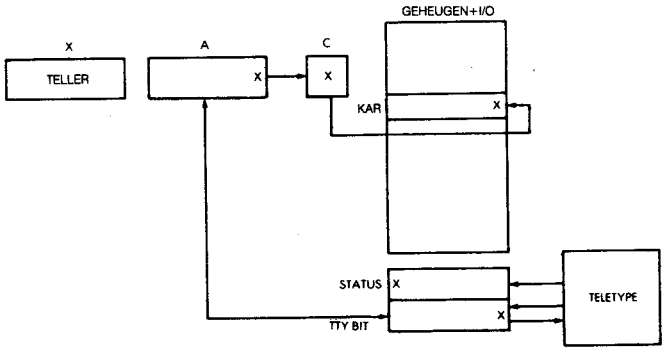


Fig. 6-15: Teletype invoer

De eerste twee instrukties zijn de wachtlus. Het programma wacht tot het statusbit waar wordt, voordat het begint bits in te lezen. Als gebruikelijk wordt aangenomen, dat het statusbit binnenkomt op bitpositie 7, omdat deze positie in een instruktie getest kan worden met BPL (Branch on plus - dit is het tekenbit).

JSR is een sprong naar een subroutine. We gebruiken een VERTR(aging) subroutine om de vertraging van 9,09 ms uit te voeren. Merk op, dat VERTR een vertragingslus kan zijn of de hardware timer kan gebruiken, als ons systeem er een heeft.

Het eerste bit dat binnenkomt is het startbit. Het moet wel naar de teletype geechoed worden, maar wordt verder genegeerd. Dit doen de instrukties 4 en 5.

Weer wachten we op het volgende bit. Maar nu is het een echt databit en moeten we het opbergen. Omdat alle schuifinstrukties een bit in de carryvlag laten vallen, hebben we twee instrukties nodig om ons databit te bewaren. (De "X" in fig. 6-15): een om het in C te laten vallen ("LSR A"), en een om het te bewaren in geheugenplaats "KAR" (ROL).

Wees beducht op een probleem: de "ROL" vernietigt de inhoud van C. Als we dit databit terug willen echoen, moeten we een voorzorgsmaatregel nemen om het op te bergen voordat het in "KAR" verdwijnt.

Tenslotte echoen we het databit: STA TTYBIT

En wachten op het volgende: JSR VERTR

tot we alle databits verzameld hebben: DEX.

Zodra we tot nul verminderd zijn, zijn alle 8 bits in KAR. We hoeven alleen nog maar de STOP bits te echoen en we zijn klaar.

Oefening 6.18: Schrijf een vertragingsroutine die een vertraging van 9,09 milliseconden geeft. (VERTR subroutine)

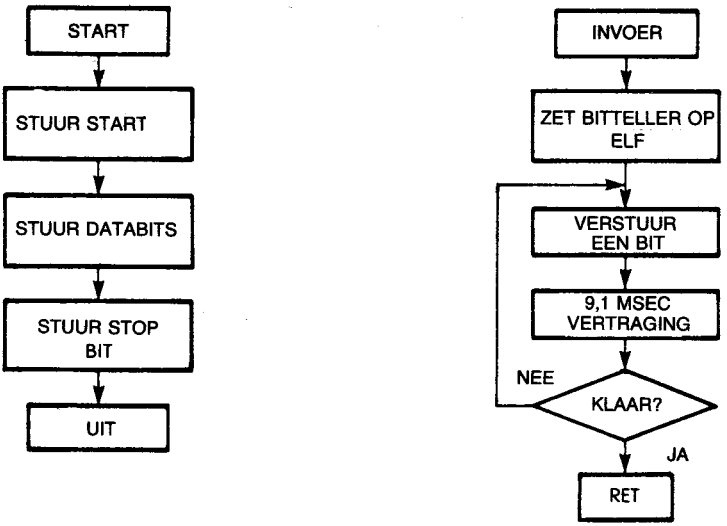


Fig. 6-16: Teletype uitvoer

Oefening 6.19: Gebruik het bovenstaande programma als voorbeeld en schrijf een PRINRK programma, dat op de teletype de inhoud van geheugenplaats KAR print.

Oefening 6.20: Modificeer het programma zodanig, dat het op een startbit wacht in plaats van op een statusbit.

Het printen van een string karakters

We zullen aannemen,dat de PRINRK routine (zie oefening 6.19) voor het printen van een karakter op onze printer, of het display, of een ander uitvoerapparaat, zorgt. We zullen hier de inhoud van geheugenplaatsen (START + N) tot (START) printen.

We zullen natuurlijk de geindexeerde adresseringsmode gebruiken. Het programma is eenvoudig:

```
PSTRING LDX   #N          AANTAL WOORDEN
NEXT    LDA   START,N
        JSR   PRINTK
        DEX
        BPL   NEXT
```

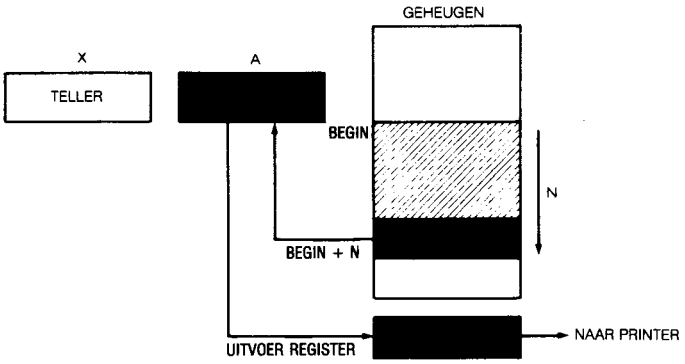


Fig. 6-17: Print een geheugenblok

SAMENVATTING RANDAPPARATUUR

We hebben nu de basis programmeertechnieken beschreven die gebruikt worden om met typische input/output apparaten te kommunice-

ren. Naast de overdracht van gegevens, zal het nodig zijn om bepaalde kontroleregisters in ieder I/O apparaat een waarde te geven, om zo op de juiste manier de overdrachtsnelheid, het interrupt mechanisme en eventuele andere opties te beinvloeden. Het handboek van ieder apparaat moet geraadpleegd worden. (Voor meer details betreffende de specifieke algoritmes om informatie uit te wisselen met alle gebruikelijke randapparaten, wordt de lezer verwezen naar ons boek C207, Microprocessor Interface Technieken.)
We hebben nu geleerd hoe we afzonderlijke apparaten moeten behandelen. In een echt systeem zijn alle randapparaten echter verbonden met de bussen, en kunnen gelijktijdig service aanvragen. Hoe gaan we de tijd van de processor plannen (schedulen)?

INVOER/UITVOER PLANNING

Omdat invoer/uitvoer verzoeken gelijktijdig op kunnen treden, moet er in een systeem een planningsmechanisme uitgevoerd zijn, dat bepaalt in welke volgorde service verleend wordt. Er wordt gebruik gemaakt van drie basis I/O technieken, die met elkaar gekombineerd

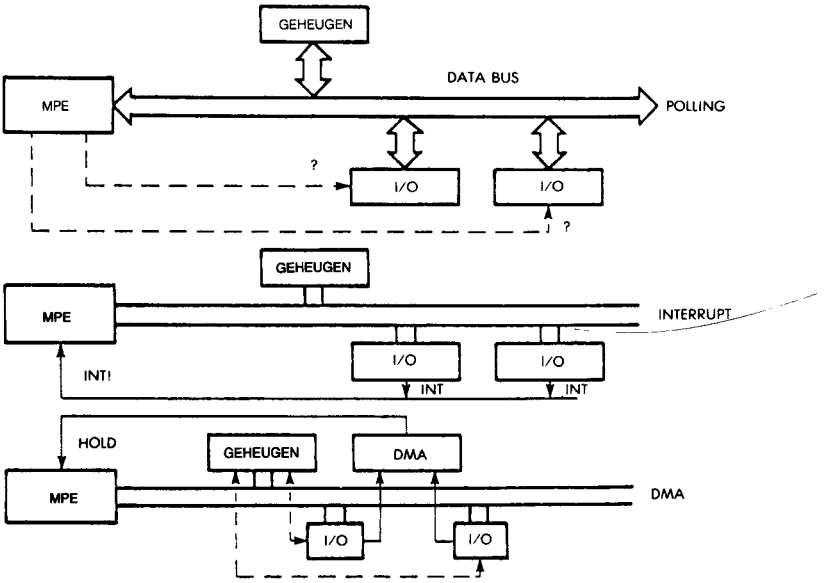


Fig. 6-18: Drie I/O kontrole methoden

kunnen worden. Dit zijn: polling, interrupt, DMA. Polling en interrupt worden hier beschreven. DMA is een zuivere hardware techniek en zal hier niet beschreven worden. (Het wordt behandeld in de naslagboeken C201 en C207.)

Polling

Konceptueel gezien, is polling de eenvoudigste methode om meedere randapparaten af te handelen. Volgens deze strategie ondervraagt de processor de met de bus verbonden apparaten na elkaar. Als een apparaat service wil, wordt service gegeven. Als het geen service wil, wordt het volgende apparaat onderzocht. Polling wordt niet alleen gebruik voor de apparaten, maar voor *iedere serviceroutine voor de apparaten.*

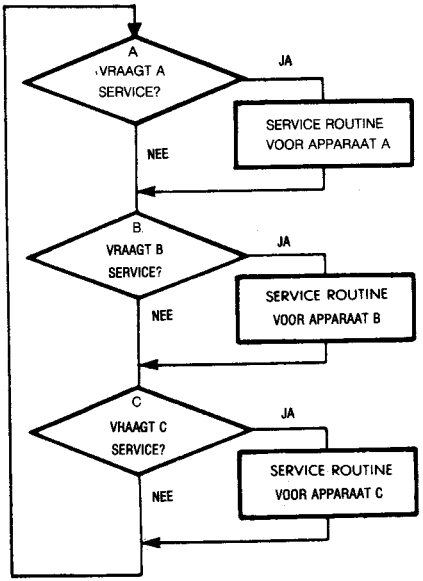


Fig. 6-19: Pollinglus stroomdiagram

Als een systeem bijvoorbeeld met een teletype, een bandrecorder en een beeldschermdisplay is uitgerust, zou de pollingroutine de teletype vragen: "heb je een karakter te verzenden?" Hij zou de *teletype outputroutine* vragen: "heb je een karakter te verzenden?" Vervolgens, aannemende dat de antwoorden negatief zijn, zou hij de bandrecorderroutines ondervragen en tenslotte het beeldschermdisplay.

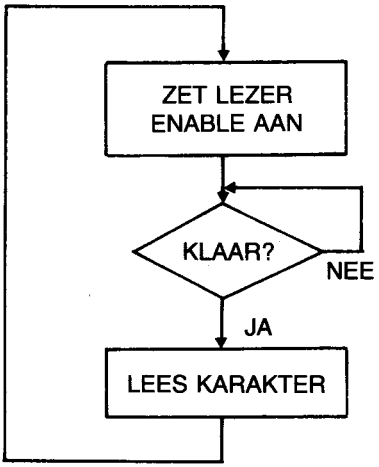


Fig. 6-20: Lezen van een ponsbandlezer

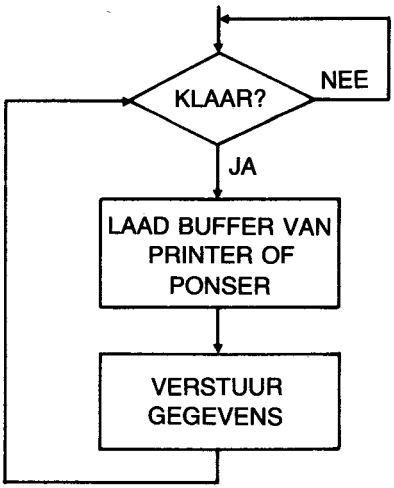


Fig. 6-21: Printen op een ponsband of gewone printer

Als er maar een apparaat met het systeem verbonden is, wordt polling ook gebruikt om te bepalen of het service nodig heeft. Als voorbeeld zijn de stroomdiagrammen om van een ponsbandlezer te lezen en om op een printer te printen gegeven in figuur 6-20 en figuur 6-21.

Voorbeeld: een pollingloop voor apparaten 1, 2, 3 en 4 (zie fig. 6-19):

```
POLL4 LDA  STATUS1  SERVICE VERZOEK IS BIT 7
      BMI  EEN
      LDA  STATUS2  APPARAAT2?
      BMI  TWEE
      LDA  STATUS3  APPARAAT3?
      BMI  DRIE
      LDA  STATUS4  APPARAAT4?
      BMI  VIER
      JMP  POLL4    TEST NOGMAALS
```

Bit 7 van het statusregister van ieder apparaat is "1" als het service wil. Als een verzoek gevonden wordt, springt het programma naar de afhandelroutine voor het apparaat, op adres EEN voor apparaat 1, TWEE voor apparaat 2, enzovoort.

Het voordeel van polling is duidelijk: het is eenvoudig, vereist geen hardware ondersteuning, en houdt alle invoer/uitvoer synchroon met het programma. Het nadeel is al net zo duidelijk: de meeste tijd van de processor gaat naar apparaten, die geen service nodig hebben. Verder zou de processor te laat service kunnen verlenen aan een apparaat, omdat hij zoveel tijd verspilt.

Het is dus wenselijk om een ander mechanisme te hebben, dat garandeert, dat de tijd van de processor gebruikt kan worden om nuttige berekeningen te verrichten, in plaats van onnodig steeds apparaten te pollen. We moeten echter benadrukken, dat polling vaak gebruikt wordt als een processor niets beters te doen heeft, omdat het de hele organisatie eenvoudig houdt. Laten we nu een belangrijk alternatief van polling eens onderzoeken: interrupts.

Interrupts

Het begrip interrupts is geillustreerd in fig. 6-18. Er is een speciale hardwarelijn beschikbaar, de interruptlijn, die verbonden is met een speciale pen van de microprocessor. Er kunnen meerdere input/output apparaten met deze interruptlijn verbonden zijn. Als een van hen service wil, stuurt het een niveau of een puls op deze lijn. Een interruptsig-

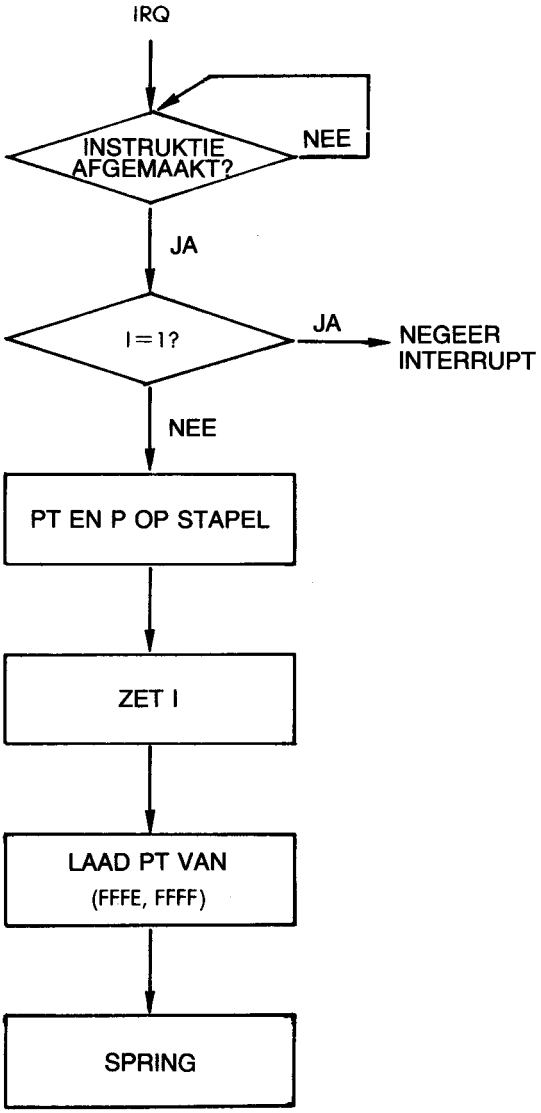


Fig. 6-22: Verwerken van interrupts

naal is een serviceverzoek van een invoer/uitvoer apparaat naar de processor.

Laten we eens kijken hoe de processor op deze interrupts reageert.

In ieder geval maakt de processor de instruktie waar hij mee bezig was af, anders zou dit een chaos veroorzaken in de microprocessor. Vervolgens moet de processor naar een interrupt afhandelroutine springen, die de interrupt verwerkt. Het springen naar een subroutine impliceert, dat de inhoud van de programmateller op de stapel opgeborgen moet worden. *Een interrupt moet daarom automatisch het opbergen van de programmateller op de stapel verzorgen.* Verder moet het statusregister ook automatisch opgeborgen worden, omdat de inhoud daarvan door iedere volgende instruktie veranderd wordt. Tenslotte moeten eventuele interne registers, die door de interruptroutine veranderd worden, ook op de stapel geplaatst worden.

Nadat al deze registers bewaard zijn, kan naar het adres gesprongen worden waar de interrupt afgehandeld wordt. Aan het einde van deze routine moeten alle registers weer hersteld worden en moet er een speciale interrupt return gegeven worden, zodat het hoofdprogramma weer met de uitvoering begint. Laten we nu de twee interruptlijnen van de 6502 eens in detail bekijken.

6502 Interrupts

De 6502 is met twee interruptlijnen uitgerust, IRQ en NMI. IRQ is de normale interruptlijn, terwijl NMI een niet-maskeerbare interruptlijn is met een hogere prioriteit. Laten we hun werking eens onderzoeken.

IRQ is de interrupt die met een niveau geaktiveerd wordt. De status van de IRQ lijn wordt door de processor al dan niet verwerkt, afhankelijk van de waarde van de interne vlag I (interrupt-masker vlag). We zullen in eerste instantie aannemen, dat interrupts wel doorkomen (geenabled zijn). Zodra IRQ geaktiveerd wordt, merkt de microprocessor dit. Zodra de interrupt geaccepteerd is (na beeindiging van de instruktie die net uitgevoerd wordt), wordt de interne I vlag automatisch gezet. Dit voorkomt, dat de microprocessor weer onderbroken wordt terwijl hij net met de interne registers manipuleert. De 6502 bewaart dan automatisch de inhoud van PC (de programateller) en P (het statusregister) in de stapel. De inhoud van de stapel, nadat een interrupt verwerkt is, is te zien in figuur 6-23.

Vervolgens haalt de 6502 automatisch de inhoud op van de geheugenplaatsen "FFFE" en "FFFF". Dit 16-bits geheugenadres zal de *in-*

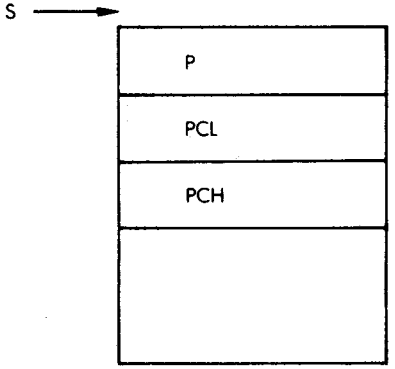


Fig. 6-23: 6502 stapel na een interrupt

terruptvektor bevatten. De 6502 zal de inhoud van dit adres halen en dan naar de gespecificeerde 16-bits vektor springen. De gebruiker moet dit 16-bits vektoradres op "FFFE"-"FFFF" opslaan. Er kunnen echter meerdere apparaten met de IRQ-lijn verbonden zijn. In dit geval springen we naar een enkele routine om de interrupt af te handelen. Hoe gaan we dan onderscheid maken tussen de verschillende apparaten? Dit zullen we in het volgende deel bestuderen.

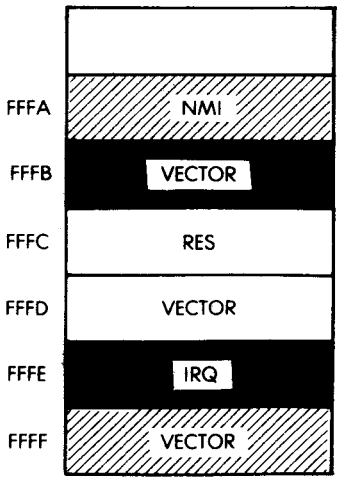


Fig. 6-24 Interrupt vektors

De NMI interrupt is in wezen gelijk aan de IRQ, met dit verschil dat hij niet door het I-bit gemaskeerd wordt. Het is een interrupt met een hogere prioriteit, meestal gebruikt voor spanningsstoringen. De werking is verder identiek, behalve dat de processor automatisch naar de inhoud van "FFFA"-"FFFB" springt. Dit is geillustreerd in fig. 6-24.

Het terugkeren van een interrupt gebeurt met een RTI. Deze instruktie draagt de bovenste drie woorden van de stapel, waarin P en PT (de 16-bits programmateller), weer over aan de microprocessor. Het onderbroken programma kan weer doorgaan. De interne toestand van de machine is precies gelijk aan die, zoals die was voordat de interrupt optrad. Het effekt is een vertraging in de uitvoering van het programma.

Aangenomen, dat de routine om de interrupts af te handelen, registers A, X en Y gebruikt, zijn er vijf instrukties nodig in deze afhandelroutine om deze registers te bewaren. Dit zijn:

```
SAVAXY  PHA     PUSH A IN STAPEL
        TXA     TRANSFEREER X NAAR A
        PHA     PUSH A
        TYA     TRANSFEREER Y NAAR A
        PHA     PUSH A
```

Helaas kan de 6502 alleen de inhoud van A en P direkt op de stapel pushen. Het gevolg is, dat het opbergen van X en Y tijd kost: er zijn vier instrukties voor nodig. Dit is geillustreerd in fig. 6-25.

Na het uitvoeren van de routine om de interrupts af te handelen, moeten deze registers weer hersteld worden en de afhandelaar van de interrupt moet eindigen met de serie van zes instrukties:

```
        PLA     PULL Y VAN DE STAPEL
        TAY     HERSTEL Y
        PLA     PULL X VAN DE STAPEL
        TAX     HERSTEL X
        PLA     HERSTEL A
        RTI     TERUG
```

Oefening 6.21: Maak gebruik van de tabel met het aantal cycli per instruktie om de tijd te berekenen, die nodig is om de registers A, X en Y te redden en te herstellen.

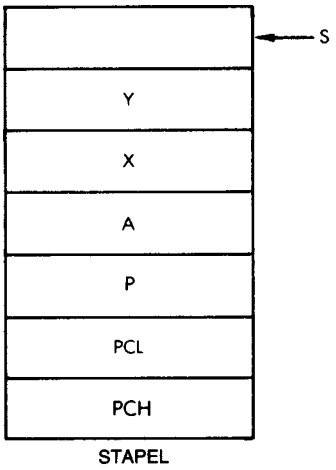


Fig. 6-25: Redden van alle registers

Bekijk voor een grafische vergelijking van het interruptproces en het pollingproces fig. 6-18, waar het pollingproces bovenin, en het interruptproces onderin geillustreerd is. Het is duidelijk, dat bij de pollingtechniek het programma veel tijd verspilt met wachten. Bij het gebruik van interrupts, wordt het programma onderbroken, de interrupt wordt afgehandeld, en het programma gaat dan weer verder. Een duidelijk nadeel van een interrupt is dat er enkele instrukties aan het begin en aan het einde geintroduceerd worden, die een vertraging tot gevolg hebben voordat de eerste instruktie van de afhandelaar van het apparaat uitgevoerd wordt. Dit is extra overhead.

Nu we de werking van de twee interruptlijnen verduidelijkt hebben, kunnen we twee belangrijke overblijvende problemen behandelen:

1 - Hoe lossen we het probleem op van meerdere apparaten, die op het zelfde ogenblik een interrupt plegen?

2 - Hoe lossen we het probleem op van de interrupt die net optreedt als een vorige afgehandeld wordt?

Meerdere apparaten verbonden met een interruptlijn

Als een interrupt optreedt, springt de processor automatisch naar het adres dat staat in FFFE-FFFF (voor een IRQ) of in FFFA-FFFB (voor een NMI). Voordat de eigenlijk verwerking kan beginnen, moet de afhandelende routine bepalen welk apparaat de interrupt genereerde. Hiervoor zijn twee methoden beschikbaar, zoals gebruikelijk een software en een hardware methode.

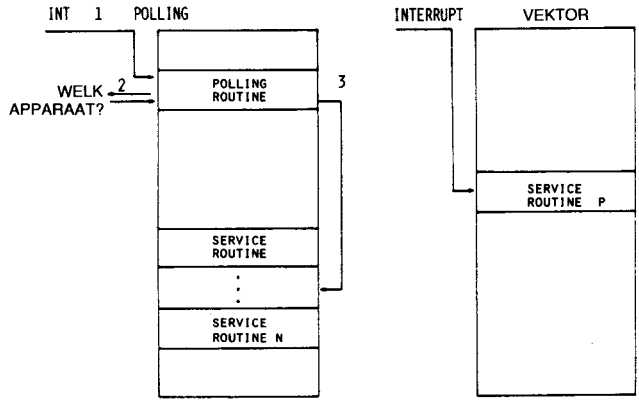


Fig. 6-26: Polling vs. vektorinterrupt

Bij de software methode wordt polling gebruikt: de microprocessor ondervraagt ieder apparaat om de beurt, en vraagt: "Heb jij een interrupt gegeven?". Zo niet, dan vraagt hij het volgende apparaat. Dit proces is geillustreerd in fig. 6-26. Een deel van het programma is:

```
LDA   STATUS1
BMI   EEN
LDA   STATUS2
BMI   TWEE
```

De hardware methode gebruikt extra komponenten, maar geeft tegelijkertijd met de interrupt het adres van het apparaat, dat de interrupt gaf. De komponent, die nu algemeen gebruikt wordt om in deze faciliteit te voorzien, wordt een "PIC" genoemd, of "priority-interrupt-controller". Een dergelijke PIC plaatst automatisch het spronga-

dres voor het apparaat, dat de interrupt gaf, op de databus. Als de 6502 naar FFFE-FFFF gaat, haalt hij dit vektoradres. Het begrip is geillustreerd in fig. 6-26.

In de meeste gevallen, als de snelheid waarmee op een interrupt wordt gereageerd niet erg kritisch is, wordt de pollingmethode gebruikt. Als de responstijd de voornaamste overweging is, moet de hardwaremethode gebruikt worden.

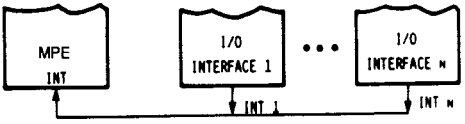


Fig. 6-27: Meerdere apparaten kunnen dezelfde interruptlijn gebruiken.

Meervoudige interrupts

Het volgende probleem dat op kan treden is dat er een nieuwe interrupt gegeven wordt tijdens de afhandeling van een vorige interrupt. Laten we eens kijken wat er gebeurt, en hoe de stapel gebruikt wordt om dit probleem op te lossen. We hebben in hoofdstuk 2 al aangegeven dat dit een andere wezenlijke funktie van de stapel is, en nu is het ogenblik gekomen om dit gebruik te demonstreren. We zullen fig. 6-28 gebruiken om meerdere interrupts te illustreren. De tijd verstrijkt van links naar rechts in de illustratie. De inhoud van de stapel is onder in de illustratie getoond. Als we links kijken, op tijdstip T0, wordt programma P uitgevoerd. Op tijdstip T1 treedt interrupt I1 op. We nemen aan dat het interrupt masker ge-enabled was, zodat I1 toegestaan is. Programma P wordt tijdelijk onderbroken. Dit is onderin de figuur te zien. De stapel bevat minstens de programmateller en statusregister P, plus eventuele andere registers die door de interruptbehandelaar of I1 zelf gered zijn.

Op T1 begint de uitvoering van interrupt I1, tot tijdstip T2. Op tijdstip T2 treedt interrupt I2 op. We nemen aan dat interrupt I2 een hogere prioriteit heeft dan interrupt I1. Als hij een lagere prioriteit gehad zou hebben, zou hij genegeerd zijn tot I1 afgehandeld zou zijn. Op tijdstip T2 worden de registers van I1 in de stapel geplaatst; dit is onder in de illustratie te zien. Weer wordt de inhoud van de programmateller en het statusregister op de stapel geplaatst. Verder kan routine voor I2 besluiten om enkele andere registers te redden. I2 wordt nu tot het ein-

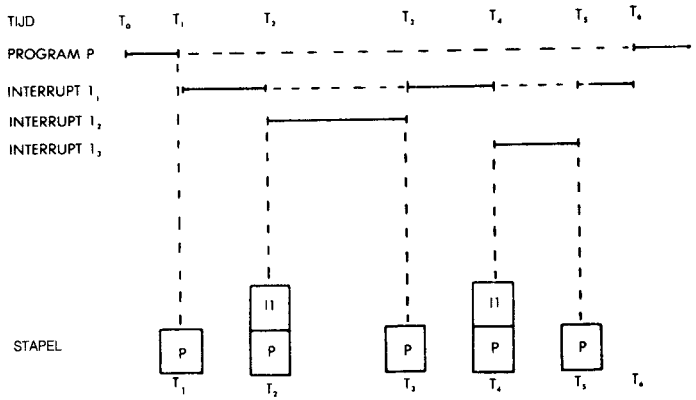


Fig. 6-28: Stapel tijdens interrupts

de toe uitgevoerd, op tijdstip T3. Als I2 klaar is, wordt de inhoud van de stapel automatisch weer teruggezet in de 6502, en dit is onder in fig. 6-28 getoond. Automatisch wordt nu interrupt I1 weer uitgevoerd. Helaas treedt er op tijdstip weer een interrupt I3 met een hogere prioriteit op. We kunnen onder in de figuur zien dat de registers van I1 weer op de stapel geplaatst worden. Interrupt T3 wordt van T4 tot T5 uitgevoerd, en eindigt op T5. Op dat ogenblik wordt de inhoud van de stapel weer in de 6502 gezet, en wordt I1 weer uitgevoerd. Nu wordt deze tot het einde toe uitgevoerd, en is klaar op T6. Op T6 worden de overblijvende registers die gered waren weer in de 6502 gezet en kan programma P weer uitgevoerd worden. De lezer kan verifieren, dat de stapel nu leeg is. In feite geeft het aantal streepjeslijnen, die de onderbroken programma's aangeven, tegelijkertijd aan hoeveel niveau's er in de stapel zijn.

Oefening 6.22: Als we aannemen, dat iedere keer als er een interrupt optreedt, de programmateller PT , register P en de accumulator gered worden, kost dit minstens vier plaatsen. (In de praktijk zouden X en Y ook gered kunnen worden, zodat er zes plaatsen nodig zijn.) Aannemende dat er maar drie registers in de stapel gered worden, hoeveel interruptniveau's staat de 6502 dan toe? (Denk er aan, dat de stapel beperkt is tot 256 plaatsen in pagina 1.)

Oefening 6.23: Neem aan dat nu vijf registers in de stapel gered worden. Wat is dan nu het aantal interrupts dat simultaan behandeld kan

worden? Is er een faktor die het aantal simultane interrupts nog verder beperkt?

Benadrukt moet echter worden, dat in de praktijk microprocessorsystemen maar met een klein aantal apparaten verbonden zijn die interrupts afgeven. Het is daarom onwaarschijnlijk dat een groot aantal interrupts gelijktijdig in zo'n systeem optreedt.

We hebben nu alle problemen opgelost die normaal verbonden zijn met interrupts. Hun gebruik is in feite eenvoudig en ze zouden eigenlijk zelfs de beginnende programmeur tot voordeel moeten strekken. Laten we onze analyse van de 6502 bronnen kompleteren door nog een instruktie te introduceren, wiens effekt identiek is met een synchrone interrupt:

BREAK

Het BRK kommando in de 6502 is identiek aan een software interrupt. Hij kan in een programma geplaatst worden en resulteert net als de IRQ in de automatische opslag van PT en P, en een indirekte sprong naar FFFE-FFFF. Deze instruktie kan met voordeel gebruikt worden om geprogrammeerde interrupts te genereren tijdens het verbeteren (debugging) van een programma. Dit heeft een breekpunt tot gevolg, dat het programma op een vastgestelde plaats onderbreekt, en vervolgens naar een routine springt die de gebruiker in staat stelt het programma te analyseren. Aangezien het netto effekt van een break en een interrupt hetzelfde is, moet er voorzien zijn in een middel voor de programmeur om te bepalen of het een break of een interrupt was. De 6502 zet de B-vlag in register P (in de stapel gered) op "1" als het een break was en op "0" als het een interrupt was. Het testen van de status van dit bit kan met het volgende programma:

1 - Hoe lossen we het probleem op van meerdere apparaten, die op het zelfde ogenblik een interrupt plegen?
2 - Hoe lossen we het probleem op van de interrupt die net optreedt als een vorige afgehandeld wordt?

```
BTEST PLA            LEES TOP VAN STAPEL IN A
      PHA            SCHRIJF TERUG
      AND  #$10      MASKEER B-BIT
      BNE  BRKPRG    GA NAAR BREAK-
                     PROGRAMMA
```

Dit testprogramma wordt normaal tussengevoegd in de pollingserie die bepaalt welk apparaat een interrupt gaf.

Opgelet: Een kenmerk van de break is dat hij automatisch de inhoud van de programmateller *plus 2* bewaart. Omdat de break maar 1 byte lang is, moet de programmeur soms de inhoud van de programmateller in de stapel aanpassen door te verminderen of te vermeerderen, om zo op het juiste adres weer met de uitvoering verder te gaan. In het bijzonder wordt de break vaak gebruikt tijdens het verbeteren, door hem over een andere instruktie heen te schrijven in het programma. Als het programma voor de uitvoering weer geassembleerd wordt, moet de inhoud van de programmateller die gered is, meestal met 1 verminderd worden.

SAMENVATTING

We hebben in dit hoofdstuk de technieken gepresenteerd, die gebruikt worden om met de buitenwereld te kommuniceren. Beginnend bij elementaire invoer/uitvoer routines tot meer ingewikkelde programma's om met echte randapparaten te kommuniceren, hebben we geleerd om alle gebruikelijke programma's te ontwikkelen, en hebben zelfs de efficientie van benchmark programma's onderzocht voor parallelle overdracht en parallel-serie konversie. Tenslotte hebben we geleerd, hoe we meerdere apparaten moeten schedulen door gebruik te maken van polling en interrupts. Er kunnen natuurlijk nog veel andere exotische I/O apparaten met het systeem verbonden zijn. Met de gepresenteerde technieken en een goed begrip van de betreffende apparaten moet het mogelijk zijn om de meeste problemen op te lossen.

In het volgende hoofdstuk zullen we de karakteristieken onderzoeken van de invoer/uitvoer interface chips, die doorgaans met de 6502 verbonden zijn. Vervolgens zullen we de basis gegevensstrukturen bekijken, waar de programmeur gebruik van zou kunnen maken.

OEFENINGEN

Oefening 6.24: Een 7-segment display kan ook andere cijfers weergeven dan het hexadecimale alfabet. Bereken de kodes voor: H,I,J,L,O,P,S,U,Y,g,h,i,j,l,n,o,p,r,t,u,y.

Oefening 6.25: Het stroomdiagram voor het afhandelen van interrupts is gegeven in onderstaande figuur 6-29. Beantwoord de volgende vragen:

a - Wat gebeurt hardware en wat software?
b - Waar wordt het masker voor gebruikt?
c - Hoeveel registers moet gered worden?
d - Hoe wordt het onderbrekende apparaat geidentificeerd?
e - Wat doet de RTI instruktie? Wat is het verschil met een subroutine return?
f - Geef een suggestie voor wat we moeten doen als de stapel overstroomt.
g - Wat is de overhead ("verloren tijd") die door het interruptmechanisme geintroduceerd wordt?

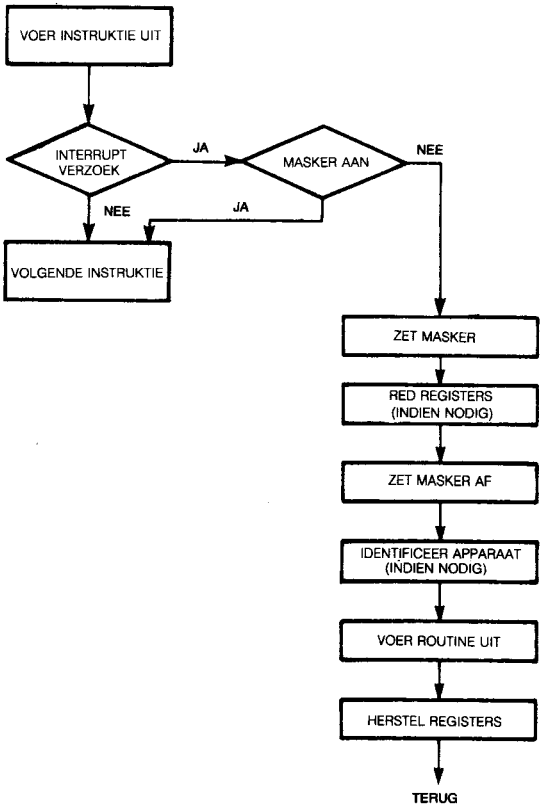


Fig. 6-29: Interrupt logika

7
INVOER/UITVOER KOMPONENTEN

INLEIDING

We hebben nu geleerd hoe we de 6502 in de meeste gebruikelijke omstandigheden moeten programmeren. We moeten echter de invoer/uitvoer chips die normaal met de microprocessor verbonden zijn nog noemen. Vanwege de vooruitgang in LSI integratie zijn er nieuwe chips geintroduceerd, die er daarvoor nog niet waren. Het gevolg is, dat voor het programmeren van een systeem niet alleen de microprocessor zelf, maar ook de *invoer/uitvoer chips geprogrammeerd moeten worden.* Het is zelfs vaak moeilijker om te onthouden hoe de verschillende kontrole opties van de I/O chip geprogrammeerd moeten worden dan het programmeren van de microprocessor zelf! Dit is niet omdat het programmeren zelf zo moeilijk is, maar omdat iedere chip zijn eigenaardigheden heeft. We zullen hier eerst de meest algemene invoer/uitvoer komponent onderzoeken, de programmeerbare invoer/uitvoer chip (kortweg "PIO"), en dan "verbeteringen" van deze standaard PIO, die nu vaak gebruikt worden met de 6502: de 6520, 6530, 6522 en de 6532.

De standaard PIO (6502)

Er is geen "standaard" PIO. De 6520 is echter, wat zijn funktie betreft, in essentie analoog aan alle gelijkwaardige PIO's die door andere fabrikanten voor hetzelfde doel gemaakt worden. Het doel van een PIO is te voorzien in een meervoudige poort voor invoer/uitvoer appara-

ten. (Een "poort" is gewoon een set van 8 I/O lijnen.) Iedere PIO voorziet tenminste in twee sets 8-bits lijnen voor I/O apparaten. Iedere I/O komponent heeft een *databuffer* nodig om de inhoud van de databus tenminste voor output te stabiliseren. Onze PIO zal daarom met een minimum van een buffer per poort uitgerust zijn.

Verder hebben we vastgesteld, dat de microprocessor een *handshake* procedure zal gebruiken, of anders interrupts, om met het randapparaat te kommuniceren. Iedere PIO moet daarom met tenminste *twee kontrolelijnen per poort* uitgerust zijn om de handshakefunktie uit te voeren.

Iedere microprocessor moet ook de status van iedere poort kunnen lezen. Iedere poort moet dus uitgerust zijn met een of meer *statusbits*. Tenslotte zal iedere PIO nog enkele opties hebben om zijn bronnen te configureren. De programmeur moet toegang hebben tot een speciaal register in de PIO om de programmeeropties te specificeren. Dit is het *kontrole-register*. Bij de 6520 is de statusinformatie een deel van het kontrole-register.

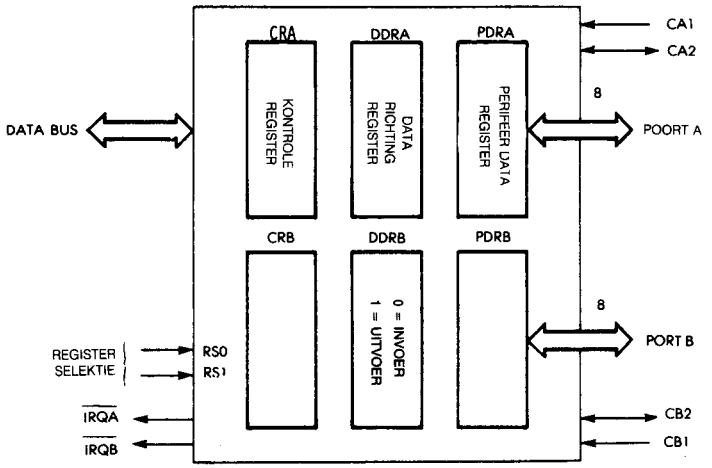


Fig. 7-1: Een typische PIO

Een essentiele voorziening van de PIO is dat iedere lijn als een inputlijn of als een outputlijn geschakeld kan worden. Het diagram van een PIO is te zien in fig. 7-1. De programmeur kan specificeren of een lijn input of output zal zijn. Om de richting van de lijnen te programmeren is er een *datarichting-register* voor iedere poort. Een "0" in een bitposi-

tie van het datarichting-register specificeert een invoer. Een "1" specificeert een uitvoer.

Het kan verrassend zijn om te zien dat een "0" voor invoer gebruikt wordt en een "1" voor uitvoer, terwijl eigenlijk een "0" met Uitvoer zou moeten korresponderen en een "1" met Invoer. Dit is met opzet gedaan: zodra de spanning van het systeem aangezet wordt, is het van groot belang dat alle I/O lijnen als *invoer* geschakeld zijn. Anders zou, als de microcomputer met een gevaarlijk randapparaat verbonden zou zijn, dit per ongeluk geaktiveerd kunnen worden. Als er een reset gegeven wordt, worden alle registers nul gemaakt en dit heeft tot gevolg, dat alle lijnen van de PIO als invoer geschakeld worden. De verbinding met de microprocessor is links in de figuur te zien. De PIO is natuurlijk verbonden met de databus, de adresbus en de kontrolebus van de microprocessor. De programmeur hoeft alleen maar het adres te specificeren waar hij binnen de PIO toegang tot wil hebben. De 6520, die uitwisselbaar is met de 6820 van Motorola, heeft een eigenaardigheid ervan geërfd: hij is uitgerust met zes interne registers. Er kunnen echter maar *vier* registers gespecificeerd worden! Dit probleem wordt opgelost door bitpositie 2 van het kontroleregister te schakelen. Als dit bit "0" is, kan het bijbehorende datarichting-register geselekteerd worden. Als het "1" is, kan het dataregister geselekteerd worden. Als de programmeur dus gegevens in het datarichting-register wil schrijven, moet hij zich er eerst van verzekeren dat bit 2 van het betreffende kontroleregister nul is, voordat hij dit register kan selekteren. Dit is

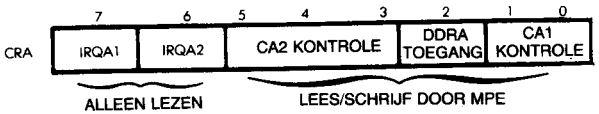


Fig. 7-2: PIO kontrolewoord formaat

RS1	RS0	CRA 2	CRB 2	GESELEKTEERD REGISTER
0	0	1	–	PERIFEER REGISTER
0	0	0	–	DATA RICHTING REGISTER
0	1	–	–	KONTROLE REGISTER
1	0	–	1	PERIPHERAL REGISTER B
1	0	–	0	DATA DIRECTION REGISTER B
1	1	–	–	CONTROL REGISTER B

Fig. 7-3: Adresseren van PIO registers

een beetje lastig te programmeren, maar het is belangrijk er aan te denken, om pijnlijke moeilijkheden te vermijden.

Om het effekt van de adresselektie van de 6520 te verduidelijken, is hierboven de adresselektie tabel gegeven. RS0 en RS1 zijn twee register-selektie signalen die van de databus worden betrokken. Met andere woorden: ze representeren twee bits van het door de programmeur te specificeren adres. CRA is het kontroleregister voor poort A. CRA(2) is bit 2 van dit register. CRB is het kontroleregister voor poort B.

Het interne kontroleregister

Het kontroleregister van de 6520 specificeert zoals we gezien hebben in bitpositie 2 een selektiemode voor de interne registers van de poort. Verder voorziet het in een aantal opties voor het genereren of detekteren van interrupts, of voor het automatisch uitvoeren van een handshakefunktie. De volledige beschrijving van de faciliteiten is hier niet nodig. De gebruiker van een systeem hoeft alleen maar de datasheet (blad met gegevens) van de 6520 te na te slaan voor de stand van de verschillende bits van het kontroleregister. Zodra het systeem opgestart wordt, moet de programmeur het kontroleregister van de 6520 met de juiste inhoud laden voor de verwachte toepassing.

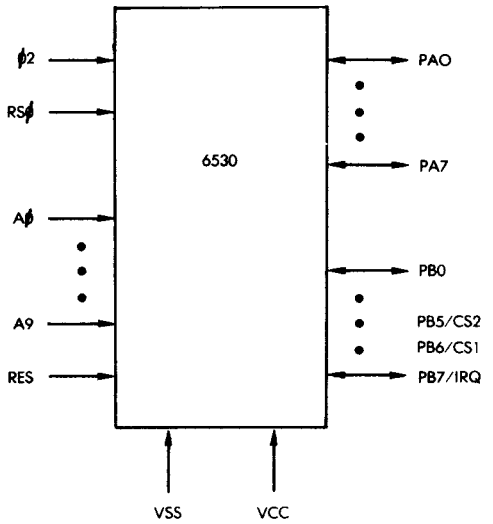


Fig. 7-4: Gebruik van een PIO: Laden van het kontroleregister

De 6530

In de 6530 is een kombinatie van vier funkties uitgevoerd: RAM, ROM, PIO en TIMER. Het RAM is een 64×8 geheugen. Het ROM is een 1K×8 geheugen. De timer geeft de programmeur meerdere intervaltiming faciliteiten. Het PIO gedeelte is in wezen analoog aan de 6520 die we beschreven hebben: Er zijn twee poorten; iedere poort heeft een dataregister en een datarichting-register, plus een kontroleregister. Een "0" in een bitpositie van het richtingregister specificeert input, terwijl een "1" output specificeert.

De programmeerbare intervaltimer kan geprogrammeerd worden om tot 256 intervallen te tellen (hij heeft intern 8 bits). De programmeur kan de tijdsperiode specificeren als zijnde 1, 8, 64 of 1024 maal de systeemklok. Als de maximale telling bereikt is wordt de interruptvlag van de chip op een logische "1" gezet. De inhoud van de timer wordt via de databus gezet. De vier mogelijke tijdsintervallen moeten op de lijnen A0 en A1 van de adresbus gespecificeerd worden.

Drie pennen van poort B hebben een dubbele funktie: PB5, PB6 en PB7 kunnen gebruikt worden voor kontrole-funkties. Pen PB7 kan bijvoorbeeld als interruptinput geprogrammeerd worden.

De chip wordt in het bijzonder op het KIM bord gebruikt. (N.B.: op de KIM is pen PB6 niet beschikbaar.)

Het programmeren van een PIO

Als voorbeeld is hier een programma om de 6520 of de 6522 te gebruiken. (We nemen aan dat het kontroleregister al gezet is.)

```
LDA   #FF    ZET DATARICHTING
STA   DDRB   KONFIGUREER B VOOR
             OUTPUT
LDA   #00
STA   IORB   GENEREER NUL OUTPUT
```

DDRB is het adres van het datarichting-register van poort B voor deze PIO. IORB is I/O register of Dataregister voor poort B, "FF" hexadecimaal is binair "11111111" = alle uitgangen.

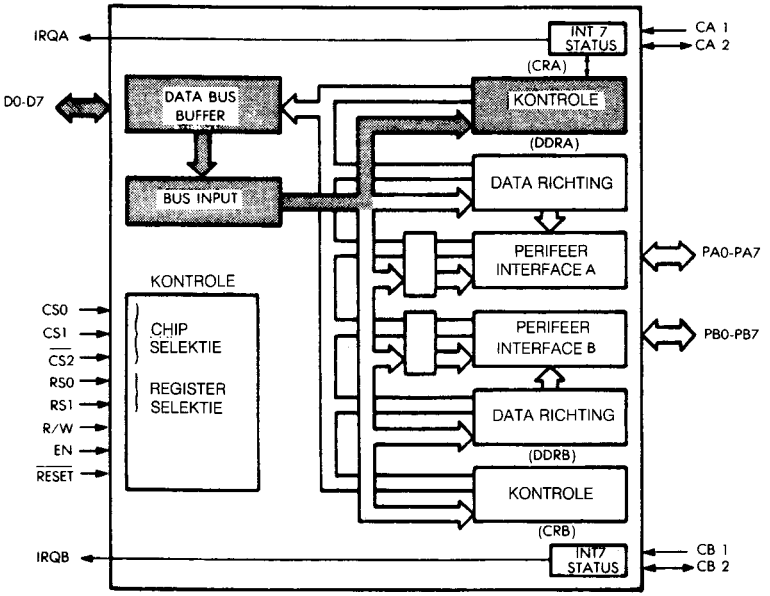


Fig. 7-5: Gebruik van een PIO: Laad kontroleregister

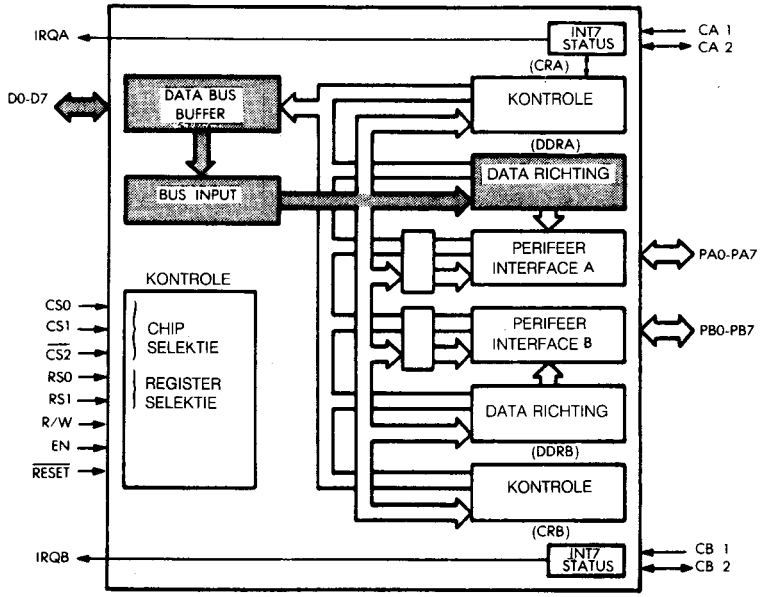


Fig. 7-6: Gebruik van een PIO: Laad datarichting

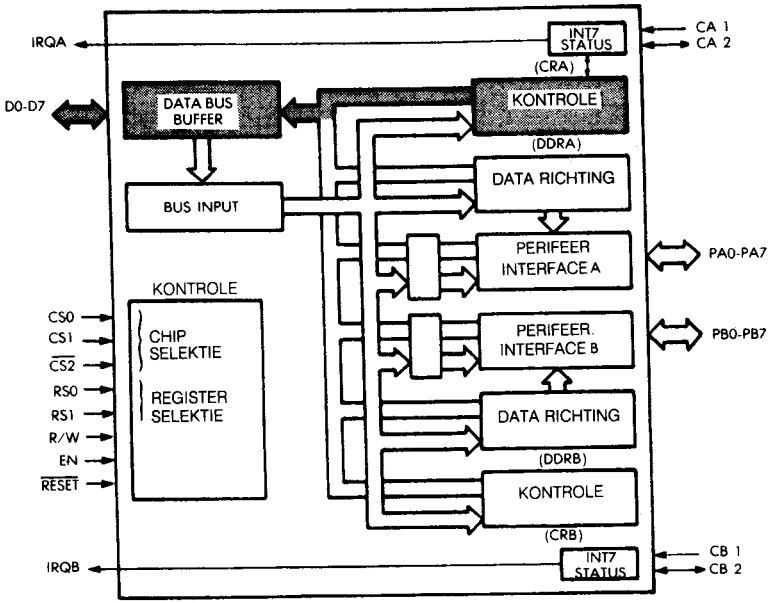


Fig. 7-7: Gebruik van een PIO: Lees status

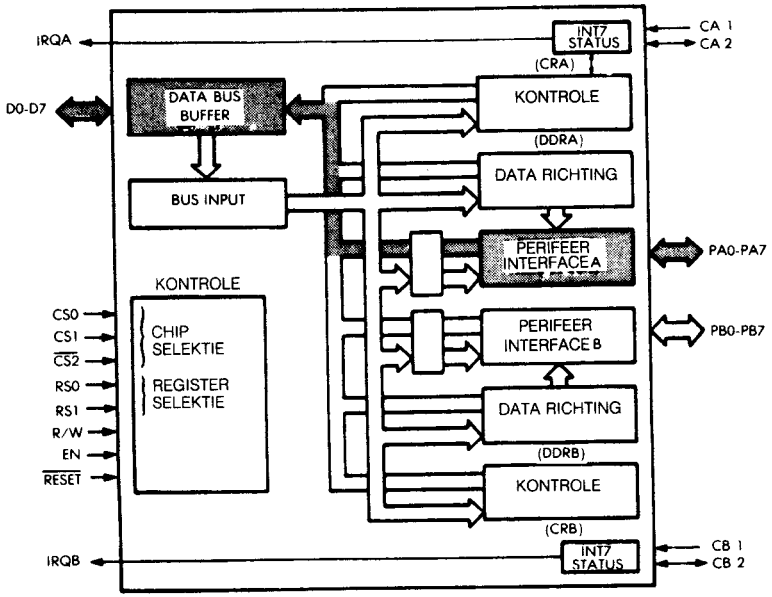


Fig. 7-8: Gebruik van een PIO: Lees invoer

De 6522

De 6522, ook wel genaamd "versatile interface adapter" (veelzijdige interface aanpasser), of "VIA", is een verbeterde versie van de 6520. Naast de kapaciteiten van de 6520 voorziet hij in twee programmeerbare intervaltimers, een parallel-serie en serie-parallel konverter, plus input datageheugens. Een gedetailleerde beschrijving van deze komponent gaat buiten het bestek van dit boek. Het zou, samen met de voorgaande beschrijvingen, voor de programmeur niet al te moeilijk moeten zijn om vertrouwd te raken met het adresseren van de interne registers van deze komponent en de programmering ervan. Deze informatie wordt verstrekt in de datasheets van de fabrikant.

De 6532

De 6532 is een kombinatiechip waarop 28×8 RAM, een PIO met twee twee-richtingspoorten, en een programmeerbare intervaltimer. Hij wordt gebruikt op het SYM bord, vervaardigd door Synertek Systems, dat identiek is met het KIM bord dat door MOS Technology en Rockwell vervaardigd wordt. Ook hier moet de lezer weer zorgvuldig de dokumentatie (= datasheets) onderzoeken om zich vertrouwd te maken met het adresseren en het gebruik van de verschillende interne registers.

SAMENVATTING

Om doelmatig gebruik te maken van dergelijke komponenten, moet helaas de funktie van ieder bit of groep bits in de verschillende kontroleregisters in detail begrepen worden. Deze ingewikkelde chips automatiseren een aantal procedures, die vroeger door software of speciale logika uitgevoerd werden. In het bijzonder zijn een groot deel van de handshake procedures in komponenten als de 6522 geautomatiseerd. Ook kan een deel van het afhandelen en detekteren van interrupts intern zijn. Met de informatie die in het voorgaande hoofdstuk gepresenteerd is, zou de lezer in staat moeten zijn om de funktie van de verschillende signalen en registers te begrijpen, met behulp van de dokumentatie. Natuurlijk zullen er nog meer ingewikkelde komponenten geintroduceerd worden, die een hardware uitvoering bieden van steeds ingewikkelder algoritmes. Ook nu weer, zou de lezer in staat moeten zijn om ze te bestuderen aan de hand van de dokumentatie van de fabrikant.

8
TOEPASSINGSVOORBEELDEN

INLEIDING

Dit hoofdstuk is ontworpen om je nieuwe programmeervaardigheid te testen met behulp van een aantal utiliteits programma's. Deze programma's of "routines" komen vaak voor in toepassingen en worden in het algemeen "utiliteits routines" (eng.: utilities) genoemd.

Ze vereisen een samengaan van de tot nu toe gepresenteerde kennis en technieken.

We zullen karakters van een I/O-apparaat halen en ze op verschillende manieren verwerken. Maar laten we eerst een deel van het geheugen op nul zetten (dit is niet altijd nodig, ieder van deze programma's wordt alleen gepresenteerd als een programmeervoorbeeld).

EEN DEEL VAN HET GEHEUGEN OP NUL ZETTEN

We willen de inhoud van het geheugen van adres BASE + 1 tot BASE + LENGTE nul maken, waarbij LENGTE minder dan 256 is.

Het programma is:

```
ZEROM   LDX   LENGTE
        LDA   #0
CLEAR   STA   BASE, X
```

```
        DEX
        BNE   CLEAR
        RTS
```

Merk op dat X als index gebruikt wordt om te wijzen naar de huidige plaats in het geheugen die nul gemaakt wordt.

De accumulator wordt maar een keer geladen met de waarde 0 (allemaal nullen) en vervolgens weggeschreven naar opeenvolgende geheugenplaatsen:
BASE + LENGTE, BASE + LENGTE − 1, etc. tot X tot 0 verminderd is. Als X nul is, keert het programma terug.

In een geheugentest bijvoorbeeld, zou dit programma gebruikt kunnen worden om het geheugen te op nul te zetten (uit te wissen), waarna de inhoud gekontroleerd kan worden.

Oefening 8.1: Schrijf een programma dat een blok van 256 woorden nul maakt en kontroleer of iedere geheugenplaats 0 is. Vervolgens schrijft het allemaal enen en kontroleert de inhoud van het blok. Daarna schrijft het overal 01010101 en verifieert de inhoud, en tenslotte schrijft het 10101010 en verifieert de inhoud.

We gaan nu onze I/O-apparaten pollen om uit te vinden welke service nodig heeft:

HET POLLEN VAN I/O APPARATEN

We zullen aannemen, dat 3 I/O apparaten met ons systeem verbonden zijn. Hun statusregisters zijn geplaatst op adressen IOSTATUS1, IOSTATUS2 en IOSTATUS3.

Als het statusbit in positie 7 is, lezen we het statusregister en testen het tekenbit. Als het statusbit ergens anders is, kunnen we met voordeel gebruik maken van de BIT instruktie van de 6502:

```
TEST    LDA   MASKER
        BIT   IOSTATUS1
        BNE   GEVONDEN1
        BIT   IOSTATUS2
```

```
        BNE   GEVONDEN2
        BIT   IOSTATUS3
        BNE   GEVONDEN3
        (fout uitgang)
```

Het MASKER zal bijvoorbeeld "10000000" bevatten als we bitpositie 7 testen. Het gevolg van de BIT-instruktie is, dat het Z bit van de statusvlaggen op 1 gezet wordt als "MASKER EN IOSTATUS" ongelijk aan nul is, d.w.z. als het korresponderende bit van IOSTATUS gelijk is aan dat van MASKER. De BNE instruktie (branch als ongelijk nul) heeft dan een sprong naar de betreffende GEVONDEN routine tot gevolg.

INLEZEN VAN KARAKTERS

Neem aan, dat we net gevonden hebben dat er een karakter klaar is bij het toetsenbord. We gaan nu karakters verzamelen in een geheugengebied genaamd buffer tot we een speciaal karakter genaamd SPC tegenkomen waarvan de kode eerder al gedefinieerd is.

De subroutine GETCHAR haalt een karakter van het toetsenbord (zie hoofdstuk 6 voor meer details) en laat het in de accumulator achter. We nemen aan dat er maximaal 256 karakters gehaald worden voor we een SPC tegenkomen.

```
STRING  LDX #0          INITIALISEER INDEX OP
                        NUL
NEXT    JSR GETCHAR
        CMP #SPC        IS DIT HET BREAK KARAK-
                        TER?
        BEQ UIT         ZO JA, KLAAR
        STA BUFFER,X    NEE, RED KARAKTER
        INX             VERMEERDER POINTER
        JMP NEXT
UIT     RTS
```

Oefening 8.2: Laten we deze basis routine verbeteren:

a - Echo het karakter terug naar het apparaat (voor een Teletype bijvoorbeeld)

b - Kontroleer of de inputstring inderdaad niet langer is dan 256 karakters

We hebben nu een karakterstring in een geheugenbuffer. Laten we die nu op verschillende manieren bewerken.

TESTEN VAN EEN KARAKTER

Laten we bepalen of het karakter in geheugenplaats LOC gelijk is aan een 0, 1 of 2:

```
ZOT     LDA     LOC
        CMP     #$00
        BEQ     NUL
        CMP     #$01
        BEQ     EEN
        CMP     #$02
        BEQ     TWEE
        JMP     NIETGEV
```

We lezen gewoon het karakter en gebruiken dan de CMP instruktie om de waarde te testen.

We gaan nu een andere test uitvoeren:

KATEGORIETEST

Laten we bepalen of het ASCII karakter in geheugenplaats LOC een cijfer tussen 0 en 9 is:

```
HAAKJE  LDA     #$40
        ADC     #$40        FORCEER OVERFLOW
        LDA     LOC
        ORA     #$80        ZET BIT 7=1
        CMP     #$B0        ASCII 0
        BCC     TELAAG
        CMP     #$BA        ASCII 9
        BEQ     UIT         PRECIES 9
        BCS     TEHOOG
UIT     CLC                 ZET C OP NUL
        CLV
        RTS
TELAAG  SEC                 ZET C OP EEN
```

```
        CLV
        RTS
TEHOOG  RTS             (C IS EEN)
```

ASCII 0 wordt hexadecimaal gerepresenteerd door "B0"
ASCII 9 wordt hexadecimaal gerepresenteerd door "BA"

Denk er aan, dat bij gebruik van een CMP instruktie het carrybit gezet wordt als de waarde van de konstante die volgt kleiner is dan of gelijk aan de accumulator. Indien groter wordt hij "0" gemaakt (gereset).

Als B0 groter is dan ons karakter, is ons karakter te laag en volgt er een sprong.

Dan vergelijken we het met BA (9). Als het kleiner of gelijk is aan 9 is alles OK,en verlaten we het programma. Anders gaan we naar TEHOOG.

Als we dit programma verlaten, willen we weten of het getal TEHOOG, TELAAG of anders tussen 0 en 9 is.

Dit wordt aangegeven door de vlaggen C en V. V wordt niet veranderd door CMP. Z, N en C worden wel beinvloed door CMP.

Als we van deze subroutine terugkeren, geeft een "0" in V "te hoog" aan, een "1" in C geeft "te laag" aan, en "0" in C geeft een juist cijfer tussen 0 en 9 aan.

Er zouden natuurlijk andere afspraken gemaakt kunnen worden, zoals het laden van een cijfer in de accumulator, om het resultaat van de test aan te geven.

Oefening 8.3: Vereenvoudig het bovenstaande programma door tegen het ASCII karakter dat op "9" volgt te testen, in plaats van tegen precies 9.

Oefening 8.4: Bepaal of het ASCII karakter in de accumulator een letter van het alfabet is.

Als je een ASCII tabel gebruikt, zul je merken, dat bijna altijd pari-

teit gebruikt wordt. De ASCII kode voor "0" is "0110000", een 7-bits kode. Als we echter bijvoorbeeld oneven pariteit gebruiken (we garanderen dat het totale aantal enen in een woord oneven is), wordt de kode "10110000". Er wordt links en extra "1" toegevoegd. Dit is "B0" hexadecimaal. Laten we daarom een programma ontwikkelen om pariteit te genereren.

PARITEIT GENERATIE

Dit programma genereert een even pariteit in bitpositie 7:

```
PARITY LDX  #$00     BITTELLING
       LDA  #$00
       STA  EENTEL   TELLING VAN DE ENEN
       LDA  KAR      LEES KARAKTER
       ROL  A        NEGEER BIT 7
NEXT   ROL  A        VOLGENDE BIT
       BCC  NUL      IS HET EEN 1?
EEN    INC  EENTEL
NUL    DEX           VERMINDER BITTELLING
       BNE  NEXT     LAATSTE BIT?
       ROL  A        HERSTEL BIT 0
       ROL  A        BIT NEGEREN?
       LSR  EENTEL   MEEST RECHTSE BIT IS
                     PARITEIT
       LSR  A        PLAATS IN A
       RTS
```

Register X wordt gebruikt om bits te tellen terwijl ze links uit de accumulator geschoven worden. Iedere keer als er links een "1" uit de accumulator wordt geschoven (dit wordt getest door BCC), wordt de eenteller vermeerderd. Als er 8 bits verschoven zijn (het programma negeert bit 7 dat het pariteits bit zal worden), wordt A nog tweemaal naar links geschoven, zodat bit 6 links van A is.

Het pariteitsbit is het meest rechtse bit van EENTEL: het wordt geinstalleerd in het carrybit door LSR en wordt bit 7 van A. Een volgende LSR kopieert dit bit weer terug in bitpositie 7 van A en we zijn klaar.

Oefening 8.5: Verifieer de pariteit van een woord, gebruik makend van

het bovenstaande programma. Je moet de juiste pariteit berekenen en vervolgens vergelijken met de pariteit van het woord.

KODEKONVERSIE: ASCII NAAR BCD

Het konverteren van ASCII naar BCD is erg simpel. We kunnen zien dat de hexadecimale representatie van ASCII karakters 0 tot 9 is: B0 tot B9.

De BCD representatie wordt verkregen door gewoon de "B" te laten vallen, d.w.z. het linker nibble (vier bits) weg te maskeren:

```
LDA     KAR
AND     #$OF        MASKEER LINKER NIBBLE
                    WEG
STA     BCDKAR
```

Oefening 8.6: Schrijf een programma om BCD naar ASCII te konverteren.

Oefening 8.7: (moeilijker) Schrijf een programma om BCD naar binair te konverteren.

Hint: $N_3\,N_2\,N_1\,N_0$ in BCD is $(((N_3 \times 10) + N_2) \times 10 + N_1) \times 10 + N_0$ in binair.

Gebruik om met 10 te vermenigvuldigen: naar links schuiven ($=\times 2$), nogmaals naar links schuiven ($=\times 4$), een ADC ($=\times 5$), nogmaals naar links schuiven ($=\times 10$).

In full-BCD notatie kan het eerste nibble het aantal BCD cijfers bevatten, de volgende nibble het teken, en iedere volgende nibble bevat een BCD cijfer (we nemen aan dat er geen decimale komma is). De laatste nibble van een blok kan ongebruikt zijn.

ZOEK HET GROOTSTE ELEMENT VAN EEN TABEL

Het beginadres van de tabel is opgeslagen in geheugenplaats BASE in pagina 0. De eerste ingang van de tabel bevat het aantal elementen van de tabel. Dit programma zoekt het grootste element van de tabel. De waarde wordt achtergelaten in A, en de plaats wordt opgeslagen in geheugenplaats INDEX.

Dit programma gebruikt registers A en Y, en maakt gebruik van indirekte adressering, zodat het ieder tabel waar dan ook in het geheugen kan afzoeken.

Dit programma test de n-de ingang eerst. Als die groter is dan 0 komt deze in A, en de plaats wordt onthouden in INDEX. Dan wordt de (n−1)de ingang getest, enz.

```
MAX       LDY   #0          TABELINDEX
          LDA   (BASE), Y   IN ENTRY 0 STAAT LENGTE
          TAY               OPSLAAN IN Y
          LDA   #0          INITIALISEER MAXIMALE WAARDE
                            OP NUL
          STA   INDEX       INITIALISEER INDEX OP NUL
LOOP      CMP   (BASE), Y   IS HUIDIGE MAXIMALE ELEMENT?
          BCS   GEENVERW    JA?
          LDA   (BASE), Y   LAAD NIEUWE MAXIMUM
          STY   INDEX       PLAATS VAN MAXIMUM
GEENVERW  DEY               WIJS NAAR VOLGENDE ELEMENT
          BNE   LOOP        BLIJF TESTEN
          RTS               KLAAR ALS Y=0
```

Dit programma werkt voor positieve gehele getallen

Oefening 8.8: Verander het programma zodanig, dat het ook werkt voor negatieve getallen in 2-komplement.

Oefening 8.9: Werkt dit programma ook voor ASCII karakters?

Oefening 8.10: Schrijf een programma dat n getallen in afnemende grootte sorteert.

Oefening 8.11: Schrijf een programma dat n namen (van ieder 3 letters) in alfabetische volgorde sorteert.

DE SOM VAN N ELEMENTEN

Dit programma berekent de 16-bit som van n ingangen (entries) in een tabel. Het startadres van de tabel staat in geheugenadres BASE, in pagina nul. De eerste ingang vande tabel bevat het aantal elementen N. De 16-bit som wordt achtergelaten in geheugenplaatsen SOMLA en SOMHO. Mochten er voor de som meer dan 16 bits nodig zijn, dan worden alleen de laagste 16 bewaart (we zeggen dat de hogere orde bits afgekapt worden).

Dit programma verandert de registers A en Y. Het gaat uit van een maximum van 255 elementen.

```
        LDA     #0          INITIALIZE SUM
        STA     SUMLO       INITIALIZE SUM
        STA     SUMHI       INITIALIZE SUM
        TAY                 INITIALIZE Y TO ZERO
        LDA     (BASE), Y   GET N
        TAY                 INTO Y
        CLC                 CLEAR CARRY FOR ADC
ADLOOP  LDA     (BASE), Y   GET NEXT ELEMENT
        ADC     SUMLO       ADD IT TO SUMLO
        STA     SUMLO       SAVE RESULT
        BCC     NOCARRY     CARRY?
        INC     SUMHI       ADD IT TO SUMHI
        CLC                 FOR NEXT SUM
NOCARRY DEY                 NEXT ELEMENT
        BNE     ADLOOP      AGAIN IF Y NOT ZERO
        RTS
```

Dit programma is eenvoudig en spreekt voor zichzelf.

Oefening 8.12: Verander het programma om:
a - een 24 bit som te berekenen
b - een 32 bit som te berekenen
c - een overflow te detekteren

EEN KONTROLESOM BEREKENING

Een kontrolesom (checksum) is een cijfer of een aantal cijfers dat berekend is aan de hand van een blok achtereenvolgende karakters. De kontrolesom wordt berekend als de gegevens opgeslagen worden en aan het einde geplaatst. Om te kontroleren of de gegevens juist zijn, worden ze gelezen, de kontrolesom wordt berekend en vergeleken met de opgeslagen waarde. Een verschil geeft een vergissing of een fout aan.

Er worden verschillende algoritmes gebruikt. Hier zullen we de exclusive OR van alle bytes in een tabel van N elementen nemen en dit resultaat in de accumulator achterlaten. Zoals gebruikelijk is het begin van de tabel opgeslagen in BASE in pagina nul. De eerste ingang van

de tabel is het aantal elementen N. Het programma verandert A en Y. N moet kleiner zijn dan 256.

```
CHECKSUM LDY     #0          POINT TO FIRST ENTRY
         LDA     (BASE), Y   GET N
         TAY                 STORE IT IN Y
         LDA     #0          INITIALIZE CHECKSUM
CHLOOP   EOR     (ADDR), Y   EOR NEXT ENTRY
         DEY                 POINT TO NEXT
         BNE     CHLOOP      KEEP GOING
         RTS
```

TELLEN VAN NULLEN

Dit programma telt het aantal nullen in onze gebruikelijke tabel, en laat het resultaat in A achter.
Het verandert A, X, Y:

```
ZEROES   LDY     #0          POINT TO FIRST ENTRY
         LDA     (ADDR), Y   GET N
         TAY                 STORE IT IN Y
         LDX     #0          INITIALIZE NO. OF ZEROES
ZLOOP    LDA     (ADDR), Y   GET NEXT ENTRY
         BNE     NOTZ        IS IT ZERO?
         INX                 YES. COUNT IT
NOTZ     DEY                 POINT TO NEXT
         BNE     ZLOOP       KEEP GOING
         RTS
```

Oefening 8.13: Verander het programma zodanig, dat geteld worden:
a - het aantal sterren (het karakter "*")
b - het aantal letters van het alfabet
c - het aantal cijfers tussen 0 en 9

EEN STRING AFZOEKEN

Een string karakters is in het geheugen opgeslagen zoals aangegeven in fig. 8-1. We zoeken nu deze string af om te kijken of er een kortere string, een "template" genaamd (TEMPLT) met lengte TPTLEN, in voorkomt. De lengte van de oorspronkelijke string is STRLEN, en het

programma komt terug met in register X de plaats waar de TEMPLT gevonden werd, of anders FF hexadecimaal. Het stroomdiagram voor dit programma is gegeven in fig. 8-2. We zoeken in de string eerst naar het eerste karakter van TEMPLT. Als dit eerste karakter niet gevonden wordt, keert het programma terug met een foutindikatie. Als dit eerste karakter wel gevonden wordt, vergelijken we het tweede karakter van TEMPLT met het volgende karakter in de string. Als deze karakters verschillend zijn, wordt het zoeken opnieuw gestart voor het eerste karakter, omdat dit karakter vaker in de string kan voorkomen. Als het eerste en het tweede karakter overeenkomen gaat het zoeken op dezelfde manier verder met de volgende karakters van TEMPLT. Het programma is getoond in fig. 8-3. Merk op dat register X gebruikt wordt als variabele wijzer die tijdens het zoeken naar het huidige element van de string wijst. Er wordt natuurlijk gebruik gemaakt van geindexeerde adressering om toegang tot het huidige element van de string te krijgen.

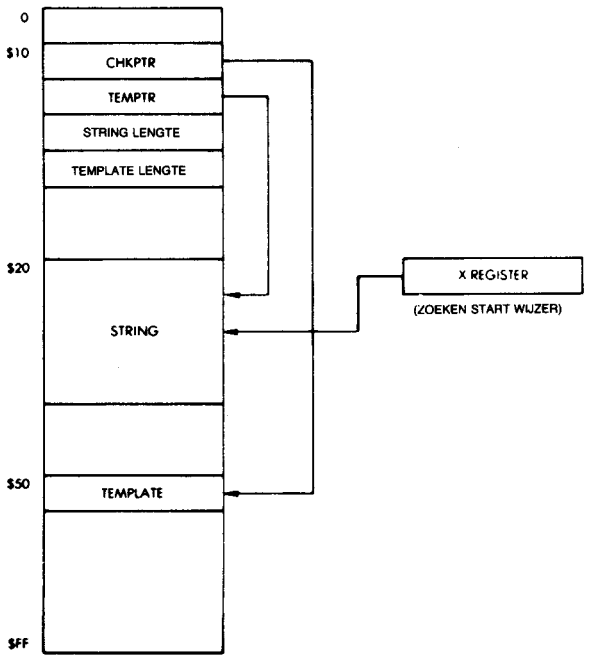


Fig. 8-1: String zoeken: het geheugen

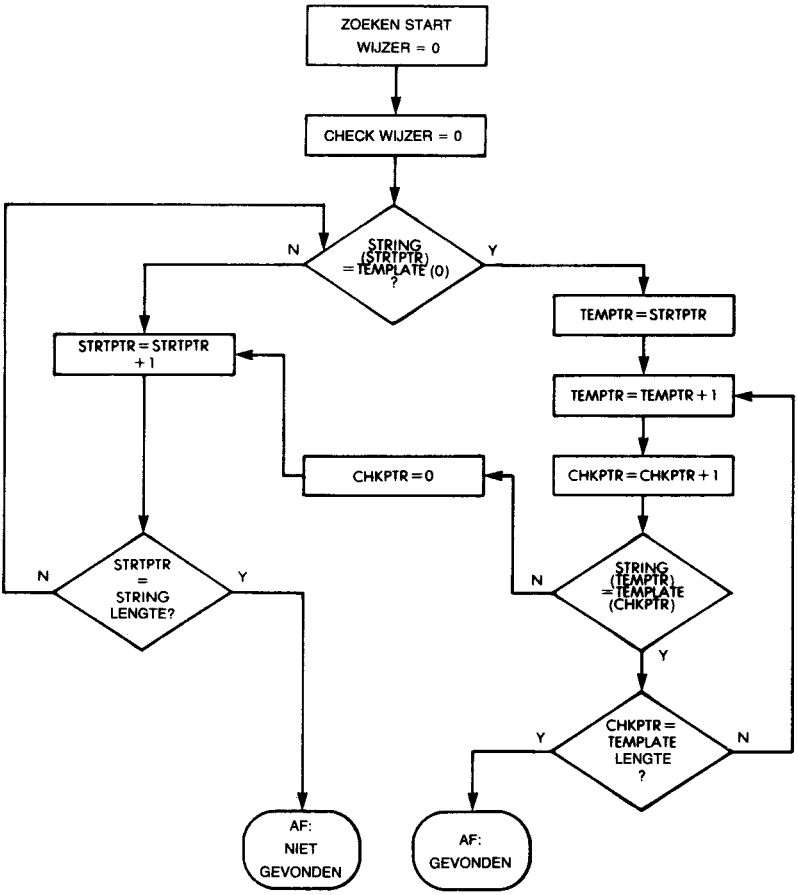


Fig. 8-2: String zoeken: stroomdiagram

```
LINE # LOC     CODE       LINE

0002  0000                ;STRING SEARCH.
0003  0000                ;FINDS LOCATION IN STRING OF LENGTH 'STRLEN'
0004  0000                ;STARTING AT 'STRING' OF A TEMPLATE OF
0005  0000                ;LENGTH 'TPTLEN' STARTING AT 'TEMPLT', AND
0006  0000                ;RETURNS WITH X=LOCATION OF TEMPLATE
0007  0000                ;IN STRING IF FOUND, OR X=$FF IF NOT FOUND.
0008  0000                ;
0009  0000                STRING = $20          ;1ST LOCATION OF STRING.
0010  0000                TEMPLT = $50          ;1ST LOCATION OF TEMPLATE.
0011  0000                       * = $10
0012  0010                CHKPTR *=*+1
0013  0011                TEMPTR *=*+1
0014  0012                STRLEN *=*+1          ;LENGTH OF STRING.
0015  0013                TPTLEN *=*+1          ;LENGTH OF TEMPLATE.
0016  0014                       * = $200
0017  0200  A2 00           NXTPOS LDX #0         ;RESET SEARCH START POINTER.
0018  0202  A5 50                  LDA TEMPLT     ;IS FIRST ELEMENT OF TEMPLATE...
0019  0204  D5 20                  CMP STRING,X   ;= CURRENT STRING ELEMENT?
0020  0206  F0 08                  BEQ CHECK      ;IF YES, CHECK FOR REST OF MATCH.
0021  0208  E8           NXTSTR INX               ;INCREMENT SEARCH START COUNTER.
0022  0209  E4 12                  CPX STRLEN     ;IS IT EQUAL TO STRING LENGTH?
0023  020B  D0 F5                  BNE NXTPOS     ;NO, CHECK NEXT STRING POSITION.
0024  020D  A2 FF                  LDX #$FF       ;YES, SET 'NOT FOUND' INDICATOR.
0025  020F  60                     RTS            ;RETURN: ALL CHRS CHECKED.
0026  0210  86 11           CHECK  STX TEMPTR     ;LET TEMPORARY POINTER=
0027  0212                       ;CURRENT STRING POINTER.
0028  0212  A9 00                  LDA #0
0029  0214  85 10                  STA CHKPTR     ;RESET TEMPLATE POINTER.
0030  0216  E6 11           CHKLP  INC TEMPTR     ;INCREMENT TEMPORARY POINTER.
0031  0218  E6 10                  INC CHKPTR     ;INCREMENT TEMPLATE POINTER.
0032  021A  A4 10                  LDY CHKPTR
0033  021C  C4 13                  CPY TPTLEN     ;DOES TEMPLATE POINTER=TEMPLATE LENGTH?
0034  021E  F0 0C                  BEQ FOUND      ;IF YES, TEMPLATE MATCHED.
0035  0220  B9 50 00               LDA TEMPLT,Y   ;LOAD TEMPLATE ELEMENT.
0036  0223  A4 11                  LDY TEMPTR
0037  0225  D9 20 00               CMP STRING,Y   ;COMPARE TO STRING CHR.
0038  0228  D0 DE                  BNE NXTSTR     ;IF NO MATCH, CHECK NEXT STRING CHR.
0039  022A  F0 EA                  BEQ CHKLP      ;IF MATCH, CHECK NEXT CHR.
0040  022C  60           FOUND  RTS               ;DONE.
0041  022D                       .END
```

Fig. 8-3: String zoeken: programma

SAMENVATTING

In dit hoofdstuk zijn gebruikelijke utiliteitsprogramma's gepresenteerd, die kombinaties van technieken gebruiken die in de voorgaande hoofdstukken beschreven zijn. Ze zouden je nu in staat moeten stellen om je eigen programma's te schrijven. Veel van deze programma's hebben een speciale gegevensstruktuur gebruikt, de tabel. Er zijn andere mogelijkheden om gegevens te struktureren, en die zullen nu beschouwd worden.

9
GEGEVENSSTRUKTUREN DEEL I: ONTWERP-KONCEPTEN

INLEIDING

Voor het ontwerpen van een goed programma moeten er twee dingen gebeuren: *ontwerp van een algoritme en ontwerp van gegevensstrukturen*. Voor de meeste eenvoudige programma's zijn er geen belangrijke gegevensstrukturen nodig, zodat het voornaamste probleem, dat bij het leren programmeren overwonnen moet worden het ontwerp van algoritmen is, en het efficient koderen in een bepaalde machinetaal. Dit hebben we nu bereikt. Voor het ontwerp van ingewikkelder programma's is een begrip van gegevensstrukturen nodig. In dit boek zijn tot nu toe al twee strukturen gebruikt: de tabel en de stapel. Het doel van dit hoofdstuk is het presenteren van andere, meer algemene, gegevensstrukturen die je zou willen gebruiken. Dit hoofdstuk is volledig onafhankelijk van de microprocessor, of zelfs van de geselekteerde computer. Het is theoretisch en behandelt de logische organisatie van gegevens in het systeem. Er zijn gespecialiseerde boeken met als onderwerp gegevensstrukturen, net zoals er gespecialiseerde boeken zijn over efficient vermenigvuldigen, delen, of andere algoritmes. Dit hoofdstuk moet daarom als een overzicht beschouwd worden, en moet zich beperken tot de wezenlijk kenmerken. Het heeft niet de pretentie volledig te zijn.

Laten we nu de belangrijkste gegevensstrukturen eens beschouwen.

Wijzers (Eng: Pointers)

Een wijzer is een getal dat gebruikt wordt om de eigenlijke gegevens aan te wijzen. Iedere wijzer is een adres. Ieder adres hoeft echter niet altijd een wijzer te zijn. Een adres is alleen dan een wijzer als het naar een bepaald type gegevens of gestruktureerde informatie wijst. We zijn al een typische wijzer tegengekomen: de stapelwijzer, die naar de top van de stapel wijst (of meestal naar een plaats boven de top). We zullen zien, dat de stapel een gebruikelijke gegevensstruktuur is, een LIFO struktuur genaamd.

Als een ander voorbeeld noemen we indirekte adressering, waarbij het indirekte adres altijd een wijzer naar de gegevens waar men toegang tot wil hebben.

Oefening 9.1: Onderzoek fig.9-1. Op adres 15 in het geheugen staat een wijzer naar tabel T. Tabel T begint op adres 500. Wat is de inhoud van de wijzer naar T?

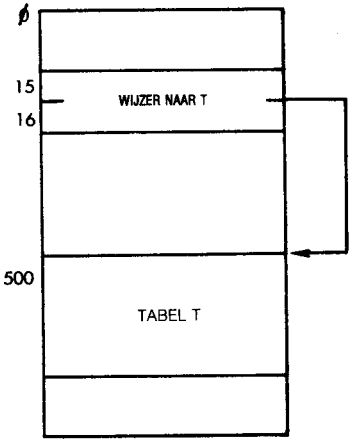


Fig. 9-1: Een indirektie wijzer

LIJSTEN

Bijna alle gegevensstrukturen zijn georganiseerd als een lijst van een of andere soort.

Sequentiele lijsten

Een sequentiele lijst, of tabel of blok, is waarschijnlijk de eenvoudigste gegevensstruktuur, en deze hebben we al gebruikt. Tabellen zijn meestal geordend volgens en of ander kriterium, zoals bijvoorbeeld een alfabetische of numerieke ordening. Het is dan eenvoudig om toegang tot een element van de tabel te krijgen door bijvoorbeeld geindexeerde adressering te gebruiken, zoals we al gedaan hebben. Een blok heeft meestal betrekking op een groep gegevens, die wel bepaalde grenzen heeft, maar geen specifieke ordening. Het zou bijvoorbeeld een karakterstring kunnen zijn. Of een sektor op een schijf. Of het zou een logisch gebied (segment genaamd) van het geheugen kunnen zijn. In dergelijke gevallen kan het moeilijker zijn om toegang te krijgen tot een bepaald element van het blok.

Om het verkrijgen van toegang tot een blok informatie te vereenvoudigen worden directories gebruikt:

Referentietabel (Eng: Directory)

Een referentietabel is een lijst van tabellen of blokken. Zo zal het filesysteem bijvoorbeeld een referentietabel-struktuur gebruiken. Als een eenvoudig voorbeeld, kan de hoofdreferentietabel van het systeem een lijst met de namen van de gebruikers bevatten. Dit is geillustreerd in fig. 9-2. De ingang van de gebruiker "Jan" wijst naar Jan zijn filere-

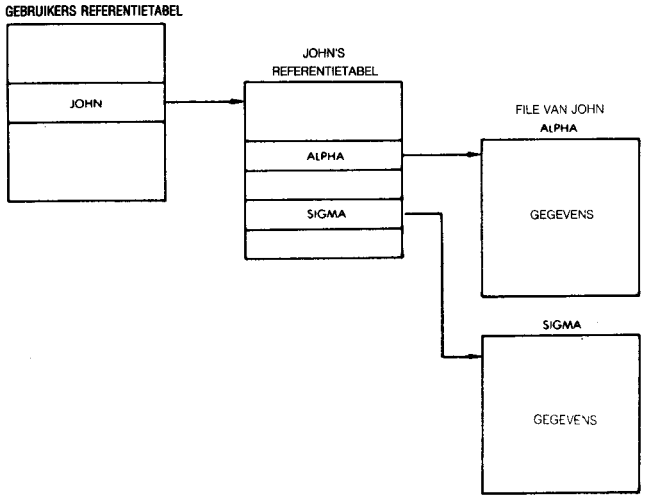


Fig. 9-2: Een referentietabel struktuur

ferentietabel. De file-referentietabel is een tabel die de namen van alle files van Jan bevat en hun plaats. Dit is weer een tabel met wijzers. In dit geval hebben we juist een referentietabel van twee niveau's ontworpen. Een flexibel referentietabel-systeem staan het invoegen van tussen-referentietabellen toe, al naar het de gebruiker uitkomt.

Geketende lijsten (Eng: linked lists)

In een systeem bevinden zich vaak blokken informatie die gegevens, gebeurtenissen of andere strukturen bevatten, die niet gemakkelijk verplaatst kunnen worden. Als ze wel eenvoudig verplaatst konden worden, zouden we ze waarschijnlijk in een tabel verzamelen om ze te structureren of te sorteren. Het probleem is nu dat we ze willen laten waar ze zijn, maar toch een ordening willen vastleggen tussen de blokken, zoals eerste, tweede, derde, vierde. Om dit probleem op te lossen wordt een geketende lijst gebruikt. Het koncept van een geketende lijst is geillustreerd in figuur 9-3. In de figuur kunnen we zien, dat een lijstwijzer, genaamd EERSTEBLOK, naar het begin van het eerste blok wijst. Een toegewezen plaats binnen Blok 1, zoals het eerste of het laatste woord, bevat een wijzer naar Blok 2, genaamd PTR1. Dit proces wordt dan herhaalt voor Blok 2 en Blok 3. Omdat Blok 3 het laatste blok is, bevat PTR3 bij afspraak een speciale "nil" waarde, of wijst anders naar zichzelf, zodat het einde van de lijst gedetekteerd kan worden. Deze struktuur is economisch omdat er maar weinig wijzers gebruikt worden (een per blok) en de gebruiker in staat stelt de blokken niet fysisch in het geheugen te hoeven verplaatsen.

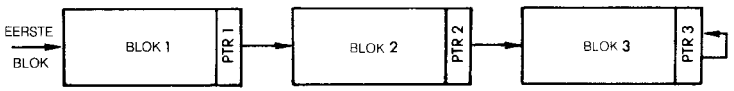


Fig. 9-3: Een geketende lijst

Laten we bijvoorbeeld eens onderzoeken hoe een nieuw blok tussengevoegd wordt. Dit is geillustreerd in fig. 9-4. Laten we aannemen dat het nieuwe blok begint op adres NIEUWBLOK, en dat het tussen Blok 1 en Blok 2 gevoegd moet worden. Wijzer PTR1 krijgt gewoon de waarde NIEUWBLOK, zodat hij naar Blok X wijst. PTRX krijgt de vorige waard van PTR1, d.w.z. hij wijst naar Blok 2. De andere wijzers in de struktuur blijven onveranderd. We hebben gezien, dat voor tussenvoegen van een nieuw blok alleen maar twee wijzers in de struktuur veranderd hoeven te worden. Het is duidelijk, dat dit efficient is.

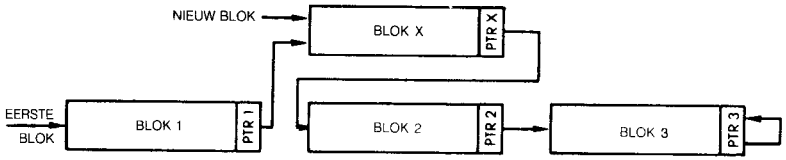


Fig. 9-4: Een nieuw blok toevoegen (inserten)

Oefening 9.2: Teken een diagram waarin te zien is hoe Blok 2 uit deze struktuur verwijderd wordt.

Er zijn verschillende soorten lijsten ontwikkeld om specifieke soorten van toegang, tussenvoeging, of verwijdering,te vereenvoudigen. Laten we de meest gebruikte soorten geketende lijsten eens beschouwen:

Rij (Eng: Queue)

Een rij wordt formeel een FIFO, of first-in-first-out (eerst-in-eerst-uit) lijst genoemd. Een rij is geillustreerd in fig. 9-5. Om het diagram te verduidelijken, kunnen we bijvoorbeeld aannemen, dat het linker blok een service routine is voor een uitvoerapparaat, zoals een printer. De rechter blokken zijn de verzoekblokken van verschillende program-

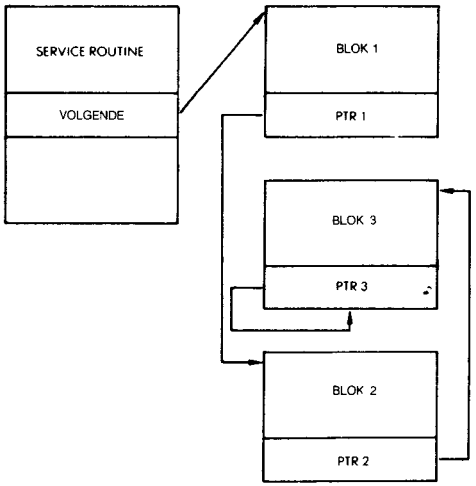


Fig. 9-5: Een rij

ma's of routines om karakters te printen. De volgorde waarin ze bediend worden, is de volgorde die door de wachtrij vastgelegd is. Het is te zien dat de volgende eenheid die service krijgt, Blok 1 is, daarna Blok 2, en de volgende is Blok 3. Bij een rij is de afspraak, dat iedere nieuwe gebeurtenis in de rij achterin geplaatst wordt. Hier wordt deze achter PTR3 geplaatst.
Dit garandeert, dat het eerste blok dat in de rij geplaatst is ook het eerst geholpen wordt. Het is in een computersysteem gebruikelijk om wachtrijen te hebben voor een aantal gebeurtenissen, als die moeten wachten op schaarse bronnen, zoals de processor of een I/O-apparaat.

Stapel

De stapel (stack) is al gedetailleerd bestudeerd in dit boek. Het is een LIFO-struktuur, last-in-first-out (laatst-in-eerst-uit). Het laatste op de top geplaatste element komt als eerste eruit. Een stapel kan uitgevoerd zijn als een gesorteerd blok, of anders als een lijst. Omdat de meeste stapels in microprocessoren gebruikt worden voor gebeurtenissen die met hoge snelheid afgehandeld moeten worden, zoals subroutines en interrupts, wordt meestal een aaneensluitend blok aan de stapel toegewezen in plaats van een geketende lijst.

Geketende lijsten vs. blokken

De rij zou ook als een blok gereserveerde plaatsen uitgevoerd kunnen worden. Het voordeel van het gebruik van een aaneensluitend blok is snelle toegang en het overbodig zijn van wijzers. Het nadeel is, dat meestal een tamelijk groot blok toegewezen moet worden om ruimte te bieden voor het slechtste geval van de struktuur. Met slechtste geval wordt die toestand bedoeld, die het meest ongunstig is. Het maakt het ook onpraktisch om elementen te verwijderen of tussen te voegen. Omdat het geheugen traditioneel een schaarse bron is, worden blokken traditioneel gereserveerd voor strukturen met een vaste omvang of voor strukturen waar snel toegang tot te krijgen moet zijn, zoals de stapel.

Circulaire lijst

Een circulaire lijst is een geketende lijst waar de laatste ingang naar de eerste wijst. Dit is geillustreerd in fig. 9-6. Bij een circulaire lijst wordt vaak een wijzer naar het huidige blok bijgehouden. Bij program-

ma's of gebeurtenissen, die op service wachten, wordt de huidige-blokwijzer iedere keer een positie naar rechts of naar links geschoven. Een circulaire lijst wordt meestal gebruikt als aangenomen wordt, dat alle blokken dezelfde prioriteit hebben. Een circulaire lijst kan echter ook gebruikt worden als een substruktuur van een andere struktuur, bijvoorbeeld om het krijgen van toegang tot het eerste blok, na het laatste, te vereenvoudigen, zoals bij een zoekaktie.

Als een voorbeeld van een circulaire lijst gaat een polling programma circulair langs de randapparaten en komt dan na de laatste weer bij de eerste terug.

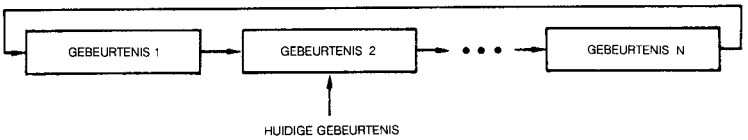


Fig. 9-6: Een circulaire lijst

Bomen (Eng: Trees)

Als er een logische relatie bestaat tussen de elementen van een struktuur (dit wordt meestal een syntax genoemd), kan een boomstruktuur gebruikt worden. Een eenvoudig voorbeeld van een boomstruktuur is een erfelijkheidsboom. Dit is geillustreerd in fig. 9-7. Te zien is, dat Smit twee kinderen heeft: een zoon, Robert, en een dochter, Jane. Jane heeft op haar beurt drie kinderen: Liz, Tom, en Phil. Tom heeft twee kinderen: Max en Chris. Robert, links in de figuur, heeft geen afstammelingen.

Dit is een gestruktureerde boom. We hebben in feite al een eenvoudige boom gezien in fig. 9-2. De referentietabel-struktuur is een boom met twee niveau's. Er kan met voordeel gebruik gemaakt worden van een boom, als elementen volgens een vaste struktuur geklassificeerd kunnen worden. Dit vereenvoudigt tussenvoeging en het krijgen van toegang. Verder kunnen bomen groepen informatie op een gestruktureerde manier vastleggen. Dit kan nodig zijn voor latere verwerking. zoals bij het ontwerp van een compiler of interpreter (zie hoofdstuk 10: software ondersteuning).

Dubbel geketende lijsten

Er kunnen nog meer verbindingen gelegd worden tussen de elemen-

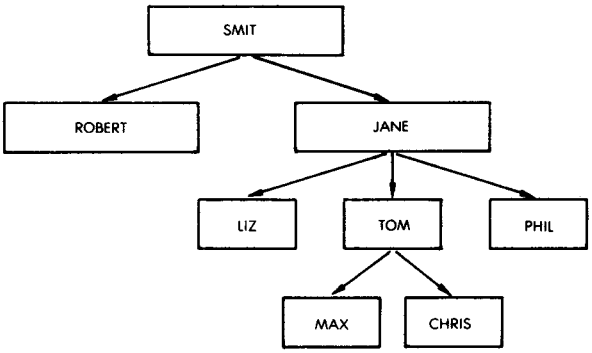


Fig. 9-7: Erfelijkheidsboom

ten van een lijst. Dit is geillustreerd in fig. 9-8. We kunnen zien, dat we de gebruikelijke serie verbindingen hebben van links naar rechts, plus een andere serie van rechts naar links. Het doel is om eenvoudig toegang te krijgen tot zowel het voorgaande, als het volgende element. Dit kost een extra wijzer per blok.

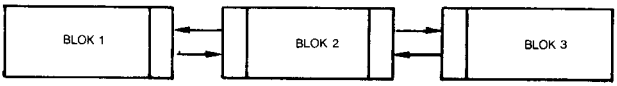


Fig. 9-8: Dubbel geketende lijst

ZOEKEN EN SORTEREN

Het zoeken en sorteren van elementen in een lijst is direkt afhankelijk van de gebruikte struktuur. Er zijn veel algoritmes ontwikkeld voor de meest voorkomende gegevensstrukturen. We hebben geindexeerde adressering al gebruikt. Dit is mogelijk als de elementen van een tabel volgens een bekend kriterium geordend zijn. Tot dergelijke elementen kan via hun nummer toegang verkregen worden.

Met *sequentieel zoeken* wordt het lineair afzoeken van een heel blok bedoeld. Het is duidelijk, dat dit inefficient is, maar kan gebruikt worden als er geen betere techniek voorhanden is, wegens het ontbreken van een ordening van de elementen.

Binair of logaritmisch zoeken probeert een element te vinden door

na iedere stap het zoekinterval te halveren. Als we bijvoorbeeld in een alfabetische lijst zoeken, kunnen we bijvoorbeeld in het midden beginnen en kijken of de naam die we zoeken voor of na dit punt ligt. Als de naam na dit punt ligt, elimineren we de eerste helft van de tabel en kijken in het midden van de tweede helft. We vergelijken deze ingang weer met wat we zoeken en beperken onze zoektocht weer tot een van de twee helften. De maximale zoeklengte is gegarandeerd minder dan log(2) n, waarbij n het aantal elementen in de tabel is.

Er bestaan veel andere technieken.

SAMENVATTING

Dit deel was bedoeld als een korte presentatie van gebruikelijke gegevensstrukturen die door de programmeur gebruikt kunnen worden. Hoewel de meeste gebruikelijke gegevensstrukturen wel onderverdeeld zijn en een naam hebben gekregen, kan de hele organisatie van gegevens in een systeem een willekeurige kombinatie ervan gebruiken, of van de programmeur eisen dat hij meer toepasselijke strukturen ontwerpt. De mogelijkheden zijn alleen beperkt door de fantasie van de programmeur. Er zijn ook een aanzienlijk aantal zoek- en sorteertechnieken ontwikkeld om de gebruikelijke gegevensstrukturen te lijf te gaan. Een uitgebreide beschrijving gaat buiten het bestek van dit boek. De inhoud van dit deel was bedoeld, om het belang van het ontwerpen van toepasselijke gegevensstrukturen te benadrukken, en om enige basistechnieken hiertoe aan te reiken.

GEGEVENSSTRUKTUREN DEEL II: ONTWERP-VOORBEELDEN

INLEIDING

We zullen hier een aantal ontwerpvoorbeelden behandelen voor veel voorkomende gegevensstrukturen: tabel, geketende lijst, gesorteerde boom. Voor deze strukturen zullen we algoritmes programmeren om te sorteren, te zoeken en tussen te voegen. Verder zullen we een paar geavanceerde technieken beschrijven zoals hashing en merging.

De lezer die in deze geavanceerde programmeertechnieken geinteresseerd is wordt aangemoedigd om de in dit deel gepresenteerde programma's in detail te bestuderen. De beginnende programmeur kan dit deel in eerste instantie overslaan en het doorlezen als hij er aan toe is.

Om deze voorbeelden te kunnen volgen, moeten de in het eerste deel behandelde begrippen goed begrepen worden. Ook wordt er in de programma's gebruik gemaakt van alle adresseertechnieken van de 6502, en worden de in de voorgaande hoofdstukken behandelde begrippen en technieken geintegreerd.

We zullen nu vier strukturen behandelen: een eenvoudige lijst, een alfabetische lijst, een geketende lijst met een referentietabel en een boom. Voor iedere struktuur worden drie programma's ontwikkeld: zoeken, invoegen en verwijderen.

Verder zullen achterin dit deel drie gespecialiseerde algoritmes afzonderlijk beschreven worden: hashing, bubble-sort en merging.

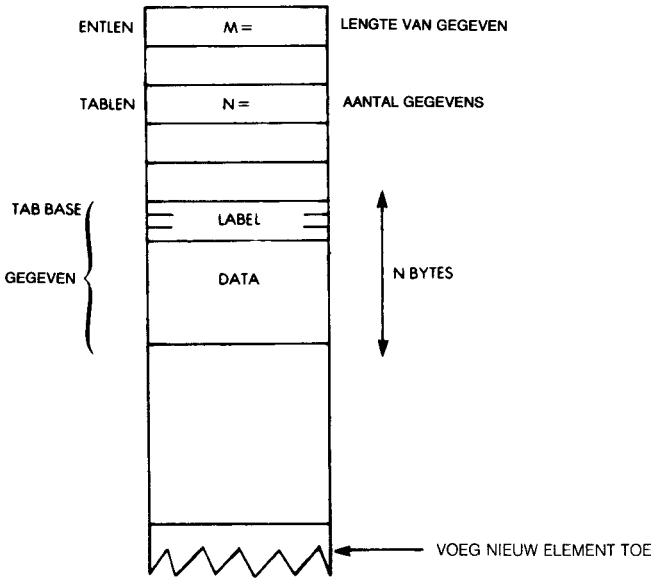


Fig. 9-9: De tabelstruktuur

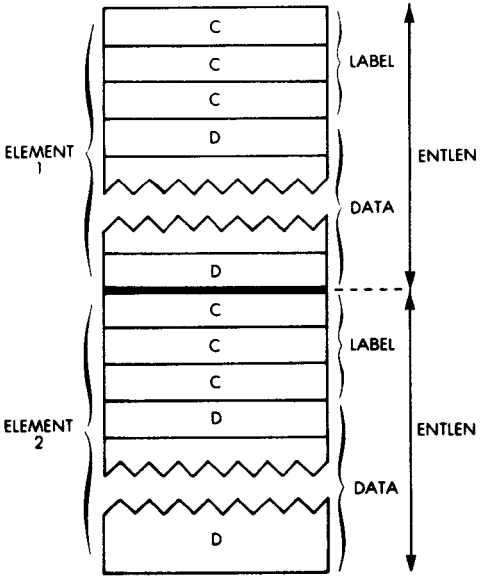
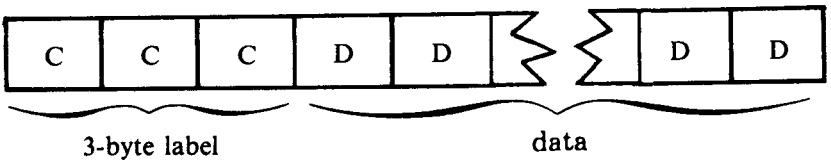


Fig. 9-10: Typische lijstingangen in het geheugen

GEGEVENSVOORSTELLING VOOR DE LIJST

Zowel de eenvoudige als de alfabetische lijst gebruiken dezelfde voorstelling voor ieder element van de lijst:



Ieder element of gegeven (entry) omvat een 3-byte label en een n-byte blok gegevens met n tussen 1 en 253. Ieder gegeven beslaat dus maximaal een pagina (256 bytes). Binnen iedere lijst hebben alle elementen dezelfde lengte (zie fig. 9-10). De programma's die met deze eenvoudige lijsten werken hebben enkele gemeenschappelijke afspraken met betrekking tot de variabelen:

ENTLEN is de lengte van een element. Als ieder element bijvoorbeeld 10 bytes gegevens heeft, is ENTLEN = 3 + 10 = 13 bytes
TABASE is de basis van de tabel of de lijst in het geheugen
POINTR is een variabele wijzer (pointer) naar het huidige element
OBJECT is het huidige gegeven dat ingevoegd of verwijderd moet worden
TABLEN is het aantal gegevens
We nemen aan, dat alle labels verschillend zijn.

EEN EENVOUDIGE LIJST

De eenvoudige lijst is georganiseerd als een tabel van n elementen. De elementen zijn niet gesorteerd (zie fig. 9-11).

Bij zoeken moet de hele lijst onderzocht worden tot het bepaalde gegeven gevonden is of het eind van de lijst bereikt is. Bij tussenvoegen worden nieuwe gegevens tussen bestaande gegevens gevoegd. Als een gegeven verwijderd wordt (deleten), worden eventuele gegevens op hogere geheugenadressen naar beneden geschoven om de tabel aaneengesloten te houden.

Zoeken

Er wordt een seriele zoektechniek gebruikt. Het label van ieder gegeven wordt letter voor letter vergeleken met het label van OBJECT.

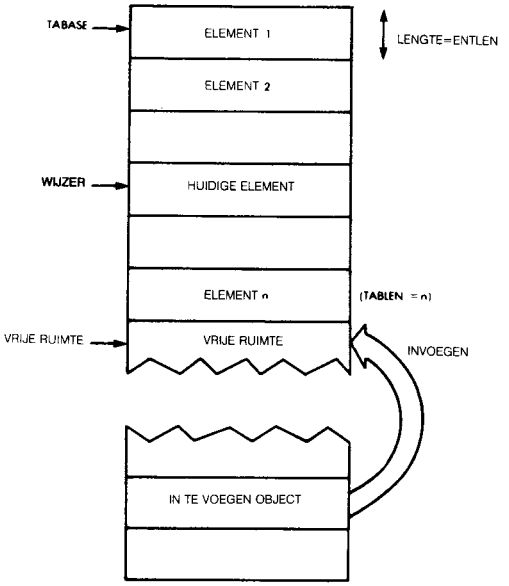


Fig. 9-11: De eenvoudige lijst

De variabele wijzer POINTR krijgt als beginwaarde de waarde van TABASE.

Indexregister X wordt geinitialiseerd met het aantal gegevens in de tabel (opgeslagen in TABLEN).

Er wordt op een voor de hand liggende manier gezocht en het overeenkomstige stroomdiagram is te zien in fig. 9-12. Het programma is aan het eind van dit deel gegeven (programma "SEARCH", fig. 9-16).

Tussenvoegen van elementen

Als er een nieuw tussengevoegd wordt, wordt het eerste beschikbare geheugenblok van (ENTLEN) bytes aan het einde van de lijst gebruikt (zie fig. 9-11).

Het programma kontroleert eerst of het nieuwe gegeven al niet in de lijst voorkomt. (Bij dit voorbeeld gaan we er van uit dat alle labels verschillend zijn.) Zo niet dan vermeerdert het de lijstlengte TABLEN, en verplaatst het OBJECT naar het einde van de lijst. Het overeenkomstige stroomdiagram is getoond in fig. 9-13.

Het nieuwe programma is te zien in fig. 9-16 aan het einde van dit deel. Het heet "NEW" en beslaat geheugenplaatsen 0636 tot 0659.

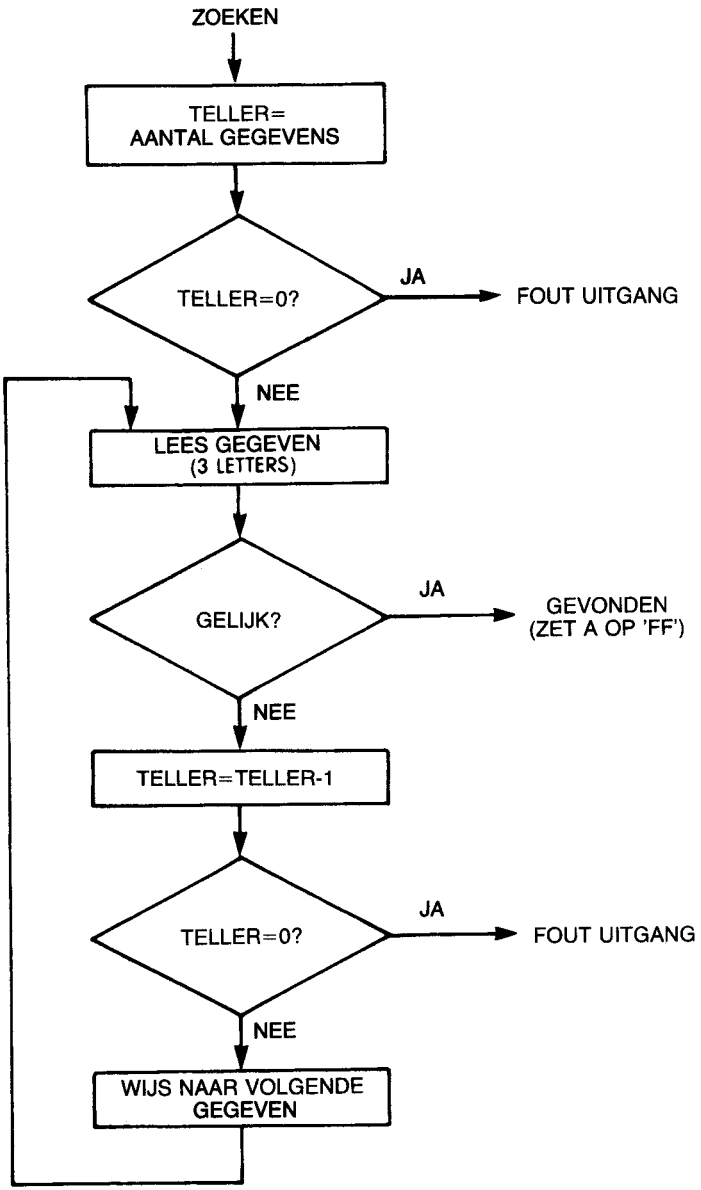


Fig. 9-12: Stroomdiagram voor het zoeken in een tabel

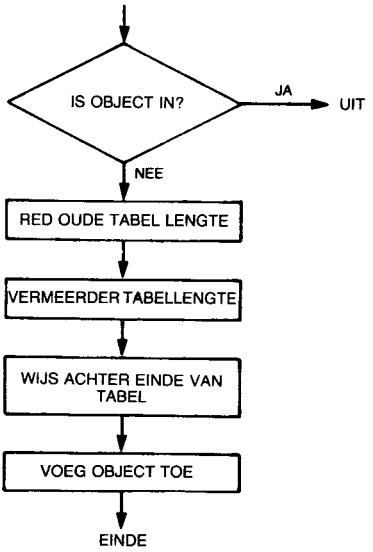


Fig. 9-13: Stroomdiagram voor het tussenvoegen

Het verwijderen van een element

Om een element uit de lijst te verwijderen, worden de elementen op hogere geheugenadressen gewoon een elementpositie opgeschoven. De lengte van de lijst wordt verminderd. Dit is geillustreerd in fig. 9-14.

Het overeenkomstige programma is erg eenvoudig en is gegeven in fig. 9-16. Het heet "DELETE" (verwijder) en beslaat geheugenadressen 0659 tot 0686. Het stroomdiagram is te zien in fig. 9-15.

Geheugenplaats TEMPTR wordt gebruikt als een tijdelijke wijzer om naar het element te wijzen dat omhoog geschoven wordt.

Indexregister Y krijgt als inhoud de lengte van een lijstelement, en wordt gebruikt voor de automatische verplaatsing van blokken. Merk op dat er gebruik gemaakt wordt van indirekt geindexeerde adressering:

```
(0672)    LOOPE  DEY
                 LDA     (TEMPTR), Y
                 STA     (POINTR), Y
                 CPY     #0
                 BNE     LOOPE
```

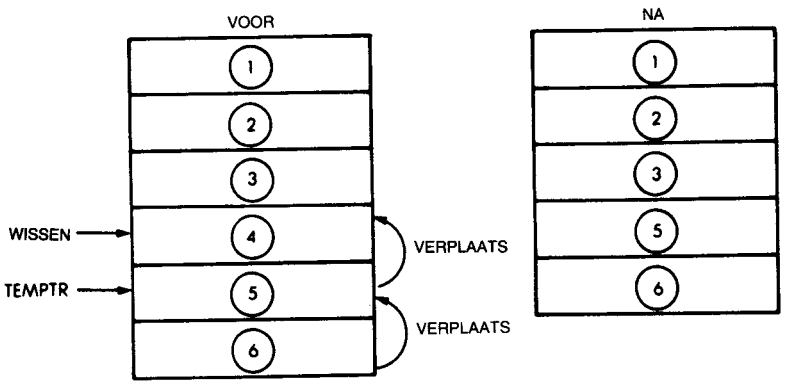


Fig. 9-14: Het verwijderen van een entry (eenvoudige lijst)

Tijdens de overdracht wijst POINTR altijd naar het "gat" in de lijst, dat wil zeggen de bestemming van de volgende blokoverdracht.

De Z-vlag wordt gebruikt om een succesvolle verwijdering aan te geven.

ALFABETISCHE LIJST

Bij een alfabetische lijst of "tabel", zijn alle elementen in tegenstelling met de vorige lijst, op alfabetische volgorde gesorteerd. Hierdoor kunnen snellere zoektechnieken gebruikt worden dan de lineaire. We zullen hier een binaire zoektechniek gebruiken.

Zoeken

Het zoekalgoritme is een klassieke binaire zoektechniek. De techniek is in wezen gelijk aan de techniek die we gebruiken om een naam in een telefoongids op te zoeken. Je begint meestal ergens in het midden en gaat dan, afhankelijk van wat daar gevonden wordt, verder of terug in de gids om het gewenste element te vinden. Deze methode is snel en redelijk eenvoudig uit te voeren.

Het stroomdiagram is te zien in fig. 9-17 en het programma in fig. 9-22.

Bij deze lijst blijven de elementen in alfabetische volgorde en de elementen worden gevonden door binair of "logaritmisch" te zoeken. Een voorbeeld is gegeven in fig. 9-18.

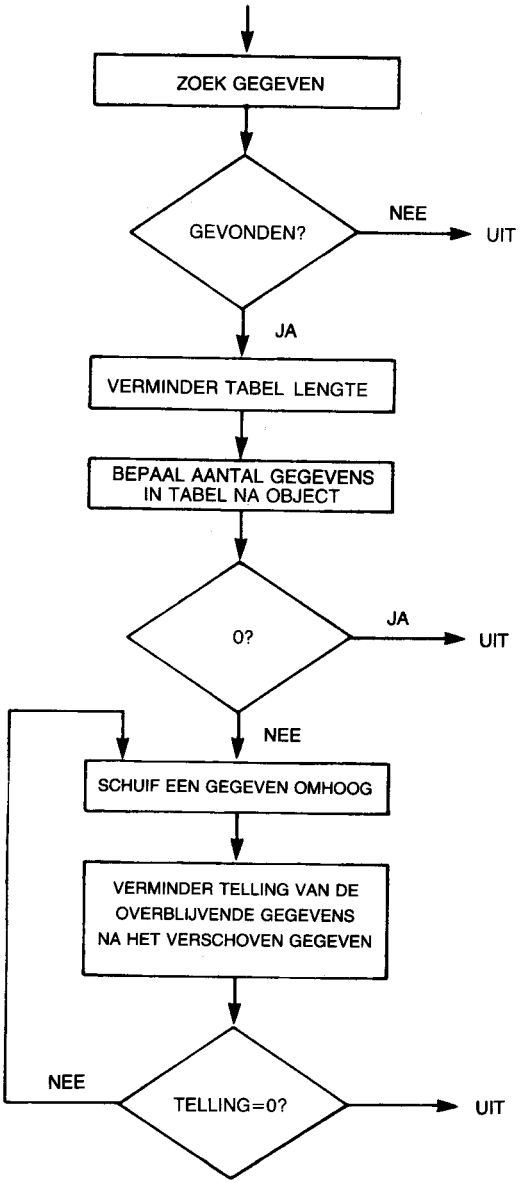


Fig. 9-15: Stroomdiagram voor het verwijderen

```
LINE # LOC     CODE       LINE

0002  0000              TABASE = $10
0003  0000              POINTR = $12
0004  0000              TABLEN = $14
0005  0000              OBJECT = $15
0006  0000              ENTLEN = $17
0007  0000              TEMPTR = $18
0008  0000              ;
0009  0000                     *=$600
0010  0600              ;
0011  0600  A5 10       SEARCH LDA TABASE      ;INITIALIZE POINTER
0012  0602  85 12              STA POINTR
0013  0604  A5 11              LDA TABASE+1
0014  0606  85 13              STA POINTR+1
0015  0608  A6 14              LDX TABLEN       ;STORE TABLEN AS A VARIABLE
0016  060A  F0 29              BEQ OUT          ;CHECK FOR 0 TABLE
0017  060C  A0 00       ENTRY  LDY #0           ;COMPARE FIRST LETTERS
0018  060E  B1 15              LDA (OBJECT),Y
0019  0610  D1 12              CMP (POINTR),Y
0020  0612  D0 0E              BNE NOGOOD
0021  0614  C8                 INY              ;COMPARE SECOND LETTERS
0022  0615  B1 15              LDA (OBJECT),Y
0023  0617  D1 12              CMP (POINTR),Y
0024  0619  D0 07              BNE NOGOOD
0025  061B  C8                 INY              ;COMPARE THIRD LETTERS
0026  061C  B1 15              LDA (OBJECT),Y
0027  061E  D1 12              CMP (POINTR),Y
0028  0620  F0 11              BEQ FOUND
0029  0622  CA          NOGOOD DEX              ;SEE HOW MANY ENTRIES ARE LEFT
0030  0623  F0 10              BEQ OUT
0031  0625  A5 17              LDA ENTLEN       ;ADD ENTLEN TO POINTER
0032  0627  18                 CLC
0033  0628  65 12              ADC POINTR
0034  062A  85 12              STA POINTR
0035  062C  90 DE              BCC ENTRY
0036  062E  E6 13              INC POINTR+1
0037  0630  4C 0C 06           JMP ENTRY
0038  0633  A9 FF       FOUND  LDA #$FF         ;CLEAR Z FLAG IF FOUND
0039  0635  60           OUT    RTS
0040  0636              ;
0041  0636              ;
0042  0636              ;
0043  0636  20 00 06    NEW    JSR SEARCH       ;SEE IF OBJECT IS THERE
0044  0639  D0 1B              BNE OUTE
0045  063B  A6 14              LDX TABLEN       ;CHECK FOR 0 TABLE
0046  063D  F0 0B              BEQ INSERT
0047  063F  A5 12              LDA POINTR       ;POINTER IS AT LAST ENTRY
0048  0641  18                 CLC              ;..MUST MOVE IT TO END OF TABLE
0049  0642  65 17              ADC ENTLEN
0050  0644  85 12              STA POINTR
0051  0646  90 02              BCC INSERT
0052  0648  E6 13              INC POINTR+1
0053  064A  E6 14       INSERT INC TABLEN       ;INCREMENT TABLE LENGTH
0054  064C  A0 00              LDY #0           ;MOVE OBJECT TO END OF TABLE
0055  064E  A6 17              LDX ENTLEN
0056  0650  B1 15       LOOP   LDA (OBJECT),Y
0057  0652  91 12              STA (POINTR),Y
0058  0654  C8                 INY
0059  0655  CA                 DEX
0060  0656  D0 F8              BNE LOOP
0061  0658  60           OUTE   RTS              ;Z SET IF WAS DONE
0062  0659              ;
0063  0659              ;
0064  0659              ;
0065  0659  20 00 06    DELETE JSR SEARCH       ;FIND WHERE OBJECT IS
0066  065C  F0 2B              BEQ OUTS         ;EXIT IF NOT FOUND
0067  065E  C6 14              DEC TABLEN       ;DECREMENT TABLE LENGTH
0068  0660  CA                 DEX              ;SEE HOW MANY ENTRIES ARE
```

Fig. 9-16: Programma's voor de eenvoudige lijst: Search (zoek), Enter (invoegen) en Delete (verwijderen).

```
0067  0661  F0 26           BEQ DONE        ;..AFTER ONE TO BE DELETED
0070  0663  A5 12     ADDEN LDA POINTR      ;ADD ENTLEN TO POINTER AND
0071  0665  18              CLC             ;..SAVE AT TEMP STORAGE
0072  0666  65 17           ADC ENTLEN
0073  0668  85 18           STA TEMPTR
0074  066A  A9 00           LDA #0
0075  066C  65 13           ADC POINTR+1    ;ADD CARRY TO HIGH BYTE
0076  066E  85 19           STA TEMPTR+1
0077  0670  A4 17           LDY ENTLEN
0078  0672  88        LOOPE DEY
0079  0673  B1 18           LDA (TEMPTR),Y  ;SHIFT ONE ENTRY OF MEMORY DOWN
0080  0675  91 12           STA (POINTR),Y
0081  0677  C0 00           CPY #0
0082  0679  D0 F7           BNE LOOPE
0083  067B  CA              DEX             ;DECREMENT ENTRY COUNTER
0084  067C  F0 0B           BEQ DONE
0085  067E  A5 18           LDA TEMPTR      ;MOVE TEMP TO POINTER
0086  0680  85 12           STA POINTR
0087  0682  A5 19           LDA TEMPTR+1
0088  0684  85 13           STA POINTR+1
0089  0686  4C 63 06        JMP ADDEN
0090  0689  A9 FF     DONE  LDA #$FF        ;CLEAR Z FLAG IF IT WAS DONE
0091  068B  60        OUTS  RTS
0092  068C                  ;
0093  068C                  ;
0094  068C                  .END


ERRORS = 0000 <0000>




SYMBOL TABLE

SYMBOL   VALUE

ADDEN    0663   DELETE   0659   DONE     0689   ENTLEN   0017
ENTRY    060C   FOUND    0633   INSERT   0644   LOOP     0650
LOOPE    0672   NEW      0636   NOGOOD   0622   OBJECT   0015
OUT      0635   OUTE     0658   OUTS     068B   POINTR   0012
SEARCH   0600   TABASE   0010   TABLEN   0014   TEMPTR   0018

END OF ASSEMBLY
```

Fig. 9-16: (Vervolg)

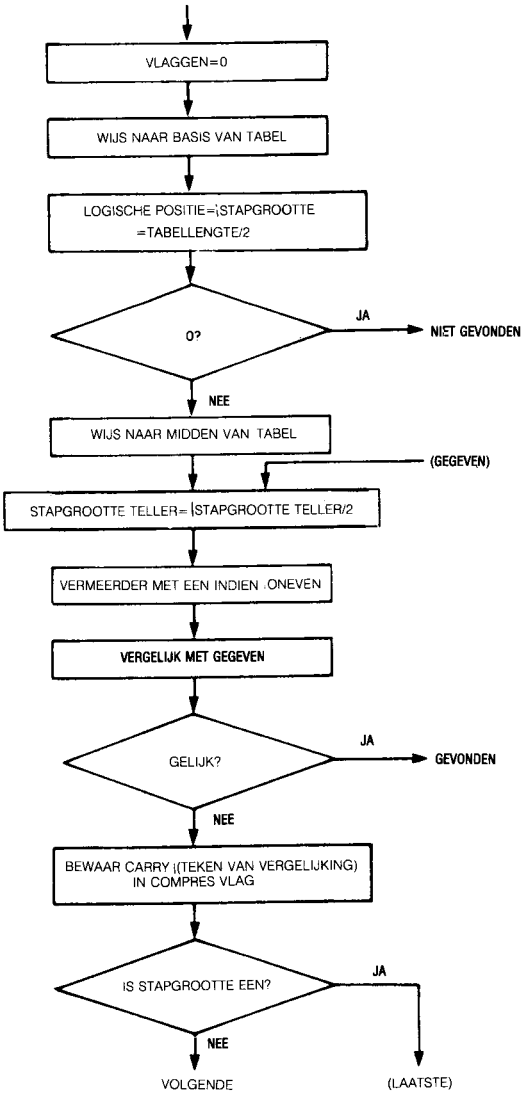


Fig. 9-17: Stroomdiagram voor binair zoeken

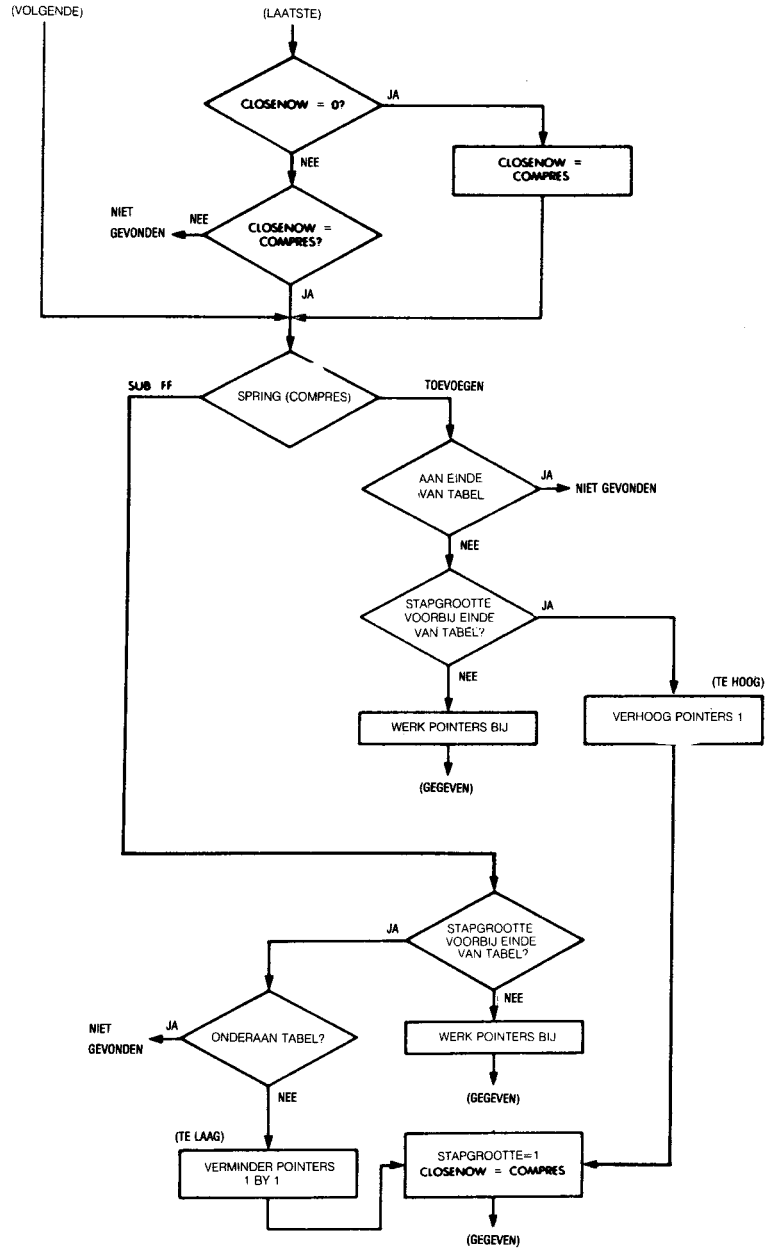


Fig. 9-17: Stroomdiagram voor binair zoeken (vervolg)

Het zoeken wordt iets gekompliceerd omdat er verschillende kondities bijgehouden moeten worden. Het voornaamste probleem dat vermeden moet worden is het zoeken naar een element dat er niet is. In een dergelijk geval zouden de gegevens met de naastliggende hogere en lagere alfabetische waarden voortdurend afwisselend getest worden.

Om dit te vermijden wordt in het programma een vlag bijgehouden om de waarde van de carry te bewaren na een niet succesvolle vergelijking. Als de INCMNT waarde, die aangeeft hoeveel de wijzer de volgende keer vermeerderd wordt, de waarde "1" bereikt, krijgt een andere vlag genaamd CLOSE de waarde van de CMPRS vlag. Dus is CMPRES niet langer gelijk aan CLOSE, omdat alle volgende vermeerderingen "1" zijn als de wijzer voorbij het punt komt waar het object zou moeten zijn, en het zoeken eindigt. Door deze eigenschap kan de NEW routine ook bepalen waar de logische en fysieke wijzers zijn, vergeleken met de plaats waar het object heengaat.

Als het OBJECT waar naar gezocht wordt niet in de tabel is, en de variabele wijzer met een vermeerderd wordt, wordt de CLOSE vlag gezet. Als de routine de volgende keer doorlopen wordt is het resultaat van de vergelijking tegengesteld aan het vorige. De twee vlaggen zijn

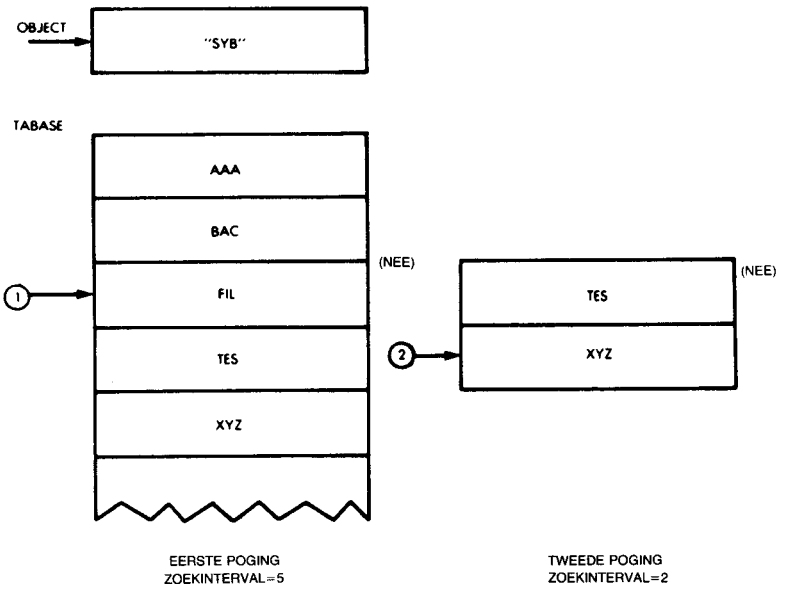


Fig. 9-18: Binair zoeken

niet langer gelijk, en het programma eindigt met de boodschap: "niet gevonden".

Een ander probleem waar we mee te maken hebben is de mogelijkheid dat we aan een van de einden van de tabel er uit schieten, als we de stapgrootte optellen of aftrekken. Dit wordt opgelost door een "optel" of "aftrek" test met de logische wijzer en de lengte om het eigenlijke aantal gegevens te bepalen, in plaats van gebruik te maken van fysieke wijzers om de posities te bepalen.

Samenvattend: er worden door het programma twee vlaggen gebruikt om informatie te onthouden: CMPRES en CLOSE. De CMPRES vlag wordt gebruikt om het feit te onthouden dat de carry "0" of "1" was na de laatste vergelijking. Dit bepaalt of het geteste element groter of kleiner was dan het element waar het mee vergeleken werd. Als de carry C "1" is, is het gegeven kleiner dan het object, en wordt CMPRES op "1" gezet. Als de carry C "0" is, is het gegeven groter dan het object en wordt CMPRES op "FF" gezet.

Merk ook op, dat als de carry "1" is de variabele wijzer naar het gegeven onder OBJECT wijst.

De tweede vlag die door het programma gebruikt wordt is CLOSE. Deze vlag wordt gelijk gemaakt aan CMPRES als de stapgrootte INCMNT "1" wordt. Hiermee wordt de volgende keer gedetekteerd dat het element niet gevonden is als CMPRES niet gelijk is aan CLOSE.

Andere variabelen die het programma gebruikt zijn:

LOGPOS geeft de logische positie in de tabel aan (elementnummer).

INCMNT geeft de stapgrootte aan waarmee de variabele wijzer vermeerderd of verminderd wordt als de volgende vergelijking faalt.

TABLEN geeft zoals gebruikelijk de totale lengte van de lijst aan. LOGPOS en INCMNT worden met TABLEN vergeleken om te verzekeren dat de grenzen van de tabel niet overschreden worden.

Het programma "SEARCH" is gegeven in fig. 9-22. Het is geplaatst op geheugenadressen 0600 tot 06E3 en moet eigenlijk zorgvuldig bestudeerd worden omdat het veel ingewikkelder is dan een lineair zoekprogramma.

Een andere komplikatie is het feit dat het zoekinterval soms even en soms oneven is. Als het even is, moet er een korrektie uitgevoerd worden. Het kan bijvoorbeeld niet naar het middelste element van een lijst met vier elementen wijzen.

Als het even is, gebruiken we een truuk om naar het middelste element te wijzen: het delen door 2 wordt bereikt door naar rechts te

schuiven. Het bit dat na de LSR instruktie in de carry "valt" is "1" als het interval oneven was. Het wordt gewoon weer bij de wijzer opgeteld:

```
(0615) DIV   LSR   A        DEEL DOOR TWEE
             ADC   #0       PAK DE CARRY
             STA   LOGPOS   NIEUWE WIJZER
```

Het OBJECT wordt dan vergeleken met het gegeven in het midden van het nieuwe interval. Als de vergelijking slaagt eindigt het programma. Anders ("NOGOOD"), als OBJECT kleiner is dan het gegeven, wordt de carry op 0 gezet. Zodra INCMNT "1" wordt, wordt de CLOSE vlag (die 0 als beginwaarde heeft) gekontroleerd of hij gezet is. Als dit niet het geval is wordt hij gezet. Als hij al gezet was, kontroleren we of we al voorbij de plaats zijn waar OBJECT zou moeten zijn, maar niet gevonden werd.

Invoegen van een element

Om een nieuw element in te voegen wordt een binaire zoekaktie uitgevoerd. Als het element al in de tabel stond hoeft het niet ingevoegd te worden. (We nemen hier aan dat alle elementen verschillend zijn.) Als het element niet gevonden werd, moet het ingevoegd worden. Na de zoekaktie geeft de waarde van de CMPRES vlag aan of het element direkt voor of direkt na het element, waar het mee vergeleken is, ingevoegd moet worden. Alle elementen volgend op de lokatie waar het geplaatst moet worden, worden dan een blokpositie opgeschoven, en het nieuwe element wordt ingevoegd.

Het invoegproces is geillustreerd in fig. 9-19 en het overeenkomstige programma is te zien in fig. 9-22.

Het programma is genaamd "NEW" en staat op geheugenplaatsen 06E3 tot 075E.

Merk op dat er weer gebruik gemaakt wordt van indirekte geindexeerde adressering voor de overdracht van blokken:

```
(072A)          LDA   ENTLEN
      ANOTHR    DEY
                LDA   (POINTR), Y
                STA   (TEMP), Y
                CPY   #0
                BNE   ANOTHR
```

Merk hetzelfde op bij geheugenplaats 0750.

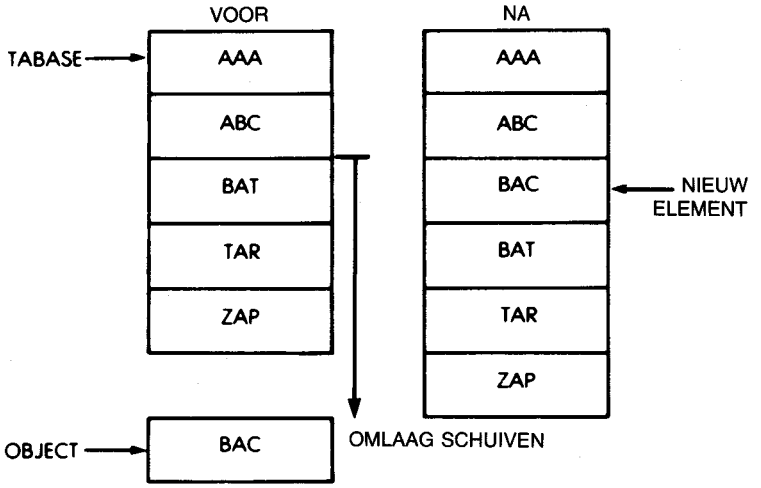


Fig. 9-19: Voeg "BAC" in

Verwijderen van een element

Net als bij het invoegen, wordt voor het verwijderen van een element een binaire zoekaktie uitgevoerd, om het object te vinden. Als het object niet gevonden wordt, hoeft het niet verwijderd te worden. Als het wel gevonden wordt, wordt het verwijderd en worden de volgende elementen een blokpositie opgeschoven. In fig. 9-20 is een voorbeeld gegeven, het programma staat in fig. 9-22. Het stroomdiagram is te zien in fig. 9-21.

De naam is "DELETE" en het staat op geheugenadressen 075F tot 0799.

GEKETENDE LIJST

We nemen aan, dat de geketende lijst weer drie alfabetische karakters bevat voor het label, gevolgd door 1 tot 250 databytes, gevolgd door een 2-bytes wijzer die het startadres bevat van het volgende gegeven, en afgesloten door een merkteken van 1 byte. Als dit merkteken "1" is, voorkomt het dat de invoeg routine een nieuw gegeven substitueert op de plaats van een reeds bestaand gegeven.

Verder bevat een directory (inhoudsopgave) een wijzer naar het eerste gegeven voor iedere letter van het alfabet om het toegang krijgen

tot de gegevens te vereenvoudigen. In het programma wordt aangenomen, dat de labels alfabetische ASCII karakters zijn. Alle wijzers aan het einde van de lijst worden op een NIL waarde gezet, waarvoor hier de waarde van de basis van de tabel gekozen is, omdat deze waarde nergens in de geketende lijst mag voorkomen.

De programma's om in te voegen en te verwijderen verrichten de gebruikelijke wijzermanipulaties. Ze gebruiken de vlag INDEXD om aan te geven of een wijzer die naar een object wijst uit een voorgaand gegeven in de lijst kwam dan wel uit de referentietabel. De overeenkomstige programma's zijn te zien in fig. 9-27, de gegevensstruktuur is getoond in fig. 9-23.

Een toepassing van deze gegevensstruktuur zou een met een computer uitgevoerd adresboek kunnen zijn, waarbij ieder persoon voorgesteld wordt door een unieke kode van drie letters (bijvoorbeeld de gebruikelijke initialen) en het gegevensveld een vereenvoudigd adres en het telefoonnummer bevat (samen maximaal 250 karakters).

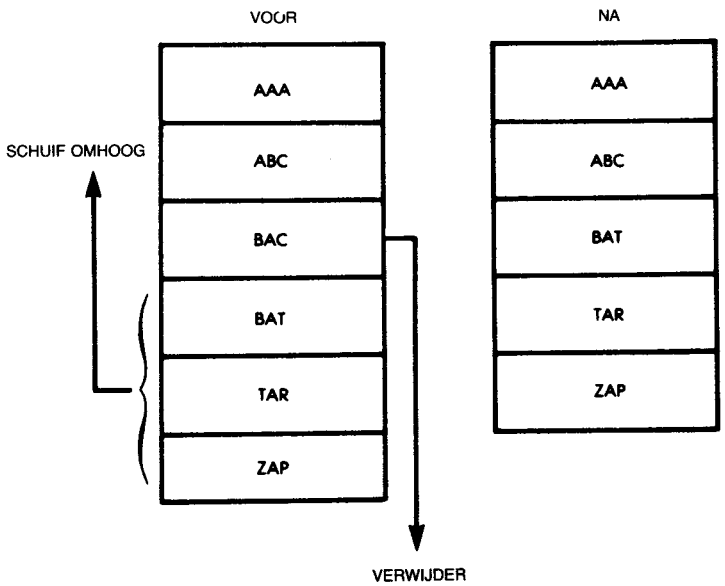


Fig. 9.20: Verwijder "BAC"

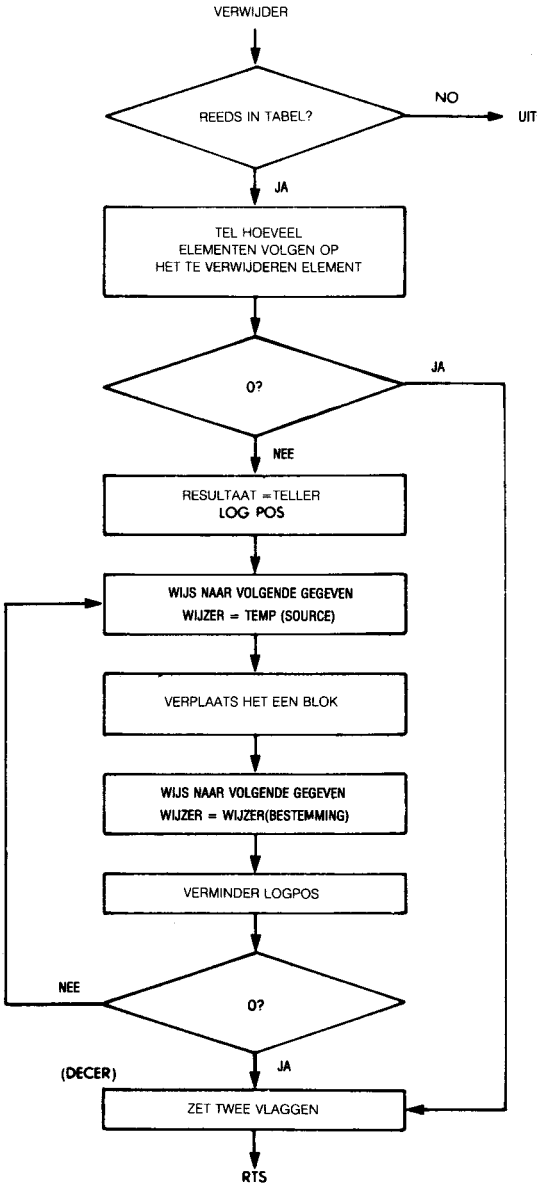


Fig. 9-21: Stroomdiagram voor verwijdering (alfabetische lijst)

```
LINE # LOC     CODE       LINE

0002  0000              CLOSE  = $10
0003  0000              CMPRES = $11
0004  0000              TABASE = $12
0005  0000              POINTR = $14
0006  0000              TABLEN = $16
0007  0000              LOGPOS = $17
0008  0000              INCMNT = $18
0009  0000              TEMP   = $19
0010  0000              ENTLEN = $1B
0011  0000              OBJECT = $1C
0012  0000              ;
0013  0000                     * = $600
0014  0600              ;
0015  0600  A9 00       SEARCH LDA #0          ;ZERO FLAGS
0016  0602  85 10              STA CLOSE
0017  0604  85 11              STA CMPRES
0018  0606  A5 12              LDA TABASE      ;INITIALIZE POINTER
0019  0608  85 14              STA POINTR
0020  060A  A5 13              LDA TABASE+1
0021  060C  85 15              STA POINTR+1
0022  060E  A5 16              LDA TABLEN      ;GET TABLE LENGTH
0023  0610  D0 03              BNE DIV
0024  0612  4C E0 06           JMP OUT
0025  0615  4A          DIV    LSR A           ;DIVIDE IT BY 2
0026  0616  69 00              ADC #0          ;ADD BACK IN 1 'S BIT
0027  0618  85 17              STA LOGPOS      ;STORE AS LOGICAL POSITION
0028  061A  85 18              STA INCMNT      ;STORE AS INCREMENT VALUE
0029  061C  A6 17              LDX LOGPOS      ;MULTIPLY ENTLEN BY LOGPOS
0030  061E  CA                 DEX             ;..ADDING RESULT TO POINTER
0031  061F  F0 0E              BEQ ENTRY
0032  0621  A5 1B       LOOP   LDA ENTLEN
0033  0623  18                 CLC
0034  0624  65 14              ADC POINTR
0035  0626  85 14              STA POINTR
0036  0628  90 02              BCC LOPP
0037  062A  E6 15              INC POINTR+1
0038  062C  CA           LOPP   DEX
0039  062D  D0 F2              BNE LOOP
0040  062F  A5 18       ENTRY  LDA INCMNT      ;DIVIDE INCREMENT VALUE BY 2
0041  0631  4A                 LSR A
0042  0632  69 00              ADC #0
0043  0634  85 18              STA INCMNT
0044  0636  A0 00              LDY #0          ;COMPARE FIRST LETTERS
0045  0638  B1 1C              LDA (OBJECT),Y
0046  063A  D1 14              CMP (POINTR),Y
0047  063C  D0 11              BNE NOGOOD
0048  063E  C8                 INY             ;COMPARE 2ND LETTERS
0049  063F  B1 1C              LDA (OBJECT),Y
0050  0641  D1 14              CMP (POINTR),Y
0051  0643  D0 0A              BNE NOGOOD
0052  0645  C8                 INY             ;COMPARE 3RD LETTERS
0053  0646  B1 1C              LDA (OBJECT),Y
0054  0648  D1 14              CMP (POINTR),Y
0055  064A  D0 03              BNE NOGOOD
0056  064C  4C E2 06           JMP FOUND
0057  064F  A0 FF       NOGOOD LDY #$FF        ;SET COMPARE RESULT FLAG
0058  0651  90 02              BCC TESTS       ;IF OBJ < POINTR :  C=0
0059  0653  A0 01              LDY #1
0060  0655  84 11       TESTS  STY CMPRES
0061  0657  A4 18              LDY INCMNT      ;IS INCR. VALUE A 1?
0062  0659  88                 DEY
0063  065A  D0 10              BNE NEXT
0064  065C  A5 10              LDA CLOSE       ;CHECK CLOSE FLAG IF IT WAS
0065  065E  F0 08              BEQ MAKCLO      ;IF CLOSE FLAG NOT SET, GO DO IT
0066  0660  38                 SEC
0067  0661  E5 11              SBC CMPRES      ;SEE IF GAVE PASSED WHERE OBJ.
0068  0663  F0 07              BEQ NEXT        ;..SHOULD BE BUT ISNT
```

Fig. 9-22: Programma's voor een alfabetische lijst:
Binair zoeken (SEARCH), verwijderen (DELETE) en invoegen (INSERT)

```
0069  0665  4C E0 06        JMP OUT
0070  0668  A5 11     NAKCLO LDA CMPRES      ;SET CLOSE FLAG TO CMPRES
0071  066A  85 10            STA CLOSE
0072  066C  24 11     NEXT   BIT CMPRES
0073  066E  30 35            BMI SUBIT
0074  0670  A5 14            LDA TABLEN      ;SEE IF ADDITION OF INCNNT
0075  0672  38               SEC             ;..WILL RUN PAST END OF TABLE
0076  0673  E5 17            SBC LOGPOS
0077  0675  F0 69            BEQ OUT         ;CHECK TO SEE IF AT END OF TABLE ALREADY
0078  0677  E5 18            SBC INCNNT
0079  0679  90 1A            BCC TOOHI
0080  067B  A6 18            LDX INCNNT      ;IS ALL RIGHT, INC POINTER BY
0081  067D  A5 1B     ADDER  LDA ENTLEN      ;..PROPER AMOUNT
0082  067F  18               CLC
0083  0680  65 14            ADC POINTR
0084  0682  85 14            STA POINTR
0085  0684  90 02            BCC AB1
0086  0686  E6 15            INC POINTR+1
0087  0688  CA        AB1    DEX
0088  0689  D0 F2            BNE ADDER
0089  068B  A5 17            LDA LOGPOS      ;INCREMENT LOGICAL POSITION
0090  068D  18               CLC
0091  068E  65 18            ADC INCNNT
0092  0690  85 17            STA LOGPOS
0093  0692  4C 2F 06         JMP ENTRY
0094  0695  E6 17     TOOHI  INC LOGPOS      ;INCR. LOGICAL POSITION
0095  0697  A5 1B            LDA ENTLEN      ;MOVE POINTER UP ONE ENTRY
0096  0699  18               CLC
0097  069A  65 14            ADC POINTR
0098  069C  85 14            STA POINTR
0099  069E  90 35            BCC SETCLO
0100  06A0  E6 15            INC POINTR+1
0101  06A2  4C B5 06         JMP SETCLO
0102  06A5  A5 17     SUBIT  LDA LOGPOS      ;SEE IF INC WILL GO OFF BOTTOM
0103  06A7  38               SEC             ;.. OF TABLE
0104  06A8  E5 18            SBC INCNNT
0105  06AA  F0 17            BEQ TOOLOW
0106  06AC  90 15            BCC TOOLOW
0107  06AE  85 17            STA LOGPOS      ;SAVE NEW LOGICAL POSITION
0108  06B0  A6 18            LDX INCNNT
0109  06B2  A5 14     SUBLOP LDA POINTR      ;SUBTRACT PROPER AMT. FROM POINTER
0110  06B4  38               SEC
0111  06B5  E5 1B            SBC ENTLEN
0112  06B7  85 14            STA POINTR
0113  06B9  B0 02            BCS SUB0
0114  06BB  C6 15            DEC POINTR+1
0115  06BD  CA        SUB0   DEX
0116  06BE  D0 F2            BNE SUBLOP
0117  06C0  4C 2F 06         JMP ENTRY
0118  06C3  A6 17     TOOLOW LDX LOGPOS      ;SEE IF POS IS ALREADY 1
0119  06C5  CA               DEX
0120  06C6  F0 18            BEQ OUT
0121  06C8  C6 17            DEC LOGPOS
0122  06CA  A5 14            LDA POINTR      ;SUB 1 ENTRY FROM POINTER
0123  06CC  38               SEC
0124  06CD  E5 1B            SBC ENTLEN
0125  06CF  85 14            STA POINTR
0126  06D1  B0 02            BCS SETCLO
0127  06D3  C6 15            DEC POINTR+1
0128  06D5  A9 01     SETCLO LDA #1
0129  06D7  85 18            STA INCNNT
0130  06D9  A5 11            LDA CMPRES
0131  06DB  85 10            STA CLOSE
0132  06DD  4C 2F 06         JMP ENTRY
0133  06E0  A2 FF     OUT    LDX #$FF        ;Z SET IF FOUND
0134  06E2  60        FOUND  RTS
0135  06E3                   ;
0136  06E3                   ;
0137  06E3                   ;
0138  06E3  20 00 06  NEW    JSR SEARCH      ;SEE IF OBJECT IS ALREADY THERE
```

Fig. 9-22: Programma's voor een alfabetische lijst:
Binair zoeken (SEARCH), verwijderen (DELETE) en invoegen (INSERT) (vervolg)

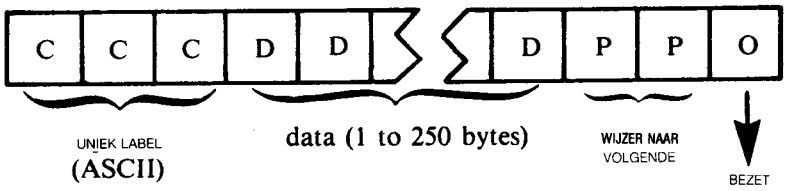
```
0139  06E6  F0 76          BEQ OUTE
0140  06E8  A5 16          LDA TABLEN      ;CHECK FOR 0 TABLE
0141  06EA  F0 62          BEQ INSERT
0142  06EC  24 11          BIT CMPRES      ;TEST LAST COMPARE RESULT
0143  06EE  10 05          BPL LOSIDE
0144  06F0  C6 17          DEC LOGPOS      ;SET LOGICAL POSITION SO
0145  06F2  4C 00 07       JMP SETUP       ;..SUB WORKS LATER
0146  06F5  A5 1B    LOSIDE LDA ENTLEN      ;SET POINTER ABOVE WHERE
0147  06F7  18             CLC             ;..OBJECT WILL GO
0148  06F8  65 14          ADC POINTR
0149  06FA  85 14          STA POINTR
0150  06FC  90 02          BCC SETUP
0151  06FE  E6 15          INC POINTR+1
0152  0700  A5 16    SETUP LDA TABLEN      ;SEE HOW MANY ENTRIES THERE
0153  0702  38             SEC             ;..ARE AFTER WHERE OBJ. WILL GO
0154  0703  E5 17          SBC LOGPOS
0155  0705  F0 47          BEQ INSERT
0156  0707  AA             TAX
0157  0708  A8             TAY
0158  0709  88             DEY             ;SEE IF ALREADY POINTING TO
0159  070A  F0 0E          BEQ SETEMP      ;..LAST ENTRY
0160  070C  A5 1B    UPLOOP LDA ENTLEN      ;MOVE POINTER TO LAST ENTRY
0161  070E  18             CLC
0162  070F  65 14          ADC POINTR
0163  0711  85 14          STA POINTR
0164  0713  90 02          BCC SET0
0165  0715  E6 15          INC POINTR+1
0166  0717  88        SET0  DEY
0167  0718  D0 F2          BNE UPLOOP
0168  071A  A5 14    SETEMP LDA POINTR      ;ADD ENTLEN TO POINTER
0169  071C  18             CLC             ;..STORE AT TEMP
0170  071D  65 1B          ADC ENTLEN
0171  071F  85 19          STA TEMP
0172  0721  90 01          BCC SET1
0173  0723  C8             INY             ;IF WAS ALREADY 0
0174  0724  98        SET1  TYA
0175  0725  18             CLC
0176  0726  65 15          ADC POINTR+1
0177  0728  85 1A          STA TEMP+1
0178  072A  A4 1B    MOVER LDY ENTLEN      ;SET Y FOR SHIFT
0179  072C  88        ANOTHR DEY
0180  072D  B1 14          LDA (POINTR),Y  ;MOVE A BYTE
0181  072F  91 19          STA (TEMP),Y
0182  0731  C0 00          CPY #0
0183  0733  D0 F7          BNE ANOTHR
0184  0735  A5 14          LDA POINTR      ;DECR. POINTER AND TEMP
0185  0737  38             SEC             ;..BY ENTLEN
0186  0738  E5 1B          SBC ENTLEN
0187  073A  85 14          STA POINTR
0188  073C  B0 02          BCS M1
0189  073E  C6 15          DEC POINTR+1
0190  0740  CA        M1    DEX
0191  0741  D0 D7          BNE SETEMP
0192  0743  A5 1B          LDA ENTLEN      ;MOVE POINTER BACK TO
0193  0745  18             CLC             ;WHERE OBJ. WILL GO
0194  0746  65 14          ADC POINTR
0195  0748  85 14          STA POINTR
0196  074A  90 02          BCC INSERT
0197  074C  E6 15          INC POINTR+1
0198  074E  A0 00    INSERT LDY #0         ;MOVE OBJECT INTO TABLE
0199  0750  A6 1B          LDX ENTLEN
0200  0752  B1 1C    INNER LDA (OBJECT),Y
0201  0754  91 14          STA (POINTR),Y
0202  0756  C8             INY
0203  0757  CA             DEX
0204  0758  D0 F8          BNE INNER
0205  075A  E6 16          INC TABLEN      ;INCREMENT TABLE LENGTH
0206  075C  A2 FF          LDX #$FF
```

Fig. 9-22: Programma's voor een alfabetische lijst:
Binair zoeken (SEARCH), verwijderen (DELETE) en invoegen (INSERT) (vervolg)

```
0207  075E  60          OUTE   RTS             ;Z SET IF NOT DONE
0208  075F                ;
0209  075F                ;
0210  075F                ;
0211  075F  20 00 06     DELETE JSR SEARCH      ;GET ADDR OF OBJECT IN TABLE
0212  0762  D0 35          BNE OUTS            ;SEE IF IT IS THERE
0213  0764  A5 16          LDA TABLEN          ;SEE HOW MANY ENTRIES ARE
0214  0766  38             SEC                 ;..LEFT AFTER OBJ. IN TABLE
0215  0767  E5 17          SBC LOGPOS
0216  0769  F0 2A          BEQ DECER
0217  076B  85 17          STA LOGPOS          ;STORE RESULT AS A COUNTER
0218  076D  A5 1B        BIGLOP LDA ENTLEN     ;SET TEMP 8 ENTRY ABOVE 1 ENTRY ABOVE OBJ.
0219  076F  18             CLC
0220  0770  65 14          ADC POINTR
0221  0772  85 19          STA TEMP
0222  0774  A9 00          LDA #0
0223  0776  65 15          ADC POINTR+1
0224  0778  85 1A          STA TEMP+1
0225  077A  A6 1B          LDX ENTLEN          ;SET COUNTERS
0226  077C  A0 00          LDY #0
0227  077E  B1 19        BYTE   LDA (TEMP),Y   ;MOVE A BYTE
0228  0780  91 14          STA (POINTR),Y
0229  0782  C8             INY                 ;IS BLOCK MOVED YET?
0230  0783  CA             DEX
0231  0784  D0 F8          BNE BYTE
0232  0786  A5 1B          LDA ENTLEN
0233  0788  18             CLC
0234  0789  65 14          ADC POINTR
0235  078B  85 14          STA POINTR
0236  078D  90 02          BCC D2
0237  078F  E6 15          INC POINTR+1
0238  0791  C6 17        D2     DEC LOGPOS
0239  0793  D0 D8          BNE BIGLOP
0240  0795  C6 16        DECER  DEC TABLEN
0241  0797  A9 00          LDA #0              ;Z SET IF WAS DONE
0242  0799  60          OUTS   RTS
0243  079A                .END

ERRORS = 0000 <0000>


SYMBOL TABLE

SYMBOL    VALUE

AD1       0688   ADDER     067B   ANOTHR    072C   BIGLOP    076D
BYTE      077E   CLOSE     0010   CNPRES    0011   D2        0791
DECER     0795   DELETE    075F   DIV       0615   ENTLEN    001B
ENTRY     062F   FOUND     06E2   INCMNT    001A   INNER     0752
INSERT    074E   LOGPOS    0017   LOOP      0621   LOPP      062C
LOSIDE    06F5   M1        0740   MAKCLO    066B   MOVER     072A
NEW       06E3   NEXT      066C   NOGOOD    064F   OBJECT    001C
OUT       06E0   OUTE      075E   OUTS      0799   POINTR    0014
SEARCH    0600   SET0      0717   SET1      0724   SETCLO    0685
SETEMP    071A   SETUP     0700   SUB0      06BB   SUBIT     06A5
SUBLOP    06B2   TABASE    0012   TABLEN    0016   TEMP      0019
TESTS     0655   TOOHI     0695   TOOLOW    06C3   UPLOOP    070C

END OF ASSEMBLY

<
```

Fig. 9-22: Programma's voor een alfabetische lijst:
Binair zoeken (SEARCH), verwijderen (DELETE) en invoegen (INSERT) (vervolg)

Laten we de struktuur een iets gedetaileerder bekijken in fig. 9-23. Het formaat van een gegeven is:



De afspraken zijn weer:

ENTLEN: totale elementlengte (in bytes)
TABASE: adres van basis van de lijst
TABLEN: aantal ingangen (1 tot 256)

Hier wijst REFBASE naar het basisadres van de directory, of de "referentietabel".

Ieder twee-byte adres in deze referentietabel wijst naar de plaats waar de overeenkomstige letter voor het eerst in de lijst voorkomt. Zo vormt iedere groep gegevens met dezelfde beginletter een aparte lijst in de struktuur. Deze eigenschap maakt het opzoeken eenvoudiger en vergelijkbaar met een adresboek. Merk op dat er geen gegevens verplaatst worden tijdens een verwijdering of een invoeging. Er worden alleen wijzers veranderd, zoals in iedere goed opgezette struktuur.

Als er geen gegeven met een specifieke beginletter gevonden wordt, of als er een gegeven is die alfabetisch op een bestaande volgt, wijzen hun wijzers naar het begin van de tabel (= "NIL"). Onderin de tabel

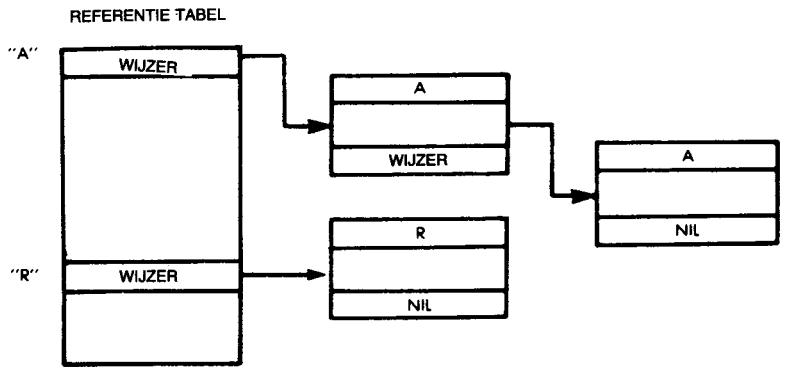


Fig. 9-23: Geketende lijst struktuur

wordt per afspraak een waarde opgeslagen, zodanig dat de absolute waarde van het verschil tussen deze waarde en "Z" groter is dan het verschil tussen "A" en "Z". Dit is de voorstelling van een End Of Table (einde van tabel), afgekort EOT. We nemen hier aan dat deze EOT waarde dezelfde hoeveelheid geheugen in beslag neemt als een gewoon gegeven, maar zou ook wel een byte lang kunnen zijn.

Aangenomen wordt dat de letters alfabetische ASCII karakters zijn. Bij een andere afspraak moet de konstante aan het begin van de PRETAB routine gewijzigd worden.

Het EOT merkteken krijgt de waarde van het begin van de tabel ("NIL").

De afspraak is dat de "NIL" wijzers aan het einde van een string (keten), of in een referentietabellokatie die niet naar een string wijst, de waarde van de basis van de tabel krijgen om zo in een unieke identifikatie te voorzien. Er kan ook een andere afspraak gevolgd worden. Zo zou in het bijzonder een ander merkteken voor de EOT enige ruimte besparen, omdat voor niet bestaande gegevens geen NIL gegevens bijgehouden hoeven te worden.

Invoegen en verwijderen gebeurt op de gebruikelijke manier (zie deel I van dit hoofdstuk) door slechts de vereiste wijzers te veranderen. De INDEXD vlag wordt gebruikt om aan te geven of de wijzer naar een object zich in de referentietabel (directory) of in een ander stringelement bevindt.

Zoeken

Het zoekprogramma SEARCH is geplaatst op geheugenadressen 0600 tot 0650. Verder maakt het gebruik van de subroutine PRETAB op adres 06F8.

Het zoekprincipe is eenvoudig:

1 - Haal het gegeven van de referentietabel die overeenkomt met de letter van het alfabet in de eerste positie van het label van OBJECT.
2 - Haal de wijzer uit de referentietabel. Haal vervolgens het element. Als dit NIL is bestaat het gegeven niet.
3 - Als het niet NIL is, vergelijk het element met OBJECT. Als de vergelijking klopt is de zoekaktie succesvol beeindigd. Haal anders de wijzer naar het volgende gegeven in de lijst.
4 - Ga terug naar 2.

In fig. 9-24 is een voorbeeld gegeven.

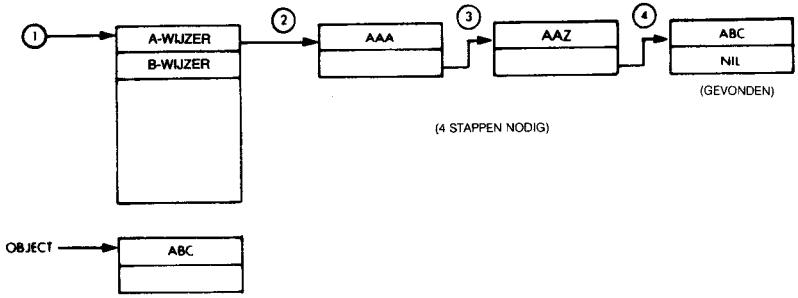


Fig. 9-24: Een geketende lijst: Zoeken

Invoegen van een element

Het invoegen is in wezen een zoekaktie gevolgd door een invoegen zodra er een NIL gevonden wordt. Er wordt een blok geheugen toegewezen na het EOT merkteken door te zoeken naar een merkteken dat de beschikbaarheid aangeeft en dat op "vrij" staat. Het programma heet "NEW" en beslaat adressen 0651 tot 06BD. In fig. 9-25 is een voorbeeld te zien.

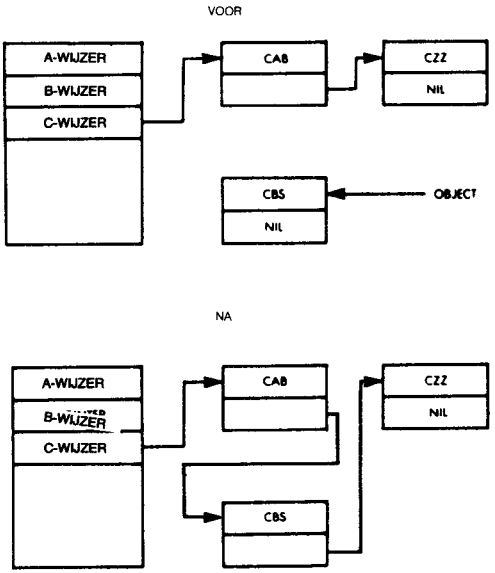


Fig. 9-25: Geketende lijst: voorbeeld van een invoeging

Verwijderen van een element

Het element wordt verwijderd door het merkteken dat de beschikbaarheid aangeeft op "vrij" te zetten, en de wijzer vanuit de referentietabel of het vorige element aan te passen. Het programma heet DELETE en beslaat adressen 06BE tot 06F7. Een voorbeeld van een verwijdering is gegeven in fig. 9-26.

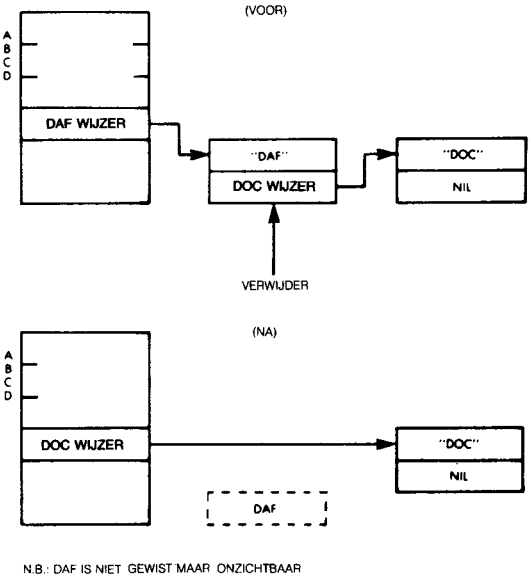


Fig. 9-26: Voorbeeld van een verwijdering (Geketende lijst)

```
LINE # LOC     CODE       LINE

0002  0000              INDEXB = $10
0003  0000              INDLOC = $11
0004  0000              POINTR = $13
0005  0000              OBJECT = $15
0006  0000              TEMP   = $17
0007  0000              REFBAS = $19
0008  0000              OLD    = $1B
0009  0000              TABASE = $1D
0010  0000              ENTLEN = $1F
0011  0000              ;
0012  0000                     * = $6000
0013  0600              ;
0014  0600  A9 01       SEARCH LDA #1          ;INITIALIZE FLAGS
0015  0602  85 10              STA INDEXB
0016  0604  20 F8 06          JSR PRETAB       ;GET REF. POINTER FOR START
0017  0607  B1 11            LDA (INDLOC),Y   ;PUT IT IN POINTR
0018  0609  85 13            STA POINTR
0019  060B  C8               INY
0020  060C  B1 11            LDA (INDLOC),Y
0021  060E  85 14            STA POINTR+1
0022  0610  A0 00       ENTRY  LDY #0          ;SEE IF ENTRY IS EOT VALUE
0023  0612  B1 13            LDA (POINTR),Y
0024  0614  C9 7C            CMP #$7C
0025  0616  F0 36            BEQ NOTFND
0026  0618  D1 15            LDA (OBJECT),Y   ;COMPARE FIRST LETTERS
0027  061A  D1 13            CMP (POINTR),Y
0028  061C  90 30            BCC NOTFND
0029  061E  D0 12            BNE NOGOOD
0030  0620  C8               INY              ;COMPARE SECOND LETTERS
0031  0621  B1 15            LDA (OBJECT),Y
0032  0623  D1 13            CMP (POINTR),Y
0033  0625  90 27            BCC NOTFND
0034  0627  D0 09            BNE NOGOOD
0035  0629  C8               INY              ;COMPARE THIRD LETTERS
0036  062A  B1 15            LDA (OBJECT),Y
0037  062C  D1 13            CMP (POINTR),Y
0038  062E  90 1E            BCC NOTFND
0039  0630  F0 1E            BEQ FOUND
0040  0632  A5 14       NOGOOD LDA POINTR+1    ;SAVE POINTR FOR POSSIBLE REF
0041  0634  85 1C            STA OLD+1
0042  0636  A5 13            LDA POINTR
0043  0638  85 1B            STA OLD
0044  063A  A4 1F            LDY ENTLEN       ;GET POINTER FROM ENTRY AND
0045  063C  B1 13            LDA (POINTR),Y   ;..LOAD IT INTO POINTR
0046  063E  AA               TAX
0047  063F  C8               INY
0048  0640  B1 13            LDA (POINTR),Y
0049  0642  85 14            STA POINTR+1
0050  0644  8A               TXA
0051  0645  85 13            STA POINTR
0052  0647  A9 00            LDA #0
0053  0649  85 10            STA INDEXB       ;RESET FLAG
0054  064B  4C 10 06         JMP ENTRY
0055  064E  A9 FF       NOTFND LDA #$FF
0056  0650  60         FOUND  RTS              ;Z SET IF FOUND
0057  0651              ;
0058  0651              ;
0059  0651              ;
0060  0651  20 00 06    NEW    JSR SEARCH       ;SEE IF OBJ. IS ALREADY THERE
0061  0654  F0 67            BEQ OUTE
0062  0656  A5 1D            LDA TABASE       ;LOOK FOR UNOCCUPIED ENTRY
0063  0658  18               CLC              ;..BLOCK
0064  0659  69 01            ADC #1           ;JUMP PAST EOT VALUE
0065  065B  85 17            STA TEMP
0066  065D  A9 00            LDA #0
0067  065F  65 1E            ADC TABASE+1
0068  0661  85 18            STA TEMP+1
0069  0663  A4 1F            LDY ENTLEN       ;SET Y TO POINT TO OCCUPANCY
```

Fig. 9-27: Programma voor een geketende lijst

```
0070  0665  C8              INY             ;..MARKER OF AN ENTRY
0071  0666  C8              INY
0072  0667  A9 01    LOOP    LDA #1          ;TEST FOR OCCUPANCY MARKER
0073  0669  D1 17            CMP (TEMP),Y
0074  066B  D0 16            BNE INSERT
0075  066D  A5 17            LDA TEMP        ;IF IS USED, MOVE TEMP TO NEXT
0076  066F  18              CLC             ;..ENTRY BLOCK
0077  0670  65 1F            ADC ENTLEN
0078  0672  90 02            BCC MORE
0079  0674  E6 18            INC TEMP+1
0080  0676  69 03    MORE    ADC #3
0081  0678  85 17            STA TEMP
0082  067A  A9 00            LDA #0
0083  067C  65 18            ADC TEMP+1
0084  067E  85 18            STA TEMP+1
0085  0680  4C 67 06          JMP LOOP
0086  0683  88        INSERT  DEY             ;SET Y BACK TO POINTING TO
0087  0684  88                DEY             ;..TOP OF DATA
0088  0685  88        LOPE    DEY             ;MOVE OBJECT INTO SPACE
0089  0686  B1 15            LDA (OBJECT),Y
0090  0688  91 17            STA (TEMP),Y
0091  068A  C0 00            CPY #0
0092  068C  D0 F7            BNE LOPE
0093  068E  A4 1F            LDY ENTLEN      ;PUT THE VALUE OF POINTR, THE
0094  0690  A5 13            LDA POINTR      ;ENTRY AFTER OBJECT, INTO
0095  0692  91 17            STA (TEMP),Y    ;POINTER AREA OF OBJECT
0096  0694  C8              INY
0097  0695  A5 14            LDA POINTR+1
0098  0697  91 17            STA (TEMP),Y
0099  0699  C8              INY
0100  069A  A9 01            LDA #1          ;SET OCCUPANCY MARKER
0101  069C  91 17            STA (TEMP),Y
0102  069E  A5 10            LDA INDEXD      ;TEST TO SEE IF REF. TABLE
0103  06A0  D0 0D            BNE SETINX      ;..NEEDS READJUSTING
0104  06A2  88              DEY
0105  06A3  A5 18            LDA TEMP+1      ;NO, CHANGE PREVIOUS ENTRY'S
0106  06A5  91 1B            STA (OLD),Y     ;..POINTER
0107  06A7  88              DEY
0108  06A8  A5 17            LDA TEMP
0109  06AA  91 1B            STA (OLD),Y
0110  06AC  4C BB 06          JMP DONE
0111  06AF  20 F8 06  SETINX JSR PRETAB      ;GET ADDRESS OF WHATS TO BE CHANGED
0112  06B2  A5 17            LDA TEMP        ;LOAD ADDR. OF OBJ. THERE
0113  06B4  91 11            STA (INDLOC),Y
0114  06B6  C8              INY
0115  06B7  A5 18            LDA TEMP+1
0116  06B9  91 11            STA (INDLOC),Y
0117  06BB  A9 FF    DONE    LDA #$FF
0118  06BB  60        OUTE    RTS             ;Z CLEAR IF DONE
0119  06BE            ;
0120  06BE            ;
0121  06BE            ;
0122  06BE  20 00 06  DELETE  JSR SEARCH      ;GET ADDR OF OBJ.
0123  06C1  D0 34            BNE OUTS
0124  06C3  A4 1F            LDY ENTLEN      ;STORE POINTER AT END
0125  06C5  B1 13            LDA (POINTR),Y  ;..OF OBJECT
0126  06C7  85 17            STA TEMP
0127  06C9  C8              INY
0128  06CA  B1 13            LDA (POINTR),Y
0129  06CC  85 18            STA TEMP+1
0130  06CE  C8              INY
0131  06CF  A9 00            LDA #0          ;CLEAR OCCUPANCY MARKER
0132  06D1  91 13            STA (POINTR),Y
0133  06D3  A5 10            LDA INDEXD      ;SEE IF REF. TABLE NEEDS
0134  06D5  F0 06            BEQ PREINX      ;..READJUSTING
0135  06D7  20 F8 06          JSR PRETAB
0136  06DA  4C EA 06          JMP MOVEIT
0137  06DD  A5 1B    PREINX  LDA OLD         ;SET FOR CHANGING PREVIOUS
0138  06DF  18              CLC             ;..ENTRY
```

Fig. 9-27: Programma voor een geketende lijst (vervolg)

```
0139  06E0  65 1F           ADC ENTLEN
0140  06E2  85 11           STA INDLOC
0141  06E4  A9 00           LDA #0
0142  06E6  65 1C           ADC OLD+1
0143  06E8  85 12           STA INDLOC+1
0144  06EA  A5 17     MOVEIT LDA TEMP        ;CHANGE WHAT NEEDS CHANGING
0145  06EC  A0 00           LDY #0
0146  06EE  91 11           STA (INDLOC),Y
0147  06F0  C8              INY
0148  06F1  A5 18           LDA TEMP+1
0149  06F3  91 11           STA (INDLOC),Y
0150  06F5  A9 00           LDA #0
0151  06F7  60        OUTS   RTS             ;Z SET IF DONE
0152  06F8              ;
0153  06F8              ;
0154  06F8              ;
0155  06F8  A0 00     PRETAB LDY #0
0156  06FA  B1 15           LDA (OBJECT),Y
0157  06FC  38              SEC              ;REMOVE ASCII LEADER FROM
0158  06FD  E9 41           SBC #041         ;..FIRST LETTER IN OBJECT
0159  06FF  0A              ASL A            ;MULTIPLY BY 2
0160  0700  18              CLC
0161  0701  65 19           ADC REFBAS       ;INDEX INTO REF. TABLE
0162  0703  85 11           STA INDLOC
0163  0705  A9 00           LDA #0
0164  0707  65 1A           ADC REFBAS+1
0165  0709  85 12           STA INDLOC+1
0166  070B  60              RTS
0167  070C                 .END

ERRORS = 0000 <0000
```

```
SYMBOL TABLE

SYMBOL   VALUE

DELETE   06BE   DONE     06BB   ENTLEN   001F   ENTRY    0610
FOUND    0650   INDEXD   0010   INDLOC   0011   INSERT   0683
LOOP     0667   LOPE     0685   MORE     0676   MOVEIT   06EA
NEW      0651   NOGOOD   0632   NOTFND   064E   OBJECT   0015
OLD      001B   OUTE     06DD   OUTS     06F7   POINTR   0013
PREINX   06DB   PRETAB   06F8   REFBAS   0019   SEARCH   0600
SETINX   06AF   TABASE   001B   TEMP     0017
END OF ASSEMBLY
```

Fig. 9-27: Programma voor een geketende lijst (vervolg)

BINAIRE BOOM

We zullen nu drie veel voorkomende routines ontwikkelen om bomen te behandelen. Onze eenvoudige struktuur is te zien in fig. 9-28. Het is een binaire boom en de knooppunten zijn namen van personen. Namen worden intern gesorteerd naar zogenaamde "tags" (aanhangsels), die gevormd worden door de eerste drie letters van iedere naam. De geheugenrepresentatie van deze boomstruktuur is getekend in fig. 9-29. De inhoud van de knooppunten is te zien, evenals de twee verbindingen. De eerste verbinding, links van de naam, is de "linker afstammeling" en de volgende verbinding, aan de rechterkant, is de "rechter afstammeling". Zo bevat het gegeven voor Dijkstra bijvoorbeeld twee verbindingen: "2" en "4". Dit geeft aan dat de linker afstammeling gegeven nummer 2 (Janssens) is, en de rechter afstammeling gegeven nummer 4 (Van Loon) is. Een "0" in het verbindingsveld geeft aan dat er geen afstammeling is. De tag van een linker afstammeling komt alfabetisch voor zijn ouder. De tag van de rechter afstammeling komt erna.

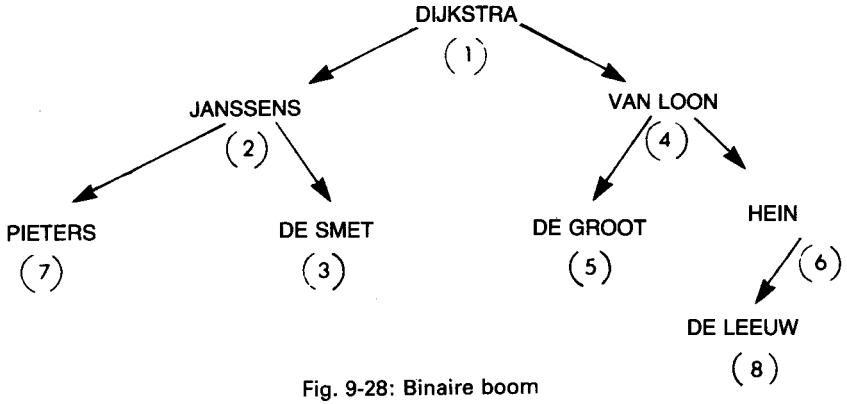


Fig. 9-28: Binaire boom

De twee voornaamste routines om een boom te behandelen zijn die om een boom op te bouwen en die om de boom te doorlopen. Het element dat ingevoegd moet worden wordt in een buffer geplaatst. De routine om de boom op te bouwen plaatst de inhoud van de buffer in het juiste knooppunt. De routine om de boom te doorlopen doet dit recursief en drukt de inhoud van alle knooppunten in alfanumerieke volgorde af. Het stroomdiagram voor de routine om de boom op te bouwen is gegeven in fig. 9-30 en de routine om de boom te doorlopen ingegeven in fig. 9-31.

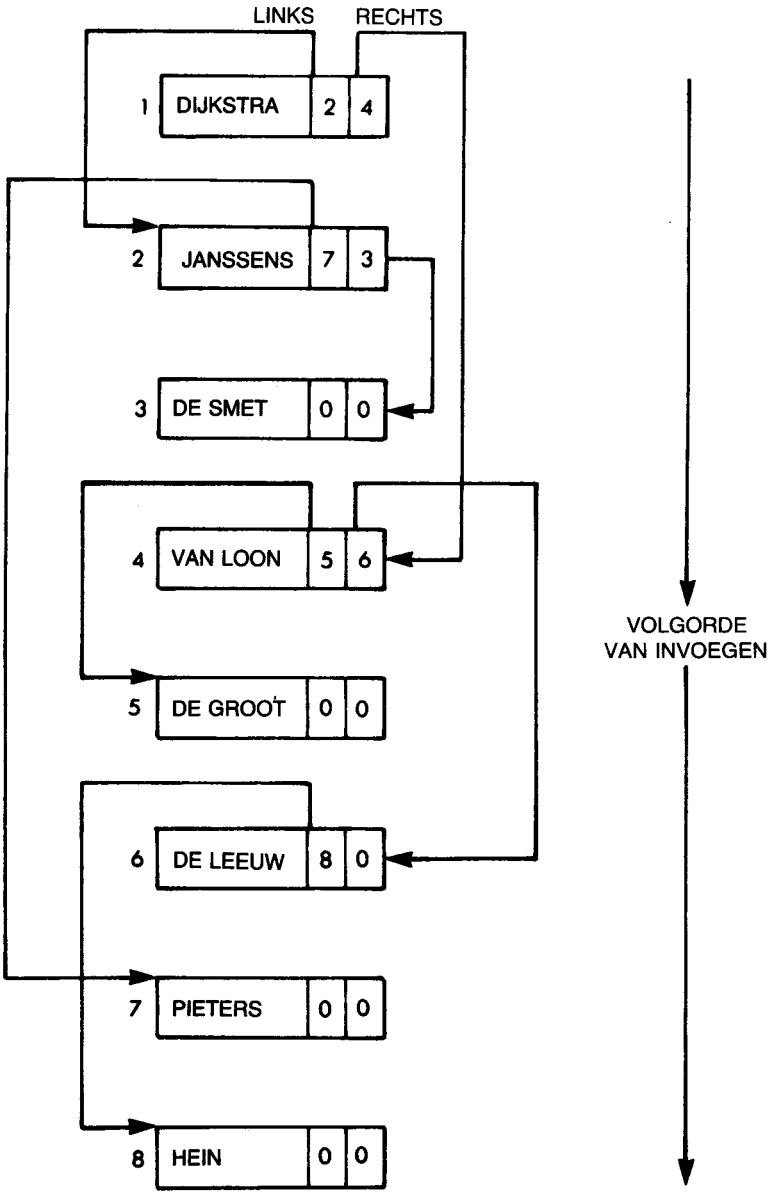


Fig. 9-29: Representatie in het geheugen

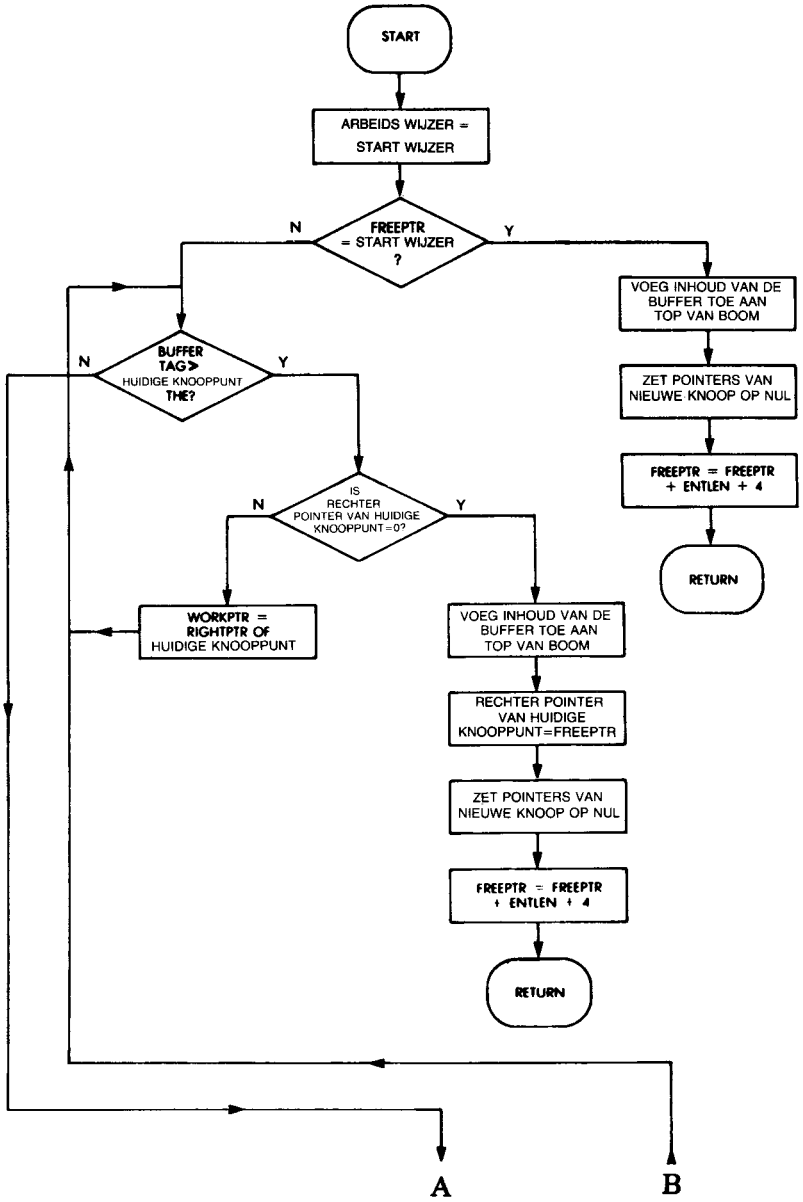


Fig. 9-30: Stroomdiagram voor het opbouwen van de boom

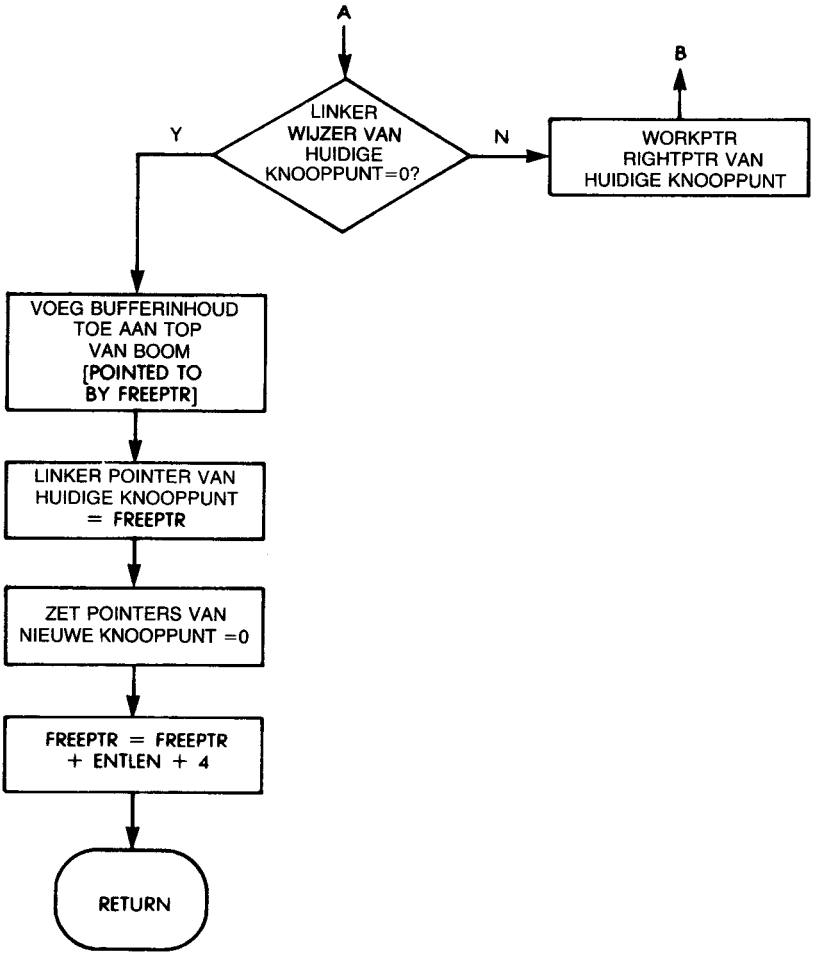


Fig. 9-30: Stroomdiagram voor het opbouwen van de boom (vervolg)

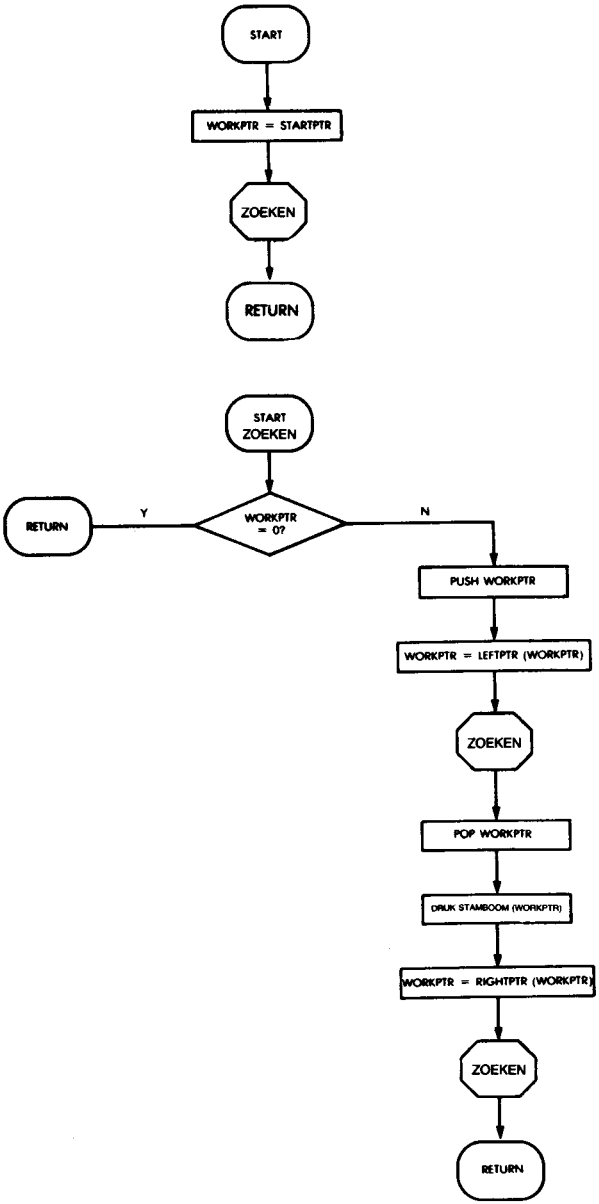


Fig. 9-31: Stroomdiagram om een boom te doorlopen

Omdat de routine voor het doorlopen van de boom recursief is, leent deze zich niet erg om door middel van een stroomdiagram voorgesteld te worden. Een andere beschrijving van de routine op het niveau van een hogere (programmeer)taal is daarom getoond in fig. 9.32. Een knooppunt van de boom is getoond in fig. 9-33. Het bevat gegevens voor de lengte ENTLEN, vervolgens twee 16-bits wijzers (de linker en de rechter wijzer). Om mogelijke verwarring te vermijden moet er op gelet worden dat de voorstelling van fig. 9-29 vereenvoudigd is en dat de rechter wijzer links van de linker wijzer in het geheugen verschijnt. De geheugentoewijzing die door dit programma gebruikt wordt is te zien in fig. 9-34 en het programma in fig. 9-37.

De INSERT (invoegen) routine beslaat adressen 0200 tot 0282. De tag van het object dat ingevoegd moet worden, wordt vergeleken met die van het gegeven. Als die groter is ga je naar rechts, als die kleiner is naar links, een positie naar beneden. Het proces wordt herhaald tot er een lege verbinding gevonden wordt of een geschikte "koppeling" gevonden wordt voor het nieuwe knooppunt (d.w.z. een knoop is groter en de volgende kleiner of omgekeerd). Het nieuwe knooppunt wordt dan ingevoegd door de juiste verbindingen te maken.

```
PROGRAM TREETRAVERSER;
BEGIN
      CALL SEARCH (STARTPOINTER);
END.

ROUTINE SEARCH (WORKPOINTER);
BEGIN
      IF WORKPOINTER = 0 THEN RETURN;
      SEARCH [LEFTPTR (WORKPOINTER)];
      PRINT TREE (WORKPOINTER);
      SEARCH [RIGHTPTR (WORKPTR)];
      RETURN;
END.
```

Fig. 9-32: Algoritme om een boom te doorlopen (TREETRAVERSAL)

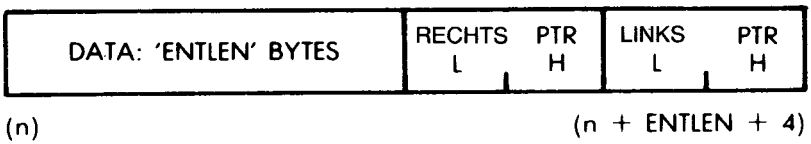


Fig. 9-33: Gegevenseenheden, of "knopen" van een boom

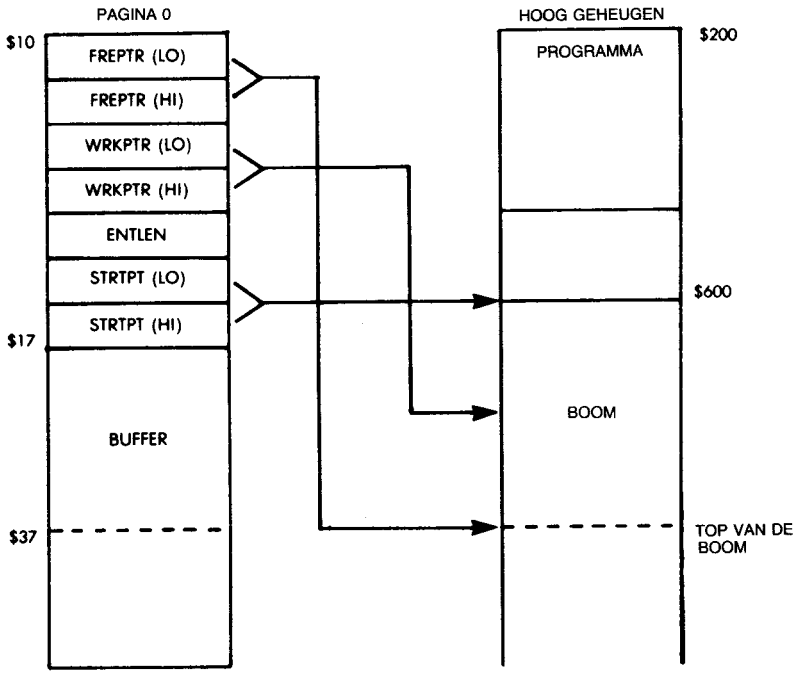


Fig. 9-34: Geheugenindeling

De TRAVERSE (doorlopen) routine beslaat adressen 0285 tot 02D6. De utiliteitsroutines OUT, ADD en CLRPTR staan op adressen 0207 tot 02FE (zie fig. 9-37).

Een voorbeeld van een invoeging in een boom is te zien in fig. 9-35, en een voorbeeld van het doorlopen van een boom in fig. 9-36.

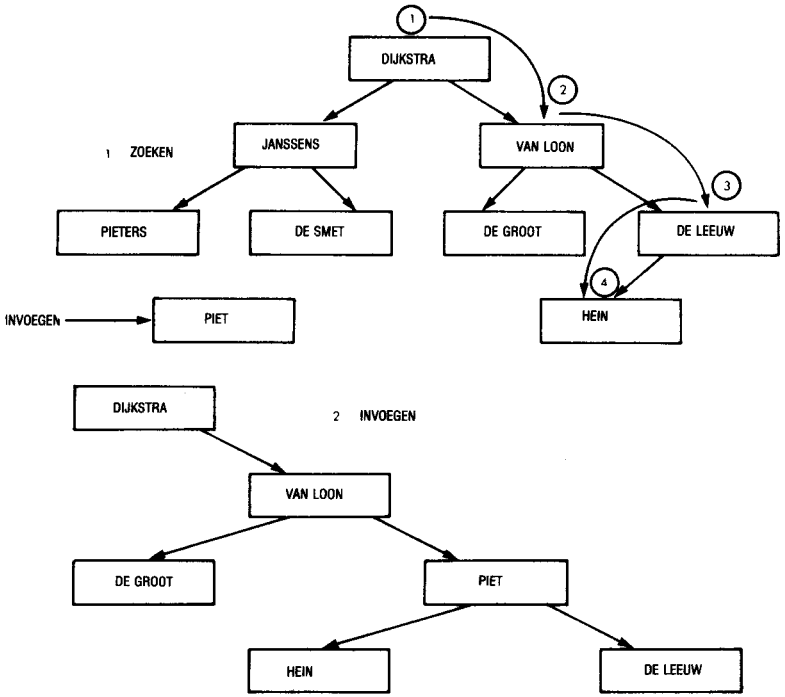


Fig. 9-35: Invoegen van een element in de boom

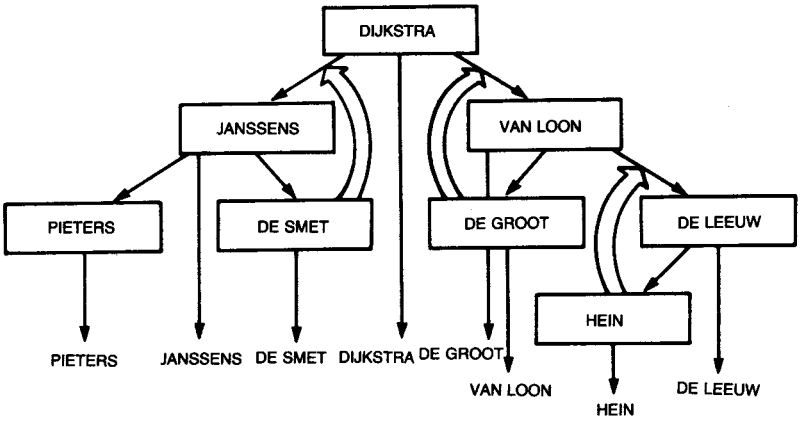


Fig. 9-36: boom

Opmerkingen t.a.v. bomen

Binaire bomen kunnen op vele manieren doorlopen worden. Een andere voorstelling van onze boom zou kunnen zijn:

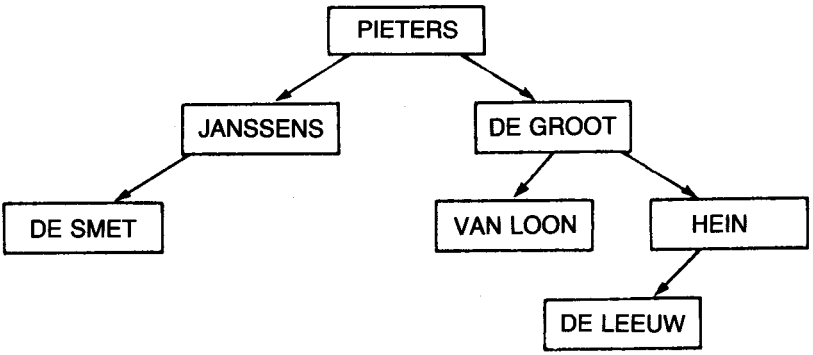


Fig. 9-38: Boom in "pre-orde":

Hij zou dan in "pre-orde" doorlopen moeten worden:

1 - druk de wortel af
2 - doorloop linker subboom
3 - doorloop rechter subboom

Er zijn meerdere technieken en afspraken.

```
0002  0000                ;TREE MANAGEMENT PROGRAM.
0003  0000                ;2 ROUTINES: ONE, WHEN CALLED, PLACES
0004  0000                ;THE CONTENTS OF THE BUFFER INTO THE
0005  0000                ;TREE; AND THE SECOND TRAVERSES
0006  0000                ;THE TREE RECURSIVELY, PRINTING ITS
0007  0000                ;NODE CONTENTS IN ALPHANUMERIC ORDER.
0008  0000                ;NOTE: 'ENTLEN' MUST BE INITIALIZED
0009  0000                ;AND 'FREPTR' MUST BE SET EQUAL TO
0010  0000                ;'STRTPTR' BEFORE EITHER ROUTINE IS USED.
0011  0000                ;
0012  0000                      * = $10
0013  0010                FREPTR *=*+2          ;FREE SPACE POINTER: POINTS TO
0014  0012                         ;NEXT FREE LOCATION IN MEMORY.
0015  0012                WRKPTR *=*+2          ;WORKING POINTER, POINTS TO CURRENT NODE.
0016  0014                ENTLEN *=*+1          ;TREE ENTRY LENGTH, IN BYTES.
0017  0015  00 06         STRTPT .WORD $600
0018  0017                BUFFER *=*+20         ;I/O BUFFER.
0019  002B                ;
0020  002B                      * = $200
0021  0200                ;
0022  0200                ;ROUTINE TO BUILD TREE: ADDS ONE DATA UNIT,
0023  0200                ;OR NODE, TO TREE. MUST BE CALLED
0024  0200                ;WITH DATA UNIT TO BE ADDED IN 'BUFFER'.
0025  0200                ;
0026  0200  A5 15         INSERT LDA STRTPT     ;WORKPOINTER := FREEPOINTER.
0027  0202  85 12                STA WRKPTR
0028  0204  A5 16                LDA STRTPT+1
0029  0206  85 13                STA WRKPTR+1
0030  0208  A5 10                LDA FREPTR      ;IF FREEPOINTER <>
0031  020A  C5 15                CMP STRTPT      ;STARTING LOCATION POINTER,
0032  020C  D0 0D                BNE INLOOP      ;GOTO INSERTION LOOP.
0033  020E  A5 11                LDA FREPTR+1
0034  0210  C5 16                CMP STRTPT+1
0035  0212  D0 07                BNE INLOOP
0036  0214  20 D7 02             JSR ADD         ;LOAD BUFFER INTO CURRENT POSITION.
0037  0217  20 E4 02             JSR CLRPTR      ;SET POINTERS OF CURRENT NODE TO 0.
0038  021A  60                   RTS             ;DONE ADDING 1ST NODE.
0039  021B  A0 00         INLOOP LDY #0          ;COMPARE BUFFER TAG TO TAG OF CURRENT
0040  021D  B9 17 00      CMPLP  LDA BUFFER,Y    ;LOCATION...
0041  0220  D1 12                CMP (WRKPTR),Y
0042  0222  90 33                BCC LESSTH      ;BUFR TAG LOWER: ADD BUFFER TO
0043  0224                            ;LEFT SIDE OF TREE.
0044  0224  F0 02                BEQ NXT         ;TAGS EQUAL, TRY NEXT CHR. IN TAGS.
0045  0226  B0 05                BCS GRTHEQ      ;BUFR TAG GREATER, ADD BUFR TO
0046  0228                          ;RIGHT SIDE OF TREE.
0047  0228  C8                NXT    INY
0048  0229  C9 04                CMP #4          ;3 CHRS. COMPARED?
0049  022B  D0 F0                BNE CMPLP       ;NO, CHECK NEXT CHR.
0050  022D  A4 14         GRTHEQ LDY ENTLEN      ;DOES
0051  022F  D1 12                LDA (WRKPTR),Y  ;RIGHT POINTER OF CURRENT NODE = 0 ?
0052  0231  D0 15                BNE NXRNOD      ;IF NOT, MOVE DOWN/RIGHT IN TREE.
0053  0233  C8                   INY
0054  0234  D1 12                LDA (WRKPTR),Y
0055  0236  D0 10                BNE NXRNOD
0056  0238  A5 11                LDA FREPTR+1    ;SET RIGHT POINTER OF CURRENT
0057  023A  91 12                STA (WRKPTR),Y  ;NODE = FREEPOINTER.
0058  023C  88                   DEY
0059  023D  A5 10                LDA FREPTR
0060  023F  91 12                STA (WRKPTR),Y
0061  0241  20 D7 02             JSR ADD         ;ADD BUFFER TO TREE.
0062  0244  20 E4 02             JSR CLRPTR      ;CLEAR POINTERS OF NEW NODE.
0063  0247  60                   RTS             ;DONE, NEW RIGHT NODE ADDED.
0064  0248  A4 14         NXRNOD LDY ENTLEN      ;SET WORKING POINTER
0065  024A  B1 12                LDA (WRKPTR),Y; RIGHT POINTER OF CURRENT NODE.
0066  024C  AA                   TAX
0067  024D  C8                   INY
0068  024E  B1 12                LDA (WRKPTR),Y
0069  0250  85 13                STA WRKPTR+1
0070  0252  86 12                STX WRKPTR
0071  0254  4C 1B 02             JMP INLOOP      ;TRY NEW CURRENT NODE.
```

Fig. 9-37: Programma's om in een boom te zoeken

```
0072  0257  A4 14     LESSTN LDY ENTLEN      ;DOES LEFT POINTER OF
0073  0259  C8                INY             ;CURRENT NODE = 0 ?
0074  025A  C8                INY
0075  025B  B1 12              LDA (URKPTR),Y
0076  025D  D0 15              BNE NXLNOD      ;IF S0, MOVE DOWN/LEFT IN TREE.
0077  025F  C8                INY
0078  0260  B1 12              LDA (URKPTR),Y
0079  0262  D0 10              BNE NXLNOD
0080  0264  A5 11              LDA FREPTR+1    ;SET LEFT POINTER OF CURRENT NODE TO
0081  0266  91 12              STA (URKPTR),Y  ;POINT TO NEW NODE.
0082  0268  88                DEY
0083  0269  A5 10              LDA FREPTR
0084  026B  91 12              STA (URKPTR),Y
0085  026D  20 B7 02           JSR ADD         ;ADD NEW NODE CONTENTS.
0086  0270  20 E4 02           JSR CLRPTR      ;CLEAR POINTERS OF NEW NODE.
0087  0273  60                RTS             ;DONE, NEW LEFT NODE ADDED.
0088  0274  A4 14     NXLNOD LDY ENTLEN      ;SET WORKING POINTER =
0089  0276  C8                INY             ;LEFT POINTER OF CURRENT NODE.
0090  0277  C8                INY
0091  0278  B1 12              LDA (URKPTR),Y
0092  027A  AA                TAX
0093  027B  C8                INY
0094  027C  B1 12              LDA (URKPTR),Y
0095  027E  85 13              STA URKPTR+1
0096  0280  86 12              STX URKPTR
0097  0282  4C 1B 02           JMP INLOOP      ;TRY NEW CURRENT NODE.
0098  0285                    ;
0099  0285                    ;TREE TRAVERSER : LISTS NODES OF TREE
0100  0285                    ;IN ALPHANUMERICAL ORDER.
0101  0285                    ;OUTPUT ROUTINE TO XFER BUFFER TO OUTPUT
0102  0285                    ;DEVICE IS NEEDED.
0103  0285                    ;
0104  0285  A5 15     TRVRSE LDA STRTPT      ;WORKING POINTER <= START POINTER.
0105  0287  85 12             STA URKPTR
0106  0289  A5 16             LDA STRTPT+1
0107  028B  85 13             STA URKPTR+1
0108  028D  A5 13     SEARCH LDA URKPTR+1
0109  028F  A6 12             LDX URKPTR      ;IF WORKING POINTER <> 0,
0110  0291  D0 07             BNE OK          ;CONTINUE;
0111  0293  A4 13             LDY URKPTR+1
0112  0295  D0 03             BNE OK
0113  0297  4C C6 02          JMP RETN        ;ELSE, RETURN.
0114  029A  48        OK     PHA             ;PUSH WORKING POINTER
0115  029B  8A                TXA             ;ONTO STACK.
0116  029C  48                PHA
0117  029D  A4 14             LDY ENTLEN      ;SET WORKING POINTER =
0118  029F  C8                INY             ;LEFT POINTER OF CURRENT NODE.
0119  02A0  C8                INY
0120  02A1  B1 12             LDA (URKPTR),Y
0121  02A3  AA                TAX
0122  02A4  C8                INY
0123  02A5  B1 12             LDA (URKPTR),Y
0124  02A7  85 13             STA URKPTR+1
0125  02A9  86 12             STX URKPTR
0126  02AB  20 8D 02          JSR SEARCH      ;SEARCH NEW NODE, RECURSIVELY.
0127  02AE  68                PLA             ;POP OLD CURRENT NODE INTO WORKING POINTER.
0128  02AF  85 12             STA URKPTR
0129  02B1  68                PLA
0130  02B2  85 13             STA URKPTR+1
0131  02B4  20 C7 02          JSR OUT         ;OUTPUT CURRENT NODE CONTENTS.
0132  02B7  A4 14             LDY ENTLEN      ;SET WORKING POINTER =
0133  02B9  B1 12             LDA (URKPTR),Y  ;CURRENT NODE'S RIGHT POINTER.
0134  02BB  AA                TAX
0135  02BC  C8                INY
0136  02BD  B1 12             LDA (URKPTR),Y
0137  02BF  85 13             STA URKPTR+1
0138  02C1  86 12             STX URKPTR
0139  02C3  20 8D 02          JSR SEARCH      ;SEARCH NEW NODE.
0140  02C6  60        RETN   RTS             ;DONE, RETURN.
```

Fig. 9-37: Programma's om in een boom te zoeken (vervolg)

```
0141  02C7             ;
0142  02C7             ;BUFFER OUTPUT ROUTINE.
0143  02C7             ;
0144  02C7  A0 00       OUT    LDY #0
0145  02C9  B1 12       XFR    LDA (WRKPTR),Y  ;GET CHR. FROM CURRENT MODE.
0146  02CB  99 17 00           STA BUFFER,Y    ;PUT IN BUFFER.
0147  02CE  C8                 INY             ;REPEAT UNTIL...
0148  02CF  C4 14              CPY ENTLEN      ;ALL CHARACTERS XFERRED.
0149  02D1  D0 F6              BNE XFR
0150  02D3  EA                 NOP             ;INSERT CALL TO SUBROUTINE
0151  02D4  EA                 NOP             ;WHICH OUTPUTS BUFFER HERE.
0152  02D5  EA                 NOP
0153  02D6  60                 RTS             ;DONE.
0154  02D7             ;
0155  02D7             ;ROUTINE WHICH PLACES BUFFER
0156  02D7             ;CONTENTS IN NEW MODE.
0157  02D7             ;
0158  02D7  A0 00       ADD    LDY #0
0159  02D9  B9 17 00    MOV    LDA BUFFER,Y    ;GET CHR. FROM BUFFER.
0160  02DC  91 10              STA (FREPTR),Y  ;STORE IN NEW MODE.
0161  02DE  C8                 INY             ;REPEAT UNTIL...
0162  02DF  C4 14              CPY ENTLEN      ;ALL CHRS XFERRED.
0163  02E1  D0 F6              BNE MOV
0164  02E3  60                 RTS             ;DONE.
0165  02E4             ;
0166  02E4             ;ROUTINE TO CLEAR POINTERS OF NEW MODE,
0167  02E4             ;AND UPDATE FREE SPACE POINTER.
0168  02E4             ;
0169  02E4  A4 14       CLRPTR LDY ENTLEN      ;SET UP INDEX TO POINT
0170  02E6                                    ;TO TOP OF POINTER LOCATIONS.
0171  02E6  A9 00              LDA #0
0172  02E8  A2 04              LDX #4          ;LOOP 4X TO CLEAR POINTERS
0173  02EA  91 10       CLRLP  STA (FREPTR),Y  ;CLEAR POINTER LOCATION.
0174  02EC  C8                 INY             ;POINT TO NEXT POINTER LOCATION.
0175  02ED  CA                 DEX
0176  02EE  D0 FA              BNE CLRLP       ;LOOP IF NOT DONE.
0177  02F0  A5 14              LDA ENTLEN      ;GET ENTRY LENGTH,
0178  02F2  18                 CLC             ;AND ADD 4 FOR POINTER SPACE.
0179  02F3  69 04              ADC #4
0180  02F5  65 10              ADC FREPTR      ;ADD TO FREE SPACE POINTER TO
0181  02F7  90 02              BCC CC          ;UPDATE IT.
0182  02F9  E6 11              INC FREPTR+1    ;TAKE CARE OF OVERFLOWS.
0183  02FB  85 10       CC     STA FREPTR      ;RESTORE UPDATED FREE SPACE PTR.
0184  02FD  60                 RTS             ;DONE.
0185  02FE                     .END

ERRORS = 0000 <0000>
END OF ASSEMBLY
```

TOEGANGS-TIJD

Fig. 9-37: Programma's om in een boom te zoeken (vervolg)

EEN HASH ALGORITME

Een veel voorkomend probleem bij het opzetten van gegevensstrukturen is het op een systematische wijze in een beperkte hoeveelheid geheugen plaatsen van identifiers (waarmee een element geidentificeerd wordt), zo dat er gemakkelijk toegang tot gekregen kan worden. Tenzij de identifiers unieke opeenvolgende getallen zijn, lenen ze zich er helaas niet voor om zonder gaten in het geheugen geplaatst te worden. Als bijvoorbeeld namen op een zodanige manier in het geheugen geplaatst zouden worden, dat er gemakkelijk toegang tot te krijgen is (dus als ze alfabetisch opgeslagen zijn),zou dit een enorme hoeveelheid geheugen vergen; voor iedere mogelijke naam zou een apart geheugenblok gereserveerd moeten worden. Het is duidelijk dat dit niet acceptabel is. Om dit probleem op te lossen kan een zogenaamd hashing-algoritme (verdeel-algoritme) gebruikt worden om een uniek (of bijna uniek) getal toe te wijzen aan iedere naam die in het geheugen geplaatst moet worden. De wiskundige funktie die deze verdeling verricht moet eenvoudig zijn, zodat het algoritme snel is en toch geraffineerd genoeg om de mogelijke namen over het geheugen te verdelen. Het resulterende getal kan dan gebruikt worden als een index naar de eigenlijke plaats, zodat snelle toegang mogelijk is. Om deze reden wordt hashing meestal gebruikt bij lijsten met alfabetische namen.

Omdat geen enkel algoritme kan garanderen dat niet twee elementen in dezelfde geheugenplaats terecht komen ("botsen"), moet een techniek ontworpen worden om het probleem van de botsingen op te lossen. Een goed hashing-algoritme verdeelt de namen gelijkmatig over de beschikbare geheugenruimte, en zorgt voor snelle toegang tot hun waarde als ze eenmaal in een tabel opgeslagen zijn. De hashtechniek die we hier gebruiken is erg eenvoudig; we nemen de exclusive OR van alle bytes van de sleutelwaarde. Om de gelijkmatigheid te vergroten roteren we na iedere optelling.

De techniek die we gebruiken om botsingen te voorkomen is een eenvoudige, en wordt wel een "progressieve open adresseertechniek" genoemd: het volgende beschikbare blok in het geheugen wordt aan het gegeven toegewezen. Dit is te vergelijken met een zakadresboekje. Laten we aannemen dat we een nieuw gegeven moeten invoegen voor VAN LOON. De "S" pagina in ons adresboekje is echter vol. We gebruiken dan de volgende pagina (in dit geval de "T" pagina). Merk op dat er nu niet een botsing met een nieuw gegeven die met "T" begint hoeft op te treden; het gegeven voor "S" kan verwijderd worden (uitgegumd worden in onze vergelijking) voordat een "T" ingevoerd moet worden.

Merk op dat er ook een kettingbotsing kan optreden. Als de ketting lang is en de tabel niet vol is, is het hash-algoritme een slecht ontwerp.

Omdat het handig is om voor het gegevensformaat een macht van 2 te gebruiken, is de lengte van de gegevens 8 karakters; zes voor de sleutel (identifier) en 2 voor de gegevens. Dit is een veel voorkomende situatie bij het opzetten van bijvoorbeeld een symbooltabel van een assembler. Maximaal zes hexadecimale symbolen zijn aan het symbool toegewezen, en twee worden gebruikt voor het adres dat het voorstelt (2 bytes).

Bij het verkrijgen van toegang tot de elementen van een hashtabel hangt de tijd die voor de zoekaktie nodig is niet van de grootte van de tabel af, maar van de vullingsgraad. Door de tabel voor minder dan 80% te vullen, is een snelle toegangstijd verzekerd (een of twee pogingen). Het is de verantwoordelijkheid van de aanroepende routine om de vullingsgraad van de tabel bij te houden en overstroming te voorkomen.

De toename van de toegangstijd met de vullingsgraad is te zien in fig. 9-39. De voornaamste routines die dit programma gebruikt zijn de initialisatie-routine (INIT), fig. 9-40; de opberg-routine (STORE), fig. 9-41; de ophaal-routine, fig. 9-42; en de hash-routine die in fig. 9-43 te zien is. De geheugenindeling is te zien in fig. 9-44, het programma is gegeven in fig. 9-45. Het programma is bedoeld om de voornaamste algoritmen, die in een hash-mechanisme gebruikt worden, te demon-

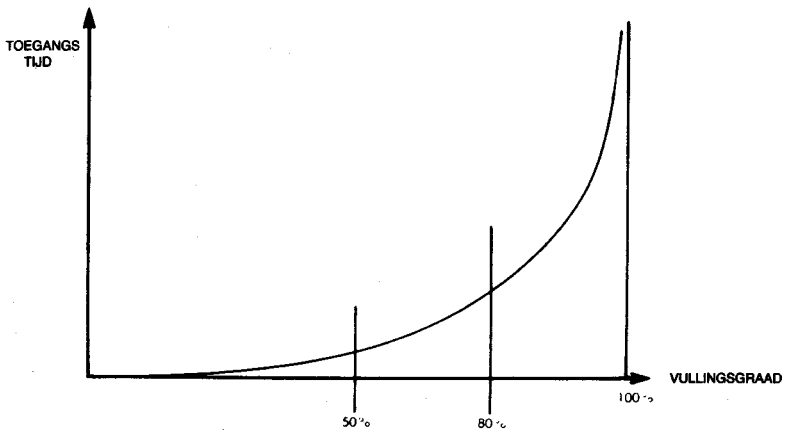


Fig. 9-39: Toegangstijd tegen vullingsgraad

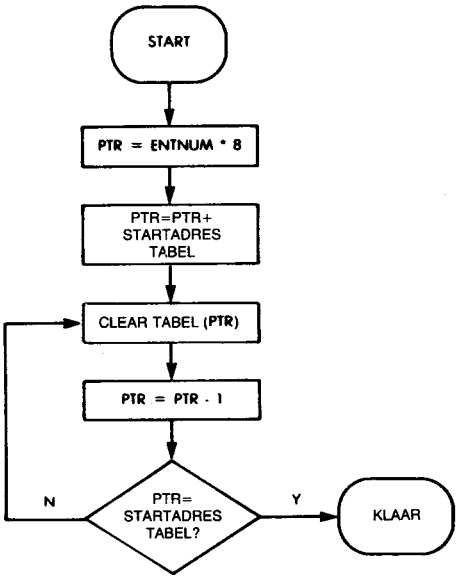


Fig. 9-40: Initialisatie-subroutine

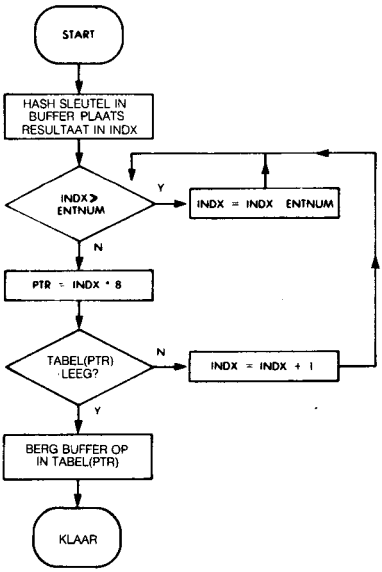


Fig. 9-41: Opberg-routine, "STORE"

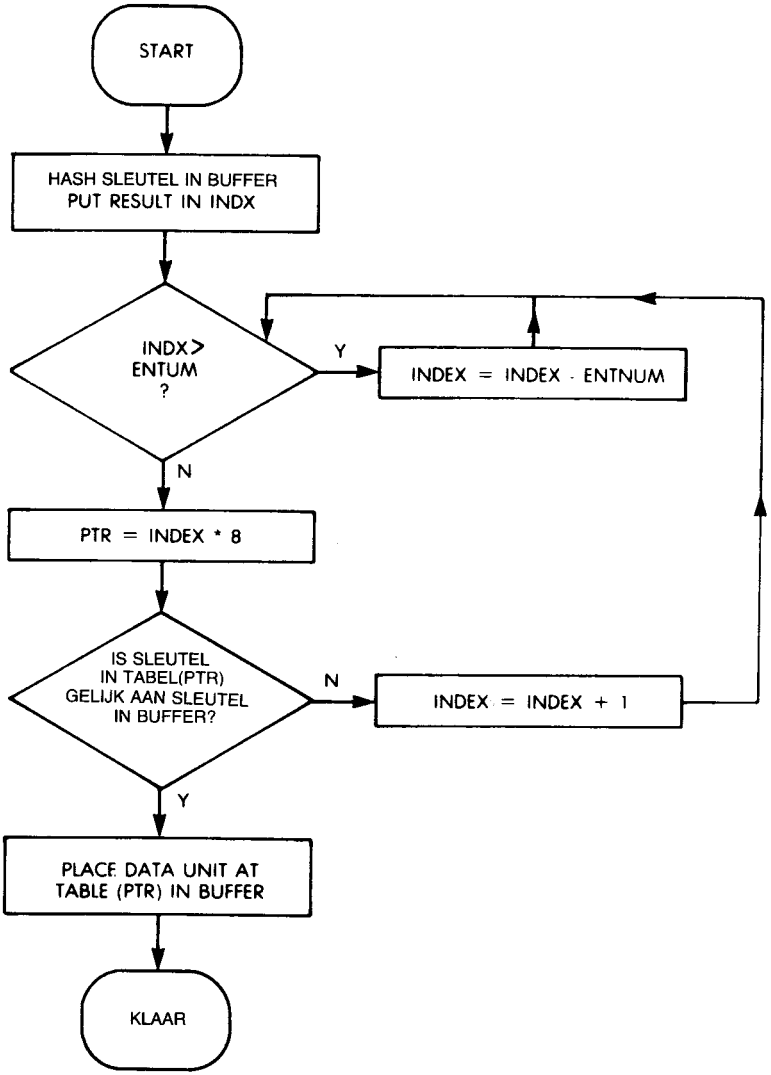


Fig. 9-42: Ophaal-routine, "FIND"

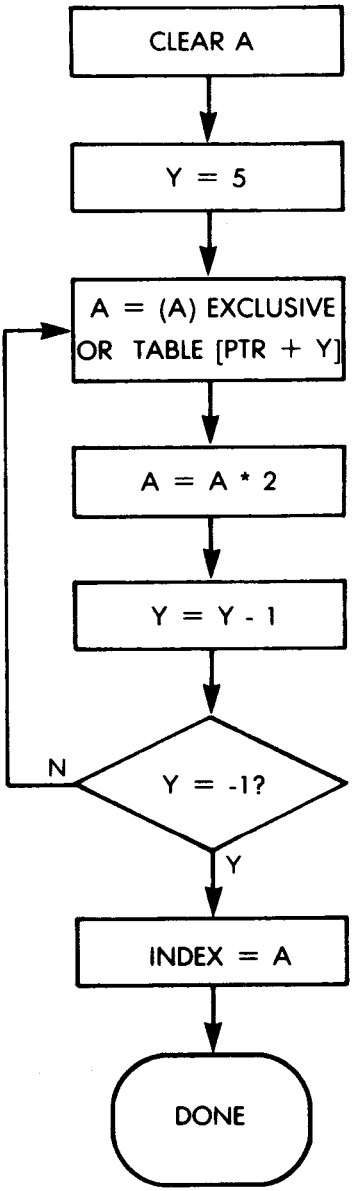


Fig. 9-43: Hash-routine

streren. Als deze programma's in een "echte" toepassing gebruikt worden, wordt ten sterkste aangeraden om de huishoudfunkties, die nodig zijn om onverwachtte situaties te voorkomen, toe te voegen.

Er moet in het bijzonder gelet worden op een volle tabel en een foute sleutel, omdat deze mogelijkheden een oneindige lus in het programma kunnen veroorzaken. De lezer wordt ten sterkste aangeraden om dit programma te bestuderen. Het zal niet alleen de geheimzinnigheid van een hash-algoritme opheffen, maar ook een belangrijk praktisch probleem oplossen dat we hebben als er een assembler ontworpen moet worden, of een andere struktuur waarbij tabellen met namen met hun equivalente waarden op een efficiente manier opgeslagen moeten worden.

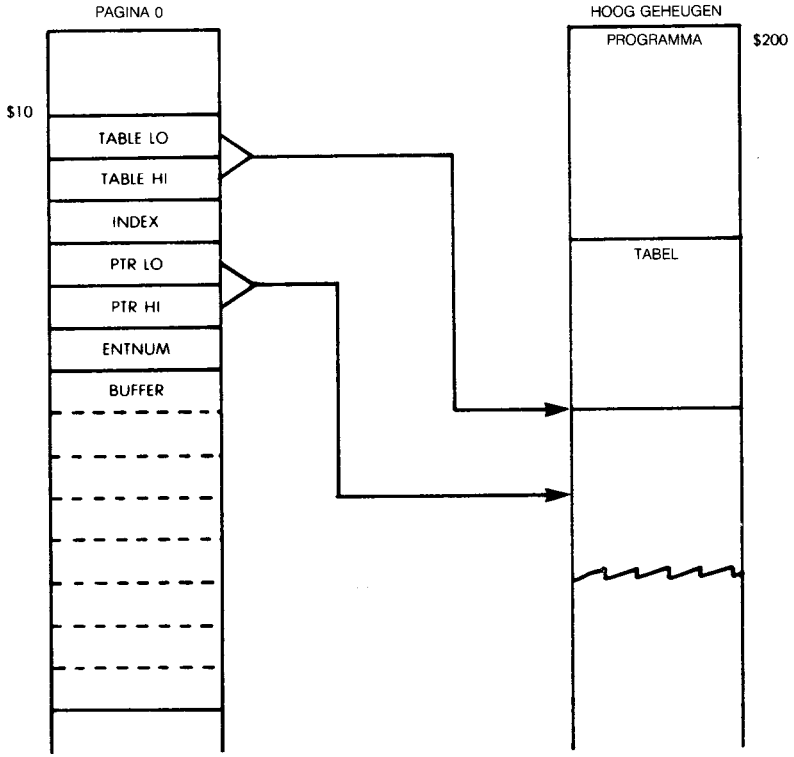


Fig. 9-44: Hash store/retrieve (opbergen/ophalen): geheugenindeling

```
LINE # LOC    CODE       LINE

0002  0000              ;PROGRAM TO STORE ASSEMBLER SYMBOLS IN A
0003  0000              ;TABLE, ACCESSED BY HASHING. THE SYMBOLS
0004  0000              ;ARE 6 CHRS, DATA 2. THE MAXIMUM NUMBER OF
0005  0000              ;8-BYTE UNITS TO BE STORED IN THE TABLE
0006  0000              ;SHOULD BE IN 'ENTNUM', BEGINNING ADDRESS OF
0007  0000              ;TABLE SHOULD BE IN 'TABLE'. NOTE THAT
0008  0000              ;TABLE MUST BE INITIALIZED WITH ROUTINE
0009  0000              ;'INIT' PRIOR TO USE.
0010  0000              ;IT IS THE RESPONSIBILITY OF THE CALLING
0011  0000              ;PROGRAM NO TO EXCEED THE TABLE SIZE.
0012  0000              ;
0013  0000                    * = $10
0014  0010  00 06       TABLE  .WORD $600      ;STARTING ADDRESS OF TABLE.
0015  0012              INDX   *=*+1           ;NUMBER OF DATA UNIT TO BE ACCESSED.
0016  0013              PTR    *=*+2           ;POINTER TO DATA UNIT IN TABLE.
0017  0015              ENTNUM *=*+1           ;NUMBER OF ENTRIES IN TABLE (256 MAX)
0018  0016              BUFFER *=*+8           ;INPUT/ OUTPUT BUFFER.
0019  001E              ;
0020  001E                    * = $200
0021  0200              ;
0022  0200              ;ROUTINE 'INIT' : INITIALIZES TABLE
0023  0200              ;TO ZEROES.
0024  0200              ;
0025  0200  A5 15       INIT   LDA ENTNUM
0026  0202  85 13              STA PTR          ;STORE # OF ENTRIES IN POINTER
0027  0204  20 72 02           JSR SHADD        ;MULTIPLY PTR*8, ADD TABLE POINTER.
0028  0207  A2 00              LDX #0           ;CLEAR X FOR INDIRECT ADDRESSING.
0029  0209  A9 00       CLRLP  LDA #0           ;GET CLEARING CONSTANT
0030  020B  A4 13              LDY PTR
0031  020D  D0 02              BNE DECR         ;IF PTR <> 0, DON'T DECREMENT HI BYTE.
0032  020F  C6 14              DEC PTR+1        ;DECREMENT HI BYTE OF POINTER.
0033  0211  C6 13       DECR   DEC PTR          ;DECREMENT LO BYTE.
0034  0213  81 13              STA (PTR,X)      ;CLEAR LOCATION.
0035  0215  A5 13              LDA PTR          ;CHECK IF POINTER = TABLE POINTER,
0036  0217  C5 10              CMP TABLE        ;IF UNEQUAL, CLEAR NEXT LOCATION.
0037  0219  D0 EE              BNE CLRLP
0038  021B  A5 14              LDA PTR+1
0039  021D  C5 11              CMP TABLE+1
0040  021F  D0 E8              BNE CLRLP
0041  0221  60                 RTS
0042  0222              ;
0043  0222              ;ROUTINE 'STORE': PLACES BUFFER CONTENTS IN
0044  0222              ;TABLE, USING 1ST 6 CHRS. OF BUFFER AS A
0045  0222              ;'KEY' TO DETERMINE HASHED ADDRESS IN
0046  0222              ;TABLE.
0047  0222              ;
0048  0222  A2 00       STORE  LDX #0           ;CLEAR X FOR INDEXED ADDRESSING.
0049  0224  20 90 02           JSR HASH         ;GET HASHED INDEX..
0050  0227  20 62 02     CMPR1 JSR LIMIT        ;MAKE SURE INDEX IS WITHIN BOUNDS.
0051  022A  A1 13              LDA (PTR,X)      ;CHECK DATA UNIT...
0052  022C  F0 05              BEQ EMPTY        ;JUMP IF EMPTY.
0053  022E  E6 12              INC INDX         ;TRY NEXT UNIT.
0054  0230  4C 27 02           JMP CMPR1        ;CHECK FOR NEXT UNIT INDEX VALID.
0055  0233  A0 07       EMPTY  LDY #7           ;LOOP BY 7 TO LOAD DATA UNIT.
0056  0235  B9 16 00     FILL   LDA BUFFER,Y     ;GET CHR FROM BUFFER,
0057  0238  91 13              STA (PTR),Y      ;PLACE IT IN BUFFER.
0058  023A  88                 DEY
0059  023B  10 F8              BPL FILL         ;XFER NEXT CHR.
0060  023D  60                 RTS              ;ADDITION DONE.
0061  023E              ;
0062  023E              ;ROUTINE 'FIND' :
0063  023E              ;FINDS ENTRY WHOSE KEY IS IN BUFFER.
0064  023E              ;ENTRY, WHEN FOUND, IS COPIED INTO
0065  023E              ;BUFFER, ALONG WITH 2 BYTES OF DATA.
0066  023E              ;
0067  023E  A2 00       FIND   LDX #0           ;CLEAR X FOR INDIRECT ADDRESSING.
0068  0240  20 90 02           JSR HASH         ;GET HASH PRODUCT.
0069  0243  20 62 02     CMPR2 JSR LIMIT        ;MAKE SURE RESULT IS WITHIN LIMITS
```

Fig. 9-45: Hash-programma

```
0070  0246  A0 05        LDY #5          ;LOOP 6X TO COMPARE BUFFER TO DATA ITEM.
0071  0248  B1 13  CHKLP  LDA (PTR),Y     ;GET CHR FROM TABLE.
0072  024A  D9 16 00      CMP BUFFER,Y    ;IS IT = BUFFER CHR?
0073  024D  D0 0E        BNE BAD         ;IF NOT, TRY NEXT DATA UNIT.
0074  024F  88           DEY
0075  0250  10 F6        BPL CHKLP       ;CHECK NEXT CHRS.
0076  0252  A0 07  MATCH  LDY #7          ;LOOP 8X TO XFER CHRS TO BUFFER.
0077  0254  B1 13  XFER   LDA (PTR),Y     ;GET CHR. FROM TABLE.
0078  0256  99 16 00      STA BUFFER,Y    ;STORE IN BUFFER.
0079  0259  88           DEY
0080  025A  10 F8        BPL XFER        ;LOOP TO XFER CHRS.
0081  025C  60           RTS             ;DONE ;DATA UNIT FOUND, IN BUFFER.
0082  025D  E6 12  BAD    INC INDX        ;NOT FOUND, TRY NEXT DATA UNIT.
0083  025F  4C 43 02      JMP CMPR2       ;VALIDATE NEW DATA UNIT INDEX.
0084  0262              ;
0085  0262              ;ROUTINE TO MAKE SURE DATA INDEX IS WITHIN
0086  0262              ;BOUNDS SET BY ENTNUM, THEN MULTIPLY INDEX
0087  0262              ;BY 8, AND ADD IT TO TABLE POINTER. THE
0088  0262              ;RESULT IS PLACED IN 'PTR' AS DATA UNIT ADDRESS.
0089  0262              ;
0090  0262  A5 12  LIMIT  LDA INDX        ;GET INDEX.
0091  0264  C5 15  TEST   CMP ENTNUM      ;INDEX > NUMBER OF DATA ITEMS?
0092  0266  90 06        BCC OK          ;JUMP IF NOT.
0093  0268  38           SEC             ;YES -
0094  0269  E5 15        SBC ENTNUM      ;SUBTRACT # OF ITEMS UNTIL
0095  026B  4C 64 02      JMP TEST        ;INDEX WITHIN BOUNDS.
0096  026E  85 13  OK     STA PTR         ;STORE GOOD INDEX IN POINTER.
0097  0270  85 12        STA INDX        ;SAVE UPDATED INDEX.
0098  0272  A9 00  SHADD  LDA #0          ;CLEAR UPPER POINTER FOR SHIFT.
0099  0274  85 14        STA PTR+1
0100  0276  06 13        ASL PTR         ;SHIFT PTR 3X LEFT - MULTIPLY BY 8.
0101  0278  26 14        ROL PTR+1
0102  027A  06 13        ASL PTR
0103  027C  26 14        ROL PTR+1
0104  027E  06 13        ASL PTR
0105  0280  26 14        ROL PTR+1
0106  0282  18           CLC
0107  0283  A5 10        LDA TABLE       ;ADD POINTER AND TABLE START
0108  0285  65 13        ADC PTR         ;ADDRESS AND PLACE RESULT IN POINTER.
0109  0287  85 13        STA PTR
0110  0289  A5 11        LDA TABLE+1
0111  028B  65 14        ADC PTR+1
0112  028D  85 14        STA PTR+1
0113  028F  60           RTS
0114  0290              ;
0115  0290              ;ROUTINE TO GENERATE DATA UNIT INDEX IN TABLE
0116  0290              ;BY HASHING 'KEY', OR CHRS OF LABEL.
0117  0290              ;
0118  0290  A9 00  HASH   LDA #0          ;CLEAR LOCATION FOR INDEX.
0119  0292  18           CLC             ;PREPARE TO ADD.
0120  0293  A0 05        LDY #5          ;LOOP 6X FOR EXCLUSIVE ORS.
0121  0295  59 16 00  EXOR   EOR BUFFER,Y   ;EXCLUSIVE-OR ACCUM. WITH BUFFER CHR.
0122  0298  2A           ROL A           ;MULTIPLY ACCUM. BY 2.
0123  0299  88           DEY             ;COUNT DOWN CHRS.
0124  029A  10 F9        BPL EXOR        ;GET NEXT CHR.
0125  029C  85 12        STA INDX SAVE HASH PRODUCT AS INDEX.
0126  029E  60           RTS             ;DONE.
0127  029F              .END

ERRORS = 0000 <0000>

SYMBOL TABLE

SYMBOL   VALUE

BAD      025D   BUFFER   0016   CHKLP    0248   CLRLP    0209
CMPR1    0227   CMPR2    0243   DECR     0211   EMPTY    0233
ENTNUM   0015   EXOR     0295   FILL     0235   FIND     023E
HASH     0290   INDX     0012   INIT     0200   LIMIT    0262
MATCH    0252   OK       026E   PTR      0013   SHADD    0272
STORE    0222   TABLE    0010   TEST     0264   XFER     0254

END OF ASSEMBLY
```

Fig. 9-45: Hash-programma (vervolg)

BUBBLE-SORT

Bubble-sort is een sorteertechniek die gebruikt wordt om elementen van een tabel in stijgende of dalende volgorde te plaatsen. Deze bubble-sort techniek ontleent zijn naam aan het feit dat kleinste element in de tabel naar boven "borrelt". Iedere keer als het "botst" met een "zwaarder" element, springt het er overheen. Een praktisch voorbeeld van bubble-sort is te zien in fig. 9-46. De lijst die gesorteerd moet worden bevat: 10, 5, 0 en 100, en moet in dalende volgorde gesorteerd worden ("0" boven). Het algoritme is eenvoudig en het stroomdiagram is gegeven in fig. 9-47.

De bovenste twee (of de onderste twee) elementen worden vergeleken. Als de onderste kleiner ("lichter") is dan de bovenste, worden ze omgewisseld. Anders blijven ze waar ze zijn. Om praktische redenen wordt een eventuele omwisseling voor later gebruik genoteerd. Vervolgens wordt het volgende paar elementen vergeleken enz. tot alle elementen twee aan twee vergeleken zijn.

De eerste doorloop is geillustreerd in fig. 9-47 door stappen 1 tot en met 6, van beneden naar boven gaande . (We zouden ook van boven naar beneden kunnen gaan.)

Als in een doorloop geen elementen verwisseld worden is de tabel gesorteerd. Als er wel een omwisseling plaats vond, beginnen we opnieuw.

In fig. 9-47 is te zien dat in dit voorbeeld vier doorlopen nodig zijn.

Het hierboven beschreven proces is eenvoudig en wordt veel toegepast.

Er is nog een kleine moeilijkheid bij het uitvoeren van een omwisseling. Om A en B om te wisselen mag je niet schrijven:

```
A = B
B = A
```

omdat dan de vorige waarde van A verloren gaat. (Probeer maar eens een voorbeeld.)

Een juiste oplossing is het gebruik van een tijdelijke variabele of plaats om de waarde van A te bewaren:

```
TIJD = A
A    = B
B    = TIJD
```

Dit werkt. (Probeer ook dit maar eens.) Dit wordt een circulaire permutatie genoemd, en op deze manier voeren alle programma's de omwisseling uit. De techniek is geillustreerd in het stroomdiagram van fig. 9-47.

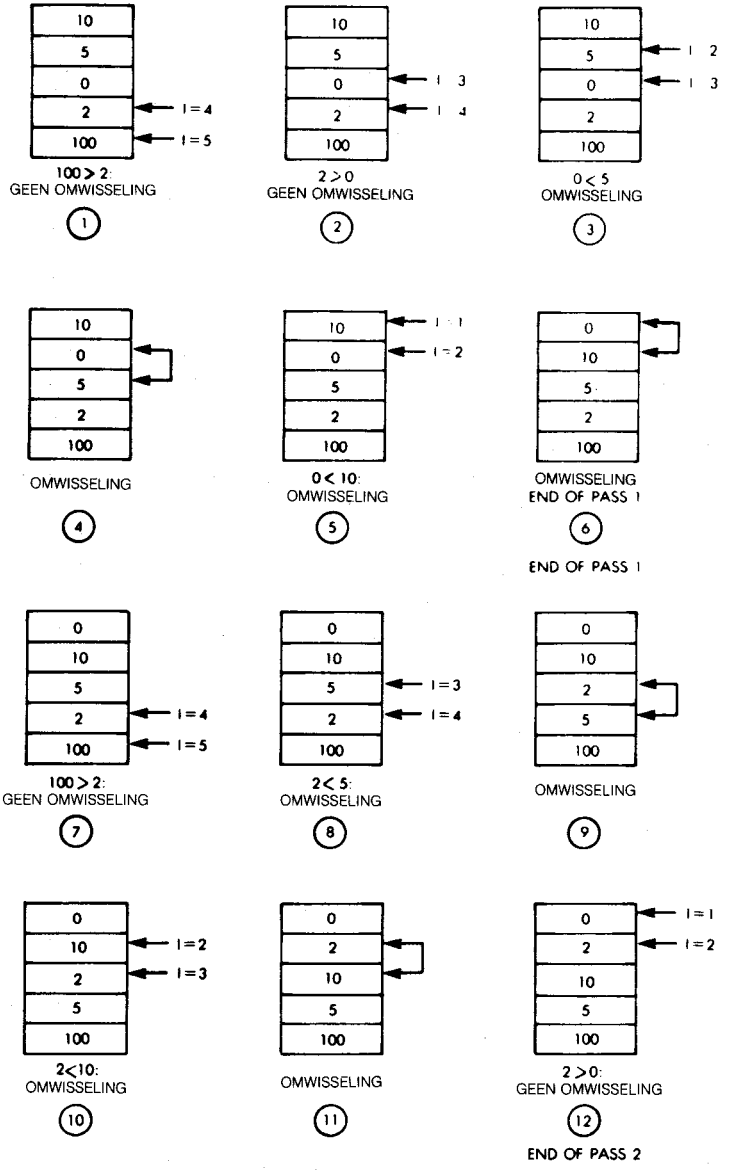


Fig. 9-46: Bubble-sort voorbeeld

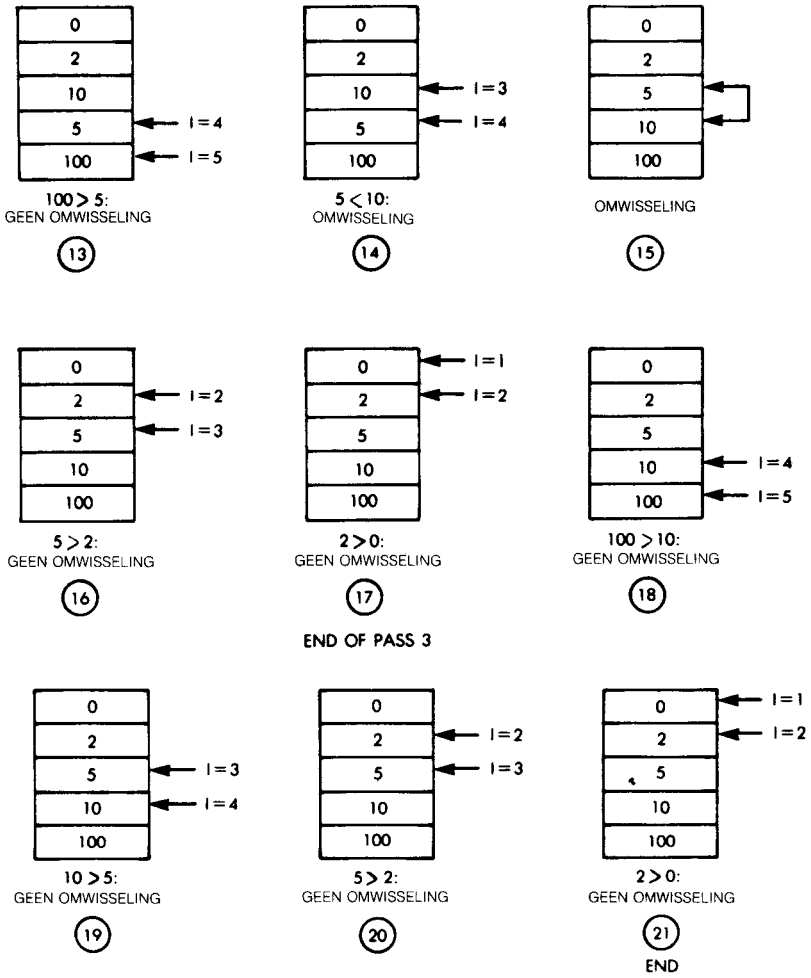


Fig. 9-46:Bubble-sort voorbeeld (vervolg)

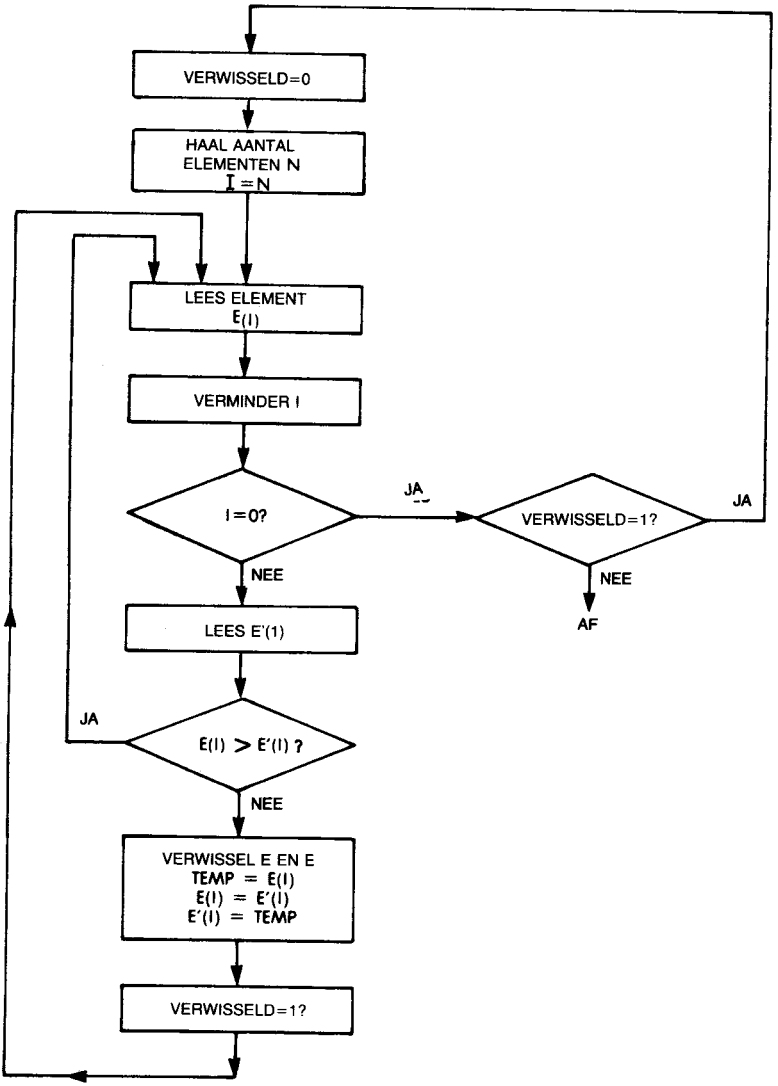


Fig. 9-47: Bubble-sort

De geheugenindeling voor het bubble-sort programma is gegeven in fig. 9-48. In dit programma zal ieder element een 8-bits positief getal zijn. Het programma staat op adressen 200 en volgende. Register X wordt gebruikt om te onthouden of er een omwisseling heeft plaats gevonden of niet, terwijl register Y gebruikt wordt als de variabele wijzer in de tabel. TAB is het beginadres van de tabel. Het programma is te zien in fig. 9-49. Er wordt gebruik gemaakt van indirekte geindexeerde adressering om efficient toegang te krijgen. Merk op hoe kort het programma is, dank zij de indirekte adresseringsmode van de 6502.

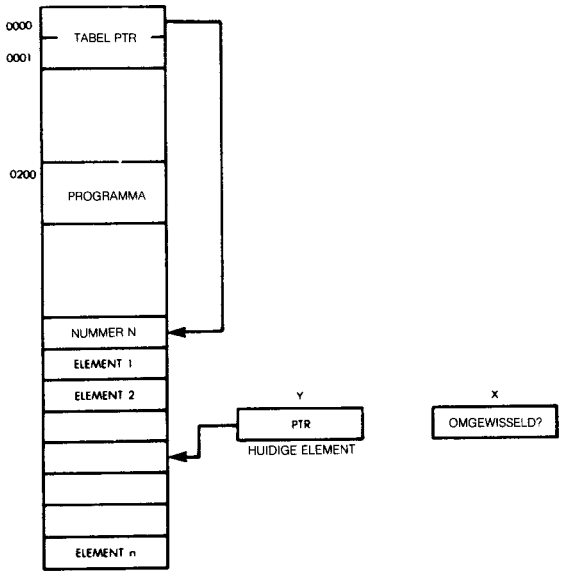


Fig. 9-48: Bubble-sort: geheugenindeling

```
SORT......PAGE 0001

LINE # LOC     CODE       LINE

0002  0000                ;     BUBBLE SORT PROGRAM
0003  0000                ;
0004  0000                      * = $0
0005  0000                ;
0006  0000  00 06         TAB   .WORD $600
0007  0002                ;
0008  0002                      * = $200
0009  0200                ;
0010  0200  A2 00         SORT  LDX #0          ;SET 'EXCHANGED' TO 0
0011  0202  A1 00               LDA (TAB,X)
0012  0204  AB                  TAY             ;NUMBER OF ELEMENTS IS IN Y
0013  0205  B1 00         LOOP  LDA (TAB),Y     ;READ ELEMENT E(I)
0014  0207  88                  DEY             ;DECREMENT NUMBER OF ELEMENTS TO READ.
0015  0208  F0 12               BEQ FINISH      ;END IF NO MORE ELEMENTS
0016  020A  D1 00               CMP (TAB),Y     ;COMPARE TO E'(I)
0017  020C  B0 F7               BCS LOOP        ;GET NEXT ELEMENT IF E(I)>E (I)
0018  020E  AA           EXCH  TAX             ;EXCHANGE ELEMENTS
0019  020F  B1 00               LDA (TAB),Y
0020  0211  C8                  INY
0021  0212  91 00               STA (TAB),Y
0022  0214  8A                  TXA
0023  0215  88                  DEY
0024  0216  91 00               STA (TAB),Y
0025  0218  A2 01               LDX #1          ;SET 'EXCHANGED' TO 1
0026  021A  D0 E9               BNE LOOP        ;GET NEXT ELEMENT
0027  021C  8A           FINISH TXA             ;SHIFT 'EXCHANGED' TO A REG. FOR COMPARE...
0028  021D  D0 E1               BNE SORT        ;IF SOME EXCHANGES MADE, DO ANOTHER PASS.
0029  021F  60                  RTS
0030  0220                      .END

ERRORS = 0000 <0000>

SYMBOL TABLE

SYMBOL   VALUE

EXCH     020E   FINISH   021C   LOOP     0205   SORT     0200
TAB      0000
END OF ASSEMBLY

<
<
```

Fig. 9-49: Bubble-sort programma

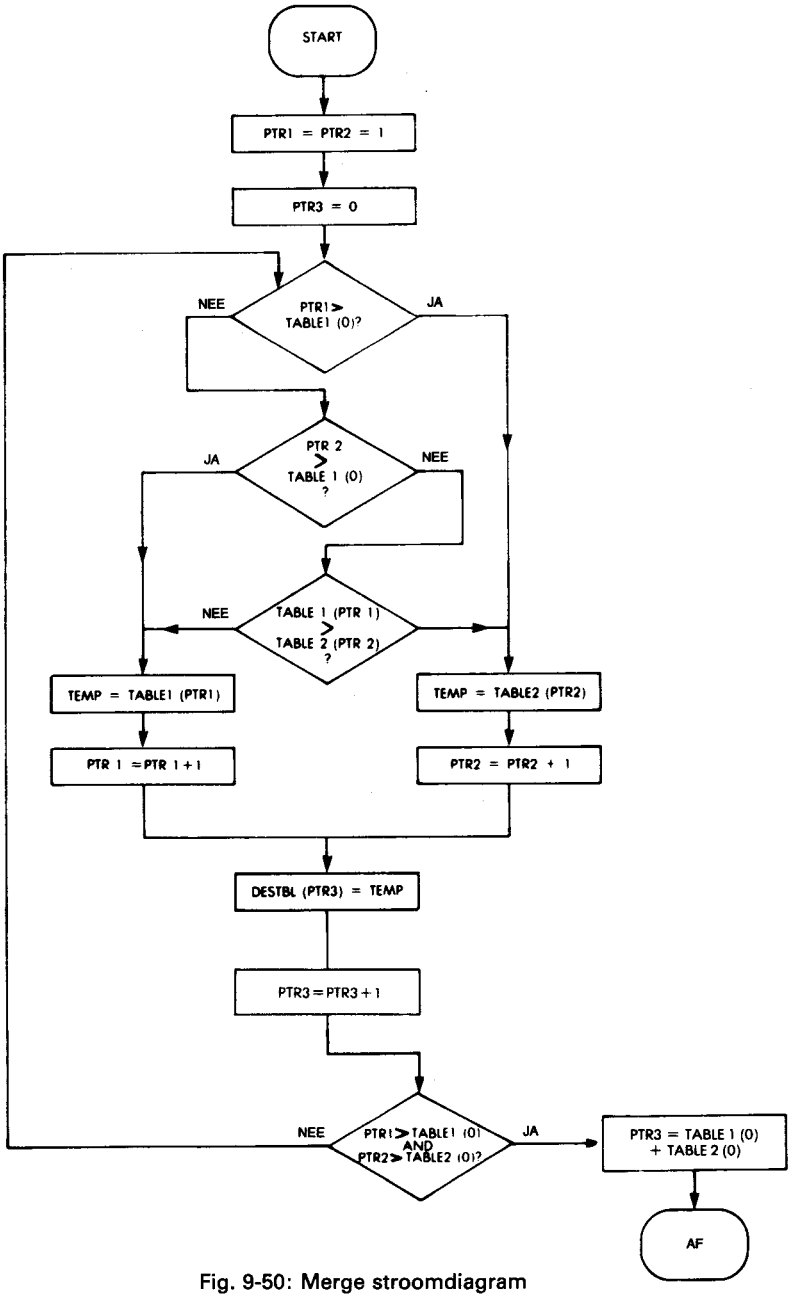


Fig. 9-50: Merge stroomdiagram

EEN MERGE ALGORITME

Een ander probleem is het mengen (merging) van twee verzamelingen gegevens in een derde verzameling. We nemen hier aan dat de twee tabellen met gegevens al gesorteerd zijn, en we willen ze nu in een derde tabel mengen. De lengte van ieder van de twee oorspronkelijke tabellen is beperkt tot 256 bytes (een pagina). Het eerste gegeven van iedere tabel bevat de lengte van de tabel.

Het algoritme voor het mengen van de twee tabellen is getoond in fig. 9-50. De overeenkomstige geheugenindeling is te zien in fig. 9-51, en het programma in fig. 9-52. Denk eraan, dat "TABLE1", "TABLE2" en "DESTBL" voor gebruik gezet moeten worden.

Het algoritme is eenvoudig. Twee variabele wijzers PTR1 en PTR2 wijzen naar de twee oorspronkelijke tabellen. PTR3 wijst naar de nieuwe tabel.

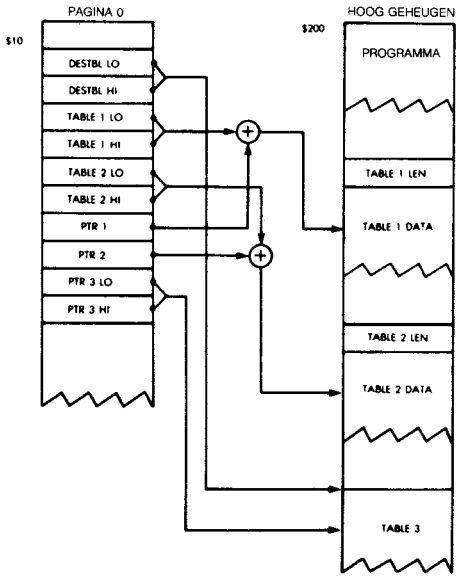


Fig. 9-51: Merge geheugenindeling

```
LINE # LOC     CODE       LINE

0002  0000                ;2-PAGE MERGE.
0003  0000                ;TAKES 2 DATA TABLES PREVIOUSLY SORTED,
0004  0000                ;AND MERGES THEM INTO A THIRD TABLE.
0005  0000                ;EACH SOURCE TABLE CAN BE UP TO ONE
0006  0000                ;PAGE (256 BYTES) IN LENGTH.
0007  0000                ;THE FIRST ELEMENT OF THE SOURCE
0008  0000                ;TABLES MUST CONTAIN THE TABLE LENGTH.
0009  0000                ;'PTR3' CONTAINS THE LENGTH OF THE
0010  0000                ;DESTINATION TABLE AT RETURN.
0011  0000                ;
0012  0000                      * = 010
0013  0010                DESTBL *=*+2          ;POINTER TO BEGINNING OF DESTINATION TABLE.
0014  0012                TABLE1 *=*+2          ;POINTER TO SOURCE TABLE 1.
0015  0014                TABLE2 *=*+2          ;POINTER TO SOURCE TABLE 2.
0016  0016                PTR1   *=*+1        ;TABLE 1 INDEX.
0017  0017                PTR2   *=*+1        ;TABLE 2 INDEX.
0018  0018                PTR3   *=*+2        ;DESTINATION TABLE INDEX.
0019  001A                ;
0020  001A                      * = 0200
0021  0200                ;
0022  0200  A5 11             LDA DESTBL+1    ;PTR3 = TABLE3
0023  0202  85 19             STA PTR3+1
0024  0204  A5 10             LDA DESTBL
0025  0206  85 18             STA PTR3
0026  0208  A9 01             LDA #1          ;SET SOURCE TABLE POINTERS TO BEGINNING,
0027  020A  85 16             STA PTR1        ;SKIPPING TABLE LENGTHS.
0028  020C  85 17             STA PTR2
0029  020E  A2 00             LDX #0          ;CLEAR X FOR INDIRECT ADDRESSING.
0030  0210  A1 14       COMPR LDA (TABLE2,X)  ;IS TABLE 2 LENGTH <
0031  0212  C5 17             CMP PTR2        ;TABLE 2 POINTER?
0032  0214  90 19             BCC TKTB1       ;IF YES, GET BYTE FROM TABLE 1.
0033  0216  A1 12             LDA (TABLE1,X)  ;IS TABLE 1 LENGTH <
0034  0218  C5 16             CMP PTR1        ;TABLE 1 POINTER?
0035  021A  90 0A             BCC TKTB2       ;IF YES, GET BYTE FROM TABLE 2
0036  021C  A4 16             LDY PTR1        ;GET POINTER FOR TABLE 1.
0037  021E  B1 12             LDA (TABLE1),Y  ;USE IT TO FETCH BYTE.
0038  0220  A4 17             LDY PTR2        ;GET POINTER FOR TABLE 2,
0039  0222  D1 14             CMP (TABLE2),Y  ;USE IT TO FIND BYTE TO COMPARE
0040  0224                     ;TO TABLE 1 BYTE.
0041  0224  90 09             BCC TKTB1       ;IF TABLE 1 BYTE LESS, TAKE IT.
0042  0226  A4 17       TKTB2 LDY PTR2        ;GET POINTER FOR TABLE 2.
0043  0228  B1 14             LDA (TABLE2),Y  ;GET NEXT BYTE FROM TABLE 2.
0044  022A  E6 17             INC PTR2        ;INCREMENT POINTER FOR TABLE 2.
0045  022C  4C 35 02          JMP STORE       ;GO STORE BYTE IN DESTINATION TABLE.
0046  022F  A4 16       TKTB1 LDY PTR1        ;GET POINTER 1...
0047  0231  B1 12             LDA (TABLE1),Y  ;AND USE IT TO GET BYTE FROM TABLE.
0048  0233  E6 16             INC PTR1        ;INCREMENT POINTER FOR TABLE 1.
0049  0235  81 18       STORE STA (PTR3,X)    ;STORE BYTE AT NEXT LOCATION IN TABLE 3.
0050  0237  E6 18             INC PTR3        ;INCREMENT LO ORDER TABLE 3 POINTER.
0051  0239  D0 02             BNE CC          ;IF NO OVERFLOW, SKIP
0052  023B  E6 19             INC PTR3+1      ;INCREMENT HI ORDER TABLE 3 POINTER.
0053  023D  A1 12       CC    LDA (TABLE1,X)  ;IS TABLE 1 LENGTH GREATER
0054  023F  C5 16             CMP PTR1        ;THAN OR EQUAL TO POINTER 1?
0055  0241  B0 CD             BCS COMPR       ;IF YES, GET NEXT BYTE.
0056  0243  A1 14             LDA (TABLE2,X)  ;IS TABLE 2 LENGTH GREATER
0057  0245  C5 17             CMP PTR2        ;THAN OR EQUAL TO POINTER 2?
0058  0247  B0 C7             BCS COMPR       ;IF YES, GET NEXT BYTE.
0059  0249  A9 00             LDA #0
0060  024B  85 19             STA PTR3+1      ;CLEAR PTR3 HI ORDER.
0061  024D  18                CLC             ;MERGE DONE, NOW..
0062  024E  A1 12             LDA (TABLE1,X)  ;ADD TABLE 1 AND 2 LENGTHS.
0063  0250  61 14             ADC (TABLE2,X)
0064  0252  85 18             STA PTR3        ;STORE SUM IN TABLE 3 TEMPORARY POINTER.
0065  0254  90 04             BCC CCC         ;AND..
0066  0256  A9 01             LDA #1          ;OVERFLOW IN...
0067  0258  85 19             STA PTR3+1      ;HI BYTE.
0068  025A  60          CCC   RTS
0069  025B                   .END

ERRORS = 0000 <0000>
END OF ASSEMBLY
```

Fig. 9-52: Merge programma

De huidige gegevens van TABLE1 en TABLE2 worden twee aan twee vergeleken. De kleinste wordt in TABLE3 gekopieerd en de bijbehorende wijzer wordt vermeerderd. Het proces wordt herhaald en eindigt als zowel PTR1 als PTR2 onderin de respektievelijke tabellen zijn aangekomen.

SAMENVATTING

Basisbegrippen en voorbeelden met betrekking tot veel voorkomende gegevensstrukturen zijn gepresenteerd.

De 6502 leent zich uitstekend voor het behandelen van ingewikkelde gegevensstrukturen vanwege zijn krachtige adresseermodes. De effitie wordt aangetoond door de beknoptheid van de getoonde programma's.

Verder zijn speciale technieken gepresenteerd voor hashing (verdelen), sorteren en merging (versmelten), die representatief zijn voor die, welke gebruikt worden om de ingewikkelde problemen op te lossen die gepaard gaan met de eigenlijke gegevensstrukturen.

De beginnende programmeur hoeft zich nog niet bezig te houden met de details van de uitvoering en de behandeling van gegevensstrukturen. Voor het doelmatig programmeren van niet voor de hand liggende algoritmes, is een goed begrip van gegevensstrukturen nodig. De voorbeelden die in dit hoofdstuk gegeven zijn, kunnen de lezer helpen om dat begrip te krijgen, en alle problemen op te lossen die optreden bij redelijke gegevensstrukturen.

10
ONTWIKKELING VAN PROGRAMMA'S

INLEIDING

Alle programma's die we tot nu toe ontwikkeld en bestudeerd hebben, zijn met de hand ontwikkeld, zonder enige hardware of software ondersteuning. De enige verbetering m.b.t. de binaire kodering is het gebuik van mnemonische symbolen geweest, die van de assembleertaal. Voor een doelmatige ontwikkeling van software moeten de hardware en software ontwikkeling hulpmiddelen begrepen worden. Het is de doelstelling van dit hoofdstuk om deze hulpmiddelen te presenteren en te evalueren.

BASIS PROGRAMMEERKEUZEN

Er zijn drie alternatieven: een programma schrijven in binaire of hexadecimale kode, in assembleertaal, of in een hogere programmeertaal. Laten we deze alternatieven eens beschouwen.

1 – Hexadecimale kodering

Het programma wordt normaal geschreven in assembleertaal mnemonics. De meeste goedkope one-board (een bord, printplaat) computers hebben echter geen assembler. De assembler is het programma dat de mnemonics automatisch vertaalt in de vereiste binaire kodes. Als er geen assembler is, moet deze vertaling met de hand gedaan worden. Binair is onplezierig in gebruik en heeft veel fouten tot gevolg, zodat

normaal hexadecimaal gebruikt wordt. In hoofdstuk 1 hebben we al gezien, dat een hexadecimaal cijfer vier binaire bits voorstelt. Om de inhoud van een byte voor te stellen zijn dus twee hexadecimale cijfers nodig. Als voorbeeld is de tabel met de hexadecimale equivalenten van de 6502 instrukties in de Appendix gegeven.

Om kort te gaan, als de mogelijkheden van de gebruiker beperkt zijn en er geen assembler is, moet hij het programma met de hand in hexadecimaal omzetten. Dit kan voor een klein aantal instrukties, zeg 10 tot 100, nog wel met de hand gebeuren. Voor langere programma's is dit proces vervelend, en een bron van fouten. Bij de meeste single-board microcomputers moeten de programma's echter hexadecimaal ingevoerd worden. Ze zijn niet uitgerust met een assembler en hebben geen volledig alfanumeriek toetsenbord om de kosten te drukken.

Samenvattend is hexadecimale kodering geen aangename manier om programma's in te voeren in een computer. Het is alleen een economische manier. De kosten van een assembler en het vereiste alfanumerieke toetsenbord worden afgewogen tegen het extra werk om het programma in het geheugen te krijgen. Dit verandert echter niet de wijze waarop het programma geschreven wordt. *Het progamma wordt nog steeds in assembleertaal geschreven*, zodat het door de menselijke programmeur onderzocht kan worden en betekenis heeft.

2 – Assembleer programmering

Het programmeren op assembleer niveau bestrijkt zowel het invoeren van programma's in het systeem in hexadecimale kode als in symbolische assembleervorm. Laten we nu eens de invoer van een programma, direkt in zijn assembleertaal voorstelling, onderzoeken. Er moet een assembleerprogramma aanwezig zijn. De assembler leest iedere mnemonische instruktie en vertaalt die in het vereiste bitpatroon, gebruik makend van 1, 2 of 3 bytes, al naar gelang de kode van de instruktie. Verder biedt een goede assembler een aantal faciliteiten om het programma te schrijven. Deze worden behandeld in het onderstaande deel over de assembler. In het bijzonder zijn er *pseudo-instrukties* beschikbaar, die de waarde van symbolen veranderen. Symbolische adressering kan gebruikt worden, en een sprong naar een symbolisch adres kan worden gespecificeerd. Tijdens de verbeteringsfase, waarin de gebruiker instrukties kan toevoegen of verwijderen, hoeft niet het hele programma herschreven te worden als er een extra in-

struktie geplaatst is tussen een spronginstruktie en het punt waarheen gesprongen wordt, zolang er maar symbolische etiketten gebruikt worden.

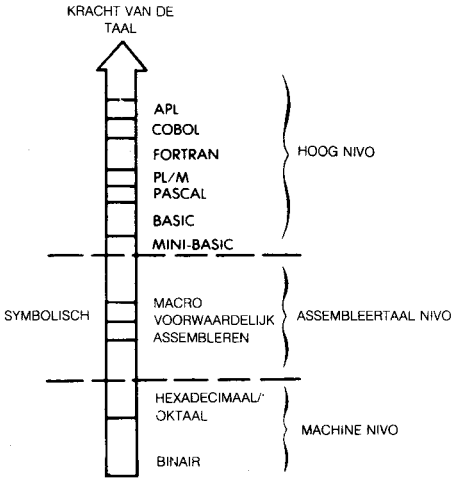


Fig. 10-1: Programmeerniveau's

De assembler zorgt automatisch voor de aanpassing van de etiketten tijdens het vertalingsproces. Verder stelt de assembler de gebruiker in staat om zijn programma in symbolische vorm te verbeteren. Er kan een disassembler gebruikt worden om de inhoud van een geheugenplaats te onderzoeken en de mnemonic te rekonstrueren die wordt voorgesteld. Onderstaand zullen de verschillende software hulpmiddelen bestudeerd worden, die meestal op een syteem beschikbaar zijn. Laten we nu het derde alternatief onderzoeken.

3 – Hogere programmeertalen

Een programma kan in een hogere programmeertaal geschreven zijn, zoals BASIC, APL, PASCAL of anderen. De technieken om in deze verschillende talen te prorammeren worden behandeld in de specifieke boeken, en zullen hier verder niet aan de orde komen. We zullen deze manier van programmeren daarom hier maar kort bespreken. Een hogere programmeertaal biedt krachtige instrukties, die het pro-

grammeren veel eenvoudiger en sneller maken. Deze instrukties moeten dan vertaald worden door een komplex programma in de uiteindelijke binaire representatie die door de microcomputer uitgevoerd kan worden. Bijna ieder hogere taal instruktie wordt vertaald in een groot aantal individuele binaire instrukties. Het programma dat deze automatische vertaling verzorgt wordt een *compiler* of een *interpreter* genoemd. Een compiler vertaalt alle instrukties van een programma na elkaar in object kode. In een volgende fase wordt deze kode uitgevoerd. In kontrast hiermee interpreteert een interpreter een enkele instruktie, voert die dan uit, en "vertaalt" dan de volgende. De interpreter heeft het voordeel van een interaktieve respons, maar heeft een lagere efficientie dan de compiler. Deze onderwerpen worden hier niet verder bestudeerd. Laten we terugkeren naar het programmeren van een microprocessor op assembly niveau.

SOFTWARE ONDERSTEUNING

We zullen hier de voornaamste software faciliteiten beschouwen, die beschikbaar zijn (of zouden moeten zijn) in het komplete systeem voor een gemakkelijke ontwikkeling van software. Sommige definities zijn al geintroduceerd. Ze worden hier samengevat, en de rest van de programma's wordt gedefinieerd voor we verder gaan.

De *assembler* is het programma dat de mnemonische voorstelling van de instrukties vertaalt in hun binaire equivalent. Het vertaalt normaal een symbolische instruktie in een binaire instruktie (1, 2, of 3 bytes). De resulterende binaire kode wordt *object kode* genoemd. Het kan meteen door de microcomputer uitgevoerd worden. Als een bijprodukt produceert de assembler ook een volledige symbolische listing van het programma, benevens de equivalentie-tabellen die door de programmeur gebruikt zijn en een lijst met in het programma voorkomende symbolen. Verderop in dit hoofdstuk zullen enkele voorbeelden gegeven worden.

Een *compiler* is het programma dat instrukties van een hogere programmeertaal omzet in hun binaire vorm.

Een *interpreter* is een soortgelijk programma dat ook hogere taal instrukties omzet in hun binaire vorm, maar die geen tussenrepresentatie bewaart en ze direkt uitvoert. In feite wordt vaak helemaal geen tus-

senkode gegenereerd, maar worden de instrukties van die hogere taal direkt uitgevoerd.

Een *monitor* is het basisprogramma dat onmisbaar is voor de hardwarebronnen van het systeem. Het kontroleert voortdurend de invoerapparaten of ze iets in te voeren hebben en behandelt de rest van de apparaten. Zo moet een minimale monitor voor een single-board computer, die uitgerust is met een toetsenbord en een LED display, voortdurend het toetsenbord afscannen om eventuele invoer van de gebruiker te detekteren, en de gespecificeerde inhoud op de licht-emiterende-diodes weergeven. Verder moet hij een aantal eenvoudige opdrachten van het toetsenbord begrijpen, zoals START, STOP, CONTINUE, LOAD MEMORY, EXAMINE MEMORY (resp. start, stop, ga door, laadt geheugen, onderzoek geheugen). Op een groter systeem wordt de monitor wel het *executive* (uitvoerende) programma genoemd. Als er ook voorzien is in een complex systeem om files te behandelen of taken te plannen, wordt deze algehele verzameling faciliteiten een *operating systeem* genoemd. Als de files op schijf (disk) opgeslagen kunnen worden, noemen we het operating systeem een *disk operating systeem*, of DOS.

Een *editor* is het programma dat ontworpen is om de invoer en het veranderen van tekst of programma's te vereenvoudigen. Het stelt de gebruiker in staat om op eenvoudige wijze karakters in te voeren, aan elkaar te koppelen, te verwijderen, regels tussen te voegen of te verwijderen etc. Het is een belangrijke voorziening voor eenvoudige invoer van tekst.

Een *debugger* (verbeteraar) is een noodzakelijke voorziening om fouten uit het programma te halen. als een programma niet werkt, zal er in de meeste gevallen geen enkele indikatie van de oorzaak zijn. De programmeur zal dus breekpunten in het programma willen plaatsen, om op vastgestelde adressen het programma te onderbreken en de inhoud van registers of geheugen te onderzoeken. Dit is de voornaamste funktie van de debugger. De debugger heeft dus de mogelijkheid een programma te onderbreken, door te starten, registers of geheugen te onderzoeken en te veranderen. Een goede debugger is uitgerust met nog meer faciliteiten, zoals het onderzoeken van gegevens in symbolische vorm, hexadecimaal, binair of andre gebruikelijke representaties, als ook het invoeren van gegevens in deze formaten.

Een *loader*, of *linking loader*, plaatst verschillende blokken met object kode op bepaalde plaatsen in het geheugen en past hun wederzijdse symbolische wijzers aan, zodat ze naar elkaar kunnen verwijzen. Hij wordt gebruikt om programma's of blokken te relocateren (nieuwe plaats geven) in verschillende geheugengebieden.

Een *simulator* of *emulator* programma wordt gebruikt om de werking van een apparaat te simuleren (meestal een microprocessor), die niet aanwezig is, en om programma's op een gesimuleerde processor te ontwikkelen voordat ze op het werkelijke board geplaatst worden. Met deze aanpak is het mogelijk een programma te onderbreken, het te veranderen en het in RAM te houden. De nadelen van een simulator zijn:

1 - Hij simuleert alleen de processor zelf en niet de I/O apparaten,
2 - De uitvoeringssnelheid is langzaam, en er wordt gewerkt met gesimuleerde tijd. Het is dus niet mogelijk om real-time apparaten te testen (real-time = werkelijke tijd), en er kunnen nog synchronisatie problemen optreden, hoewel de logica van het programma korrekt bevonden is.

Een *emulator* is in wezen een real-time simulator. Hij gebruikt een processor om een andere in alle details te simuleren.

Utiliteitsroutines zijn in wezen al die routines, die voor de meeste toepassingen nuttig zijn, en waarvan de gebruiker zou wensen, dat de fabrikant ze geleverd had!

Ze kunnen vermenigvuldigings-, delings-, en andere rekenkundige routines omvatten, routines om blokken te verplaatsen, karakters te testen, invoer/uitvoer apparaten te behandelen (I/O-drivers), en nog vele andere.

De volgorde van programmaontwikkeling

We zullen nu een typische volgorde onderzoeken waarin een programma op assembleer-niveau wordt ontwikkeld. We zullen aannemen, dat alle gebruikelijke software faciliteiten aanwezig zijn, om hun waarde te demonstreren. Als ze niet op een systeem beschikbaar zijn,

is het nog wel mogelijk om programma's te ontwikkelen, maar met minder gemak, en de tijd die nodig is om het programma te verbeteren zal waarschijnlijk toenemen.

De gebruikelijke aanpak is om eerst een algoritme te ontwerpen en de gegevensstrukturen te definieren voor het probleem dat opgelost moet worden. Vervolgens wordt een uitgebreide verzameling stroomdiagrammen ontwikkeld die de stroom van het programma voorstellen. Tenslotte worden de stroomdiagrammen vertaald in de assembleertaal voor de microprocessor: dit is de koderingsfase.

Vervolgens moet het programma in de computer ingevoerd worden. We zullen in het volgende deel de hardware opties onderzoeken die in deze fase gebruikt kunnen worden.

Het programma wordt onder kontrole van de editor in het RAM geheugen van het systeem geladen. Zodra een deel van het programma geladen is, zoals de subroutines, wordt dit getest.

Maar eerst wordt de assembler gebruikt. Als de assembler nog niet in het systeem aanwezig was, wordt hij geladen van een extern geheugen, zoals de schijf. Dan wordt het programma geassembleerd, d.w.z. vertaald in binaire kode. Dit heeft het object programma als resultaat, dat meteen uitgevoerd kan worden.

Het is niet normaal dat men verwacht, dat een programma de eerste keer foutloos werkt. Om de juiste werking te verifieren, wordt meestal een aantal breekpunten (breakpoints) op cruciale plaatsen in het programma geplaatst, waar tussenresultaten eenvoudig getest kunnen worden. Hiervoor wordt de debugger gebruikt. Op geselekteerde plaatsen worden breekpunten gespecificeerd. Vervolgens wordt er een "GO" kommando gegeven, zodat de uitvoering van het programma begint. Het programma stopt dan automatisch bij de gespecificeerde breekpunten. De programmeur kan dan kontroleren, door de inhoud van het geheugen en de registers te onderzoeken, of de gegevens tot nu toe goed zijn. Als deze goed zijn gaan we door naar het volgende breekpunt. Als we foute gegevens vinden, was er een fout in het programma. Nu bekijkt de programmeur meestal zijn programmalisting om te kontroleren of de kodering juist is. Als er in de kodering geen fout gevonden kan worden is de fout misschien van logische aard, en moet het stroomdiagram geraadpleegd worden. We zullen hier aannemen dat de stroomdiagrammen gekontroleerd zijn en redelijk juist zijn. Het is waarschijnlijk dat de fout in de kodering zit. Daarom moet

een deel van het programma veranderd worden. Als de symbolische voorstelling van het programma nog in het geheugen zit, roepen we gewoon de editor weer aan en verbeteren de nodige regels, en herhalen dan de voorgaande procedure. In sommige sytemen is de geheugenhoeveelheid misschien niet voldoende, en kan het nodig zijn om de symbolische voorstelling van het programma naar schijf of kassette weg te schrijven, voordat de objectkode uitgevoerd wordt. In dat geval moet de symbolische voorstelling van het programma natuurlijk weer van het betreffende medium geladen worden, voor we de editor kunnen aanroepen.

De bovenstaande procedure wordt net zovaak herhaald tot het programma korrekt is. We moeten benadrukken, dat helen beter is dan genezen. Een juist ontwerp heeft meestal een programma tot gevolg, dat snel korrekt uitgevoerd wordt, als eenmaal de gebruikelijke typefouten en voor de hand liggende kodeerfouten verwijderd zijn. Slordig ontwerp heeft programma's tot gevolg die veel tijd nodig hebben om verbeterd te worden. Het verbeteren kost in het algemeen meer tijd dan het eigenlijke ontwerpen. Om kort te gaan: het loont altijd om iets meer tijd aan het ontwerp te besteden, zodat de verbetertijd korter wordt.

Met deze aanpak kan de gehele organisatie van het programma getest worden, maar niet in real-time met I/O-apparaten. Als I/O-apparaten getest moeten worden, is de direkte oplossing het overbrengen van het programma in EPROM's, en deze in de schakeling te plaatsen en dan af te wachten of het funktioneert.

Er is echter een betere oplossing. En wel het gebruik van een *in-circuit emulator*. Een in-circuit emulator gebruikt de 6502 (of een andere microprocessor) om de 6502 in (bijna) real-time te emuleren. Hij emuleert de 6502 fysiek. De emulator is uitgerust met een kabel, afgesloten met een 40-pens konnektor, die dezelfde aansluitingen heeft als de 6502. Deze konnektor kan dan aangesloten worden op het echte toepassingsbord dat ontwikkeld wordt. De signalen die door deze emulator gegenereerd worden zijn precies gelijk aan die van de 6502, alleen misschien iets langzamer. Het wezenlijke voordeel is, dat het programma dat getest wordt, nog steeds in het RAM geheugen van het ontwikkelsysteem zit. Het zal de echte signalen genereren die kommuniceren met de echte I/O apparaten die men wil gebruiken. Het gevolg is, dat het nog steeds mogelijk is om gebruik te maken van alle faciliteiten van

het ontwikkelsysteem (editor, verbeteraar, filesysteem) terwijl de invoer/uitvoer in real-time getest wordt.

Verder voorziet een goede editor in een aantal speciale faciliteiten, zoals een *trace*. Een trace is een opname van de laatste instrukties of de status van verschillende databussen in het systeem, vlak voor een breekpunt. In het kort geeft de trace een film van de gebeurtenissen die vlak voor het breekpunt of de fout optraden. Hij kan zelfs een scope triggeren op een gespecificeerd adres, of bij het optreden van een bepaalde kombinatie van bits. Een dergelijke voorziening is erg waardevol, omdat als er een fout optreedt, het meestal te laat is. De instruktie of de gegevens die verantwoordelijk waren voor het optreden van de fout, zijn voor het detekteren ervan uitgevoerd. De beschikbaarheid van een trace stelt de gebruiker in de gelegenheid om vast te stellen

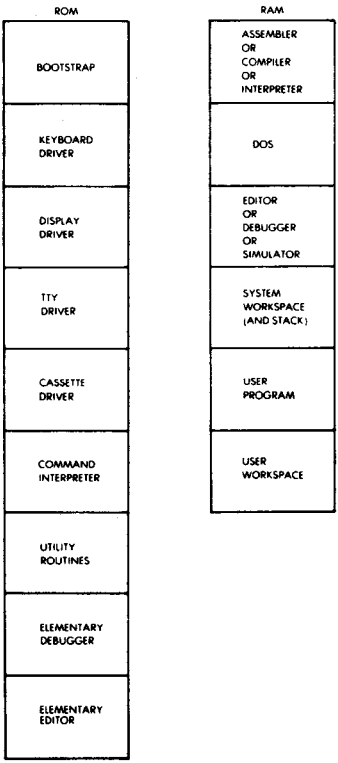


Fig. 10-2: Een typische geheugenindeling

welk segment van het programma de fout veroorzaakte. Als de trace niet lang genoeg is, zetten we eerder een breekpunt.

Hiermee is onze beschrijving van de gebruikelijke volgorde van gebeurtenissen bij het ontwikkelen van een programma volledig. We zullen nu de hardware alternatieven die voor het ontwikkelen van programma's eens onderzoeken.

DE HARDWARE ALTERNATIEVEN

1 – Single-board microcomputer

De single-board computer is de goedkoopste oplossing om programma's te ontwikkelen. Hij is meestal uitgerust met een hexadecimaal toetsenbord, plus enige funktietoetsen, en 6 LED's, die gegevens en adressen kunnen weergeven. Omdat hij uitgerust is met een beperkte hoeveelheid geheugen, is er meestal geen assembler aanwezig. Op zijn best heeft hij een kleine monitor, en zo goed als geen faciliteiten om te editen of te verbeteren, afgezien van een paar kommando's. Alle programma's moeten dus hexadecimaal ingevoerd worden. Ze worden ook hexadecimaal weergegeven op de LED's. Een single-board microcomputer heeft in theorie dezelfde hardware mogelijkheden als iedere andere computer. Maar eenvoudig vanwege het beperkte geheugen voorziet hij niet in de gebruikelijke faciliteiten van een groter systeem, maakt dus het ontwikkelen van programma's langduriger. Omdat het vervelend is om programma's in hexadecimaal formaat te ontwikkelen, kan een single-board computer het beste gebruikt worden voor onderwijs en training, waar korte programma's ontwikkeld worden en de korte lengte geen obstakel is voor het programmeren. Single-board computers zijn waarschijnlijk de goedkoopste manier om in de praktijk programmeren te leren. Ze kunnen echter niet voor de ontwikkeling van ingewikkelde programma's gebruikt worden, zonder dat er extra geheugenkaarten toegevoegd worden en de gebruikelijke software ondersteuning beschikbaar gemaakt wordt.

2 – Het ontwikkelsysteem

Een ontwikkelsysteem is een microcomputer systeem, uitgerust met een aanzienlijke hoeveelheid RAM geheugen (32K, 48K), en de nodige invoer/uitvoer apparaten zoals een beeldscherm, een printer,

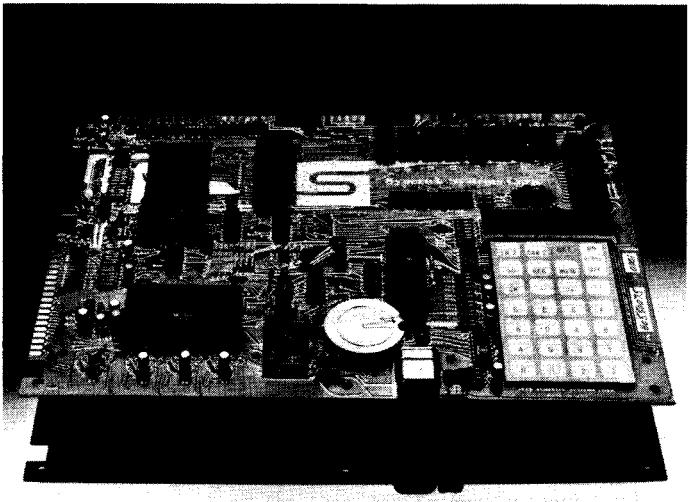


Fig. 10-3: SYM1 is een typisch microcomputer bord

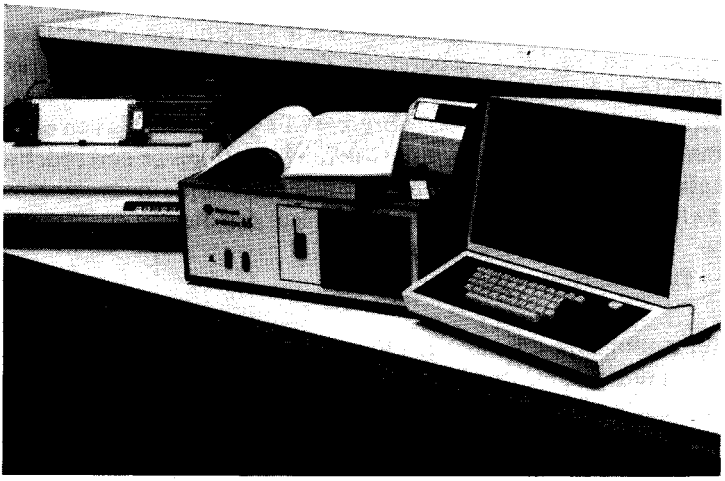


Fig. 10-4: Rockwell/Mostek System 65 is een ontwikkelsysteem

schijven en meestal een PROM programmer, benevens misschien een in-circuit emulator. Een ontwikkelsysteem is speciaal ontworpen om de ontwikkeling van programma's in een industriele omgeving te vereenvoudigen. Het biedt meestal alle, of de meeste, software faciliteiten die we in het voorgaande deel genoemd hebben. In principe is het het meest ideale software ontwikkelgereedschap.

De beperking van een microcomputer ontwikkelsysteem is, dat het niet altijd een compiler of een interpreter kan ondersteunen.
De reden hiervoor is, dat voor een compiler in de meeste gevallen een zeer grote hoeveelheid geheugen nodig is, vaak meer dan er op het systeem beschikbaar is. Voor het ontwikkelen van programma's in assembler, biedt het alle benodigde faciliteiten. Omdat ontwikkelsystemen minder vaak verkocht worden dan hobbycomputers, is hun prijs vaak aanzienlijk hoger.

3 – Hobby microcomputers

De hardware van een hobby microcomputer is natuurlijk precies hetzelfde als van een ontwikkelsysteem. Het voornaamste verschil is het feit dat hij meestal niet uitgerust is met de geavanceerde hulpmiddelen om software te ontwikkelen, die een industrieel ontwikkelsysteem heeft. Zo hebben de meeste hobbycomputers slechts een elementaire assembler, minimale editors, minimale filesystemen, geen faciliteiten om een PROM programmer aan te sluiten, geen in-circuit emulator, en geen krachtige verbeteraar. Ze vormen dus een tussenstap tussen de single-board microcomputer en het volledige microprocessor ontwikkelsysteem. Voor een gebruiker, die programma's van een middelmatige uitgebreidheid wil ontwikkelen, vormen ze een redelijk alternatief, omdat ze goedkoop zijn en een redelijke hoeveelheid ontwikkelhulpen voor software hebben, ook al zijn die niet erg uitgebreid.

4 – Time-sharing systeem

Het is mogelijk om van verschillende maatschappijen een terminal te huren die aangesloten is op een time-sharing netwerk. Deze terminals delen de tijd van grotere computers en genieten alle voordelen van grote installaties. Op vrijwel alle kommerciele time-sharing systemen zijn *cross-assemblers* voor alle microcomputers aanwezig. Een cross-assembler is gewoon een assembler voor bijvoorbeeld de 6502, die uitgevoerd wordt op bijvoorbeeld een IBM370. Formeel is een cross-assembler een assembler voor microprocessor X, die uitgevoerd wordt

op processor Y. De aard van de gebruikte computer is niet van belang. De gebruiker schrijft nog steeds een programma in 6502 assembleertaal, en de cross-assembler vertaalt het naar het juiste binaire patroon. Het verschil is echter, dat het programma hier niet uitgevoerd kan worden. Het kan wel uitgevoerd worden door een gesimuleerde processor als die aanwezig is, vooropgesteld dat er geen I/O bronnen gebruikt worden. Deze oplossing wordt daarom alleen in industriele toepasingen gebruikt.

5 – Eigen computer

Als er een grote computer beschikbaar is, kunnen er ook cross-assemblers zijn die het ontwikkelen van programma's vereenvoudigen. Als deze computer time-sharing faciliteiten biedt, is deze oplossing in wezen gelijk aan de bovenstaande. Als hij alleen batch-service biedt (batch: programma's invoeren d.m.v. ponskaarten b.v.), is dit waarschijnlijk een van de meest ongemakkelijke methoden om programma's te ontwikkelen, omdat het in batch mode inleveren van programma's op het niveau van assembleertalen resulteert in een erg lange ontwikkeltijd.

Frontpaneel of geen frontpaneel?

Het frontpaneel is een hardware voorziening, die vaak gebruikt wordt om het verbeteren van programma's te vereenvoudigen. Het is een traditioneel hulpmiddel om de binaire inhoud van een register of het geheugen weer te geven. Alle funkties van een frontpaneel kunnen echter door een terminal overgenomen worden, en de steeds meer toegepaste beeldbuis biedt nu een service die bijna equivalent is met het kontrolepaneel door de binaire waarde van bits weer te geven. Een bijkomend voordeel van een beeldbuis is, dat men naar wens over kan schakelen van binaire voorstelling naar hexadecimale, symbolische of decimale voorstelling (als de nodige konversieroutines beschikbaar zijn natuurlijk). Het nadeel van de beeldbuis is dat voor het verkrijgen van een bepaald beeld meerdere toetsen aangeslagen moeten worden, in plaats van eenvoudig een knop omdraaien. Omdat echter de kosten van een kontrolepaneel aanzienlijk zijn, wordt dit op de meeste microcomputers niet meer toegepast. De waarde van een kontrolepaneel wordt vaker door emotionele argumenten, die gebaseerd zijn op oude ervaringen, ingeschat, dan door rationele argumenten. Het is niet onmisbaar.

SAMENVATTING VAN DE HARDWARE VOORZIENINGEN

Er kunnen drie algemene gevallen onderscheiden worden: als je maar een minimaal budget hebt en programmeren wilt leren, koop dan een single-board microcomputer. Hiermee kunnen alle eenvoudige programma's uit dit boek, en meer, ontwikkeld worden. Eens echter, als je programma's van meer dan een paar honderd instrukties wilt ontwikkelen, zul je de beperkingen van deze aanpak merken.

Als je een industriele gebruiker bent heb je een volledig ontwikkelsysteem nodig. Iedere mindere oplossing heeft een langere ontwikkeltijd tot gevolg. De afwegingen zijn duidelijk: hardwaremiddelen tegen programmeertijd. Natuurlijk kan een minder dure aanpak gevolgd worden als de te ontwikkelen programma's eenvoudig zijn. Als er echter ingewikkelde programma's geschreven moeten worden, is het moeilijk om hardware besparingen te verdedigen bij de aanschaf van een ontwikkelsysteem, omdat de programmeerkosten verreweg dominant zijn bij een projekt.

Voor prive doeleinden is de hobbycomputer meestal wel voldoende, hoewel hij minimale faciliteiten heeft. Voor de meeste hobbycomputers moet er nog goede ontwikkelsoftware komen. De gebruiker zal zijn wensen moeten analyseren aan de hand van de in dit hoofstuk gegeven opmerkingen.

Laten we nu het meest onmisbare hulpmiddel eens in detail analyseren: de assembler.

DE ASSEMBLER

We hebben in het hele boek assembleertaal gebruikt, zonder de formele syntax te geven van deze taal. Het moment is nu daar om deze definities te geven. De assembler is ontworpen om met eenvoudige symbolen het gebruikersprogramma te representeren, en het toch voor het assembleerprogramma mogelijk te maken om deze mnemonics eenvoudig om te zetten in hun binaire representatie.

Assembler velden

Bij het intypen van een assemblerprogramma hebben we gezien, dat er velden gebruikt worden. Die zijn:

Het labelveld, optioneel, dat een symbolisch adres kan bevatten voor de instruktie die volgt.

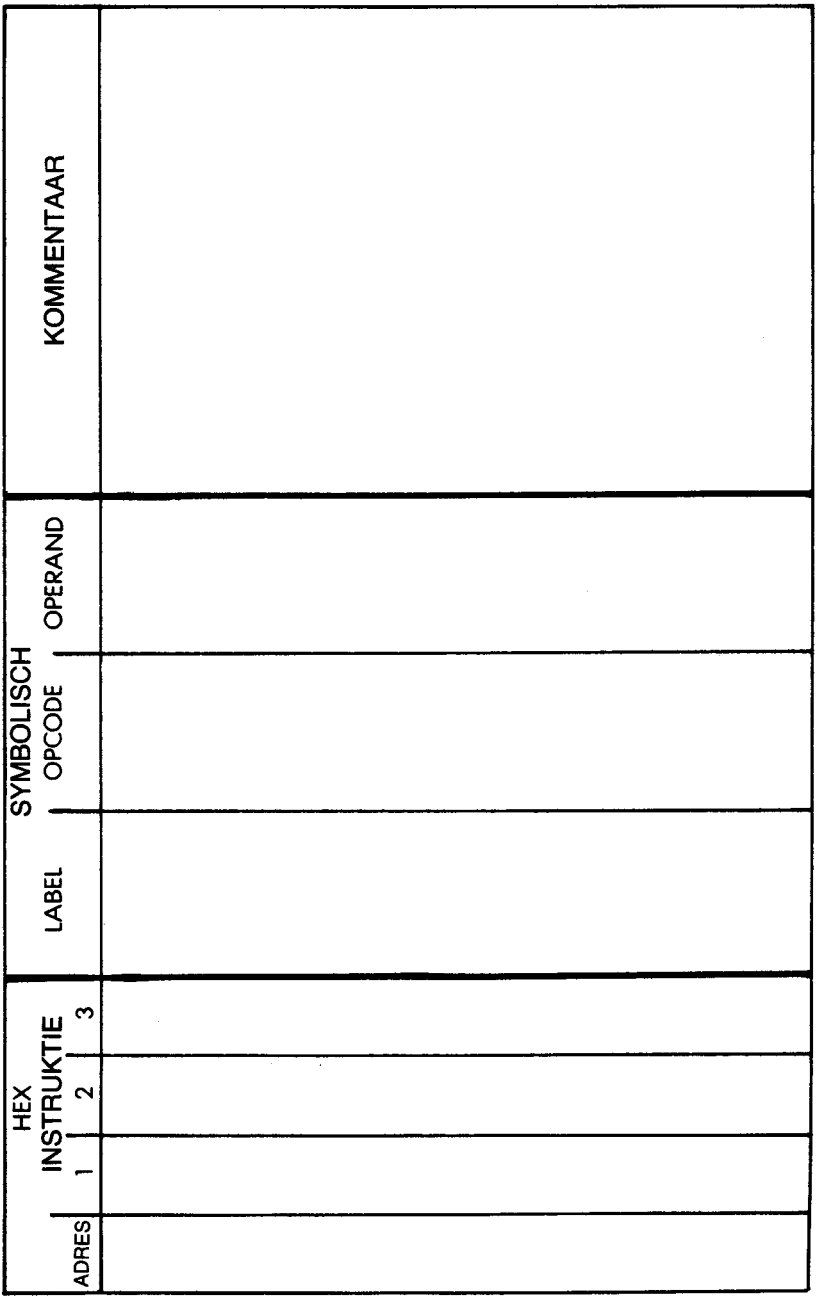


Fig. 10-5: Microprocessor programmeerformulier

Het instruktieveld, dat de opcode bevat en eventuele operanden. (Er kan een apart operandveld onderscheiden worden.)

Het kommentaarveld, meest rechts, optioneel, kan gebruikt worden om het programma te verduidelijken.

Als het programma eenmaal ingevoerd is in de assembler, produceert de assembler er een *listing* van. Bij het maken van een listing, genereert de assembler meestal nog drie extra velden, links van de pagina. Onderstaand is een voorbeeld gegeven: Helemaal links is het regelnummer. Iedere regel die door de programmeur is ingetypt, krijgt een symbolisch regelnummer toegewezen.

Het volgende veld is het eigenlijke adresveld, met daarin de hexadecimale waarde van de programmateller op dat ogenblik.

Het volgende veld is de hexadecimale voorstelling van de instruktie.

Dit demonstreert een van de mogelijke voordelen van een assembler: zelfs al schrijven we progamma's voor een single-board computer, die alleen hexadecimaal accepteert, zouden we het programma toch in assembler moeten schrijven, vooropgesteld dat we toegang hebben tot een systeem dat met een assembler is uitgerust. We kunnen dan de assembler van het systeem gebruiken. De assembler genereert automatisch de juiste hexadecimale kodering. Daarna typen we gewoon deze hex kodes in in ons systeem. Dit demonstreert in een eenvoudig voorbeeld het nut van extra software hulpmiddelen.

Tabellen

Als een assembler het symbolische programma vertaalt naar binair, verricht hij twee wezenlijke taken:

1 - Hij vertaalt de mnemonische instrukties in hun binaire kodering
2 - Hij vertaalt de symbolen die voor adressen en konstantes gebruikt worden in hun binaire representatie.

Om het verbeteren van een programma te vereenvoudigen, toont de assembler aan het einde van de listing de equivalenten van de symbolen en de hex waarde. Dit wordt de symbooltabel genoemd.

Sommige symbooltabellen vermelden niet alleen de hex waarde van de symbolen, maar ook de regelnummers waar ze voorkomen, een extra voorziening.

Foutboodschappen

Tijdens het assembleerproces detekteert de assembler syntaktische fouten en geeft ze als een deel van de listing. Typische diagnoses zijn: ongedefinieerde symbolen, reeds gedefinieerde labels, ongeldige opcodes, ongeldig adres, ongeldige adresseermode. Een meer gedetailleerde diagnose is natuurlijk wenselijk, en hierin is vaak wel voorzien. Dit varieert per assembler.

```
LINE # LOC     CODE       LINE

0057  0342  A9 00            LDA #$00
0058  0344  8D 0B A0         STA ACR1          ;TURN BOTH TIMERS OFF
0059  0347  8D 08 AC         STA ACR2
0060  034A  A2 20            LDX #OFFDEL       ;GET TONES-OFF DELAY CONSTANT
0061  034C  20 55 03   OFF    JSR DELAY         ;DELAY WHILE TONE IS OFF
0062  034F  CA               DEX
0063  0350  D0 FA            BNE OFF
0064  0352  4C 02 03         JMP DIGIT         ;GO BACK FOR NEXT DIGIT OF PHONE NUMBER
0065  0355                  ;
0066  0355                  ;THIS IS A SIMPLE DELAY ROUTINE FOR THE TONE ON AND OFF PERI
0067  0355                  ;
0068  0355  A9 FF      DELAY  LDA #DELCON       ;GET DELAY CONSTANT
0069  0357  38        WAIT   SEC               ;DELAY FOR THAT LONG
0070  0358  E9 01            SBC #$01
0071  035A  D0 FB            BNE WAIT
0072  035C  60               RTS
0073  035D                  ;
0074  035D                  ;THIS IS A TABLE OF THE CONSTANTS FOR THE TONE FREQUENCIES
0075  035D                  ;FOR EACH TELEPHONE DIGIT.  THE CONSTANTS ARE TWO BYTES
0076  035D                  ;LONG, LOW BYTE FIRST.
0077  035D                  ;
0078  035D  13        TABLE  .BYTE $13,$02,$76,$01 ;TWO TONES FOR '0'
0078  035E  02
0078  035F  76
0078  0360  01
0079  0361  CD               .BYTE $CD,$02,$9E,$01 ;TWO TONES FOR '1'
0079  0362  02
0079  0363  9E
0079  0364  01
0080  0365  CD               .BYTE $CD,$02,$76,$01 ;      '2'
0080  0366  02
0080  0367  76
0080  0368  01
0081  0369  CD               .BYTE $CD,$02,$53,$01 ;           '3'
0081  036A  02
0081  036B  53
0081  036C  01
0082  036D  89               .BYTE $89,$02,$9E,$01 ;      '4'
0082  036E  02
0082  036F  9E
0082  0370  01
0083  0371  89               .BYTE $89,$02,$76,$01 ;      '5'
0083  0372  02
0083  0373  76
0083  0374  01
0084  0375  89               .BYTE $89,$02,$53,$01 ;      '6'
0084  0376  02
0084  0377  53
0084  0378  01
0085  0379  4B               .BYTE $4B,$02,$9E,$01 ;      '7'
0085  037A  02
0085  037B  9E
0085  037C  01
0086  037D  4B               .BYTE $4B,$02,$76,$01 ;    '8'
0086  037E  02


LINE # LOC     CODE       LINE

0086  037E  26
0086  0380  01
0087  0381  4B               .BYTE $4B,$02,$53,$01 ;        '9'
0087  0382  02
0087  0383  53
0087  0384  01
0088  0385                  .END

SYMBOL TABLE

SYMBOL    VALUE

ACR1      A00B    ACR2      A00B    DELAY     0355    DELCON    00FF
DIGIT     0302    NDEND     030A    NUMPTR    0000    OFF       034C
OFFDEL    0020    ON        033C    ONDEL     0040    PHONE     0100
T1CH      A005    T1LH      A007    T1LL      A004    T2CH      A005
T2LH      A007    T2LL      A004    TABLE     035D    WAIT      0357

END OF ASSEMBLY
```

Fig. 10-6: Assemblerlisting: Een voorbeeld

De assembleertaal

Opcodes hebben we al gedefinieerd. We zullen hier de symbolen, konstanten en operatoren definieren die als een deel van de assembler-syntax gebruikt kunnen worden.

Symbolen

Symbolen worden gebruikt om numerieke waarden, gegevens of adressen, voor te stellen. Gewoonlijk mogen symbolen uit maximaal 6 karakters bestaan en moeten met een letter beginnen. Er is nog een beperking: de 56 opcodes van de 6502 en de namen van de registers, d.w.z. A, X, Y, S, P mogen niet gebruikt worden als symbolen.

Toekennen van een waarde aan een symbool

Labels zijn speciale symbolen waarvan de waarde niet door de programmeur toegekend hoeft te worden. Ze korresponderen automatisch met het regelnummer waar ze in voorkomen. Andere symbolen zoals konstanten of geheugenadressen moeten door de programmeur voor hun gebruik gespecificeerd worden. Voor dit doel wordt het gelijkteken (=), of anders een speciale *pseudo-instruktie*. Het is een instruktie voor de assembler die niet in een uitvoerbare opdracht vertaald wordt: het is een assembler pseudo-instruktie.

De konstante ALPHA wordt bijvoorbeeld gedefinieerd als:

```
ALPHA = $A000
```

Deze pseudo-instruktie kent de waarde "A000" hexadecimaal toe aan de variabele ALPHA. De assembler pseudo-instrukties worden verderop in dit hoofdstuk beschouwd.

Konstanten (Eng: literals)

Konstanten kunnen als gewoonlijk decimaal, hexadecimaal, octaal of binair uitgedrukt worden. Om het grondtal waarin het getal gerepresenteerd is aan te geven wordt een prefix gebruikt. Bij een decimaal getal wordt geen prefix gebruikt. Om 18 in de accumulator te laden schrijven we:

```
LDA #18 (# geeft een konstante aan)
```

Een hexadecimaal getal wordt voorafgegaan door het symbool $.
Een octaal symbool wordt voorafgegaan door het symbool @
Een binair symbool wordt voorafgegaan door %.

Om bijvoorbeeld "11111111" in de accumulator te laden, schrijven we:

```
LDA #%11111111
```

Konstante ASCII karakters kunnen ook gebruikt worden in een konstantenveld. In oudere assemblers was het de gewoonte om het ASCII symbool in aanhalingstekens (") in te sluiten. In meer recente assemblers wordt het alfanumerieke type vooafgegaan door een enkel aanhalingsteken, om het aantal in te typen karakters te verminderen.

Om bijvoorbeeld het symbool "S" in de accumulator te laden (in ASCII), schrijven we:

```
LDA #'S
```

Om het aanhalingsteken zelf te kunnen laden is de afspraak:

```
LDA #'"
```

Oefening 10.1: Zullen de volgende twee instrukties de zelfde waarde in de accumulator laden: LDA #'5 en LDA #$5?

Operatoren

Om het schrijven van een symbolisch programma nog verder te vereenvoudigen, staan de meeste assemblers het gebruik van operatoren toe. Op zijn minst moeten dit plus en min zijn, zodat bijvoorbeeld

```
LDA ADR1, en
LDX ADR1 + 1
```

gespecificeerd kunnen worden.

Het is belangrijk om te begrijpen, dat de uitdrukking ADR1 + 1 door de assembler berekend wordt om te bepalen welk geheugenadres als binair equivalent gebruikt moet worden. Dit wordt berekend bij het assembleren, en niet tijdens het uitvoeren.

Verder kunnen er nog meer operatoren zijn, zoals vermenigvuldigen en delen, die erg handig zijn bij het gebruik van tabellen in het geheu-

gen. Nog meer gespecialiseerde operatoren kunnen aanwezig zijn, zoals bijvoorbeeld groter en kleiner dan.

Een uitdrukking moet natuurlijk op een positieve waarde uitkomen. Negatieve getallen kunnen meestal niet gebruikt worden en moeten hexadecimaal uitgedrukt worden.

Tenslotte wordt er een speciaal symbool gebruikt om het huidige regelnummer aan te geven: *. Dit symbool moet geinterpreteerd worden als "huidige locatie" (waarde van PT).

Oefening 10.2: Wat is het verschil tussen de volgende twee instrukties?:

```
LDA %10101010
LDA #%10101010
```

Oefening 10.3: Wat is het effekt van de volgende instruktie?

```
BMI* −2
```

Pseudo-instrukties

Pseudo-instrukties zijn speciale opdrachten van de programmeur aan de assembler om symbolen of het geheugen een waarde te geven, of die gebruikt worden om het uitvoeren of het printen van de assembler te besturen.

Om een specifiek voorbeeld te geven, zullen we hier de 9 pseudo-instrukties van het Rockwell Development System ("System 65") beschrijven. Dit zijn: .BYT, .WOR, .GBY, .PAGE, .SKIP, .OPT, .FILE en .END

Toewijzings pseudo-instrukties

Een gelijkteken wordt gebruikt om een numerieke waarde aan een symbool toe te kennen. Bijvoorbeeld:

```
BASE = $1111
* = $1234
```

Het effekt van de eerste instruktie is het toekennen van de waarde 1111 hex aan BASE.

Het effekt van de tweede instruktie is het forceren van het regelnummer naar waarde 1234 hex. Met andere woorden: de volgende uitvoerbare instruktie wordt geplaatst op geheugenadres 1234 hexadecimaal.

Oefening 10.4: Schrijf een pseudo-instruktie die zorgt dat het programma begint op adres 0.

Pseudo-instrukties om het geheugen te initialiseren

Hiervoor zijn drie pseudo-instrukties beschikbaar: .BYT, .WOR, .GBY.
.BYT kent de karakters die volgen toe aan opeenvolgende geheugenbytes.

Voorbeeld: RESERV .BYT 'SYBEX'.

Dit heeft tot gevolg, dat de letters "SYBEX" in opeenvolgende geheugenplaatsen opgeslagen worden.

.WOR wordt gebruikt om twee-bytes adressen in het geheugen op te slaan, het laagste byte eerst.

Voorbeeld: .WOR #1234, #2345

.GBY is identiek met .BYT, behalve dat het 16 bits waarden opslaat, hoge byte eerst. Het wordt gewoonlijk gebruikt voor 16-bits gegevens in plaats van 16-bits adressen.

Invoer/uitvoer pseudo-instrukties

Dit zijn: .PAGE, .SKIP, .OPT.

.PAGE zorgt dat de assembler de pagina afmaakt, d.w.z. naar de eerste regel van een nieuwe pagina gaat. Verder kan er een titel voor de pagina gespecificeerd worden. Voorbeeld: .PAGE 'pagina titel'

.SKIP wordt gebruikt om blanco regels te genereren in de listing. Het aantal regels kan gespecificeerd worden. Bijvoorbeeld: .SKIP 3

.OPT specificeert vier opties: list, generate, errors, symbols.

Generate (genereer) wordt gebruikt om object kode te genereren voor strings met de .BYT instruktie.
Error (fout) geeft aan of diagnostische fouten afgedrukt moeten worden.

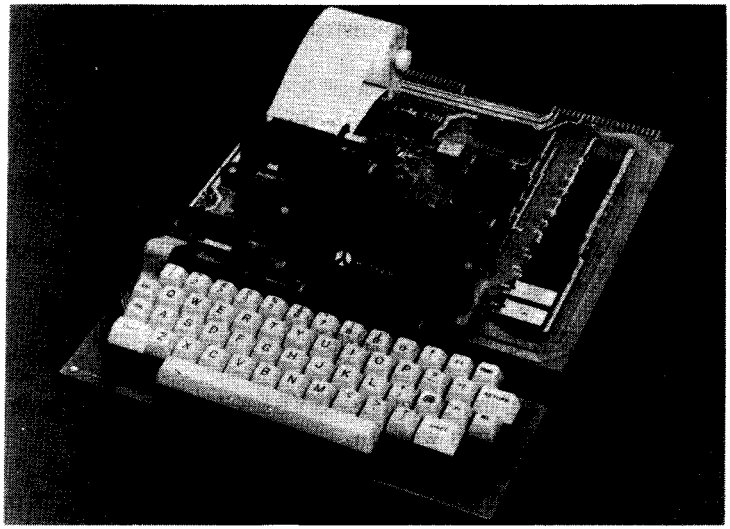


Fig. 10-7: AIM65 is een board met een miniprinter en volledig toetsenbord

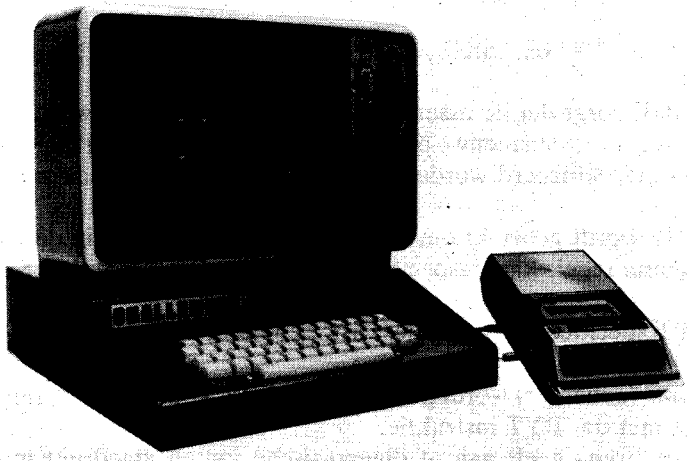


Fig. 10-8: Ohio Scientific is een Personal Microcomputer

Symbol (symbool) geeft aan of er een listing van de symbooltabel gemaakt moet worden.

De laatste twee pseudo-instrukties geven het formaat van de assemblerlisting aan: .FILE en .END.

Bij de ontwikkeling van grote programma's zullen delen van het programma apart geschreven en verbeterd worden. Op een gegeven ogenblik zullen deze delen (files) weer aan elkaar geknoopt moeten worden. De laatste opdracht van de eerste file zal dan de pseudo-instruktie .FILE NAAM/1 bevatten, waarin 1 het nummer van de schijfeenheid is en NAAM de naam van de volgende file. De volgende file kan op zijn beurt weer met andere files verbonden zijn. Aan het einde van de laatste file zal de pseudo-instruktie .END NAAM/1 staan, wat een wijzer terug naar de eerste file is.

Tenslotte is er de mogelijkheid om extra kommentaar in de listing tussen te voegen: ";".

";" kan gebruikt worden om naar wens kommentaar toe te voegen in plaats van instrukties. Dit is een belangrijke voorziening om goed gedokumenteerde programma's te maken.

MACRO's

Een macrovoorziening is meestal nog niet aanwezig op de bestaande 6502 assemblers. We zullen hier echter definieren wat een macro is, en waar hij voor gebruikt kan worden.
Het is te hopen dat er snel een macro-voorziening op de 6502 assemblers komt.

Een macro is gewoon een naam die toegekend is aan een groep instrukties. Een macro is er voornamelijk voor het gemak van de programmeur. Als bijvoorbeeld een groep van vijf instrukties op meerdere plaatsen in het programma gebruikt wordt, zouden we een macro kunnen definieren, in plaats dat we steeds die vijf instrukties schrijven. We zouden bijvoorbeeld kunnen schrijven:

```
REDREG MACRO PHA
               TXA
               PHA
               TYA
               PHA
       ENDM
```

En dan schrijven we de naam REDREG in plaats van die vijf instrukties.

Iedere keer als we de naam REDREG schrijven, worden de vijf korresponderende instrukties gesubstitueerd voor die naam. Een assembler met deze voorziening wordt een macro-assembler genoemd.

Macro of subroutine

Nu lijkt het of een macro hetzelfde is als een subroutine. Dat is niet het geval. Als de assembler gebruikt wordt om object kode te genereren, wordt iedere keer de macro vervangen door de vijf instrukties. Ten tijde van de uitvoering komen deze vijf instrukties even vaak voor als oorspronkelijk de macronaam.
In tegenstelling hiermee wordt de subroutine maar eenmaal gedefinieerd, en wordt vervolgens meermalen gebruikt: het programma springt naar het subroutineadres. Een macro wordt een *assembleertijd* voorziening genoemd, een subroutine is een *uitvoeringstijd* voorziening. Hun werking is heel verschillend.

Macro parameters

Iedere macro kan met een aantal parameters uitgerust zijn. Laten we als voorbeeld de volgende macro nemen:

```
SWAP  MACRO  M,N,T
      LDA    M
      STA    T
      LDA    N
      STA    M
      LDA    T
      STA    N
      ENDM
```

Deze macro verwisselt de inhoud van de registers M en N. Een verwisseling tussen twee of meer registers of geheugenplaatsen is geen voorziening van de 6502. Om dit uit te voeren kan een macro gebruikt worden. "T" is bijvoorbeeld gewoon een naam voor een tijdelijke geheugenplaats van het programma.
Laten we als voorbeeld de inhoud van de geheugenplaatsen ALPHA

en BETA omwisselen. De instruktie die dit doet is onderstaand gegeven:

SWAP MACRO ALPHA, BETA, TEMP

In deze instruktie is TEMP een tijdelijke geheugenplaats, waarvan we weten dat hij beschikbaar is en door de macro gebruikt kan worden. De resulterende expansie van de macro wordt:

```
LDA   ALPHA
STA   TEMP
LDA   BETA
STA   ALPHA
LDA   TEMP
STA   BETA
```

Het nut van een macro zou nu duidelijk moeten zijn: het is gemakkelijk voor de programmeur om pseudo-instrukties te gebruiken, die door macro's gedefinieerd zijn. Op deze manier kan de schijnbare instruktieset van de 6502 naar wens uitgebreid worden. Helaas moet men in gedachten houden, dat iedere macro geexpandeerd wordt tot het aantal gebruikte instrukties. Een macro wordt daarom langzamer uitgevoerd dan een enkele instruktie. Vanwege het gemak bij het schrijven van en lang programma is een macro faciliteit zeer wenselijk voor dergelijke toepassingen.

Verdere macro-faciliteiten

De eenvoudige macro-faciliteit kan nog uitgebreid worden met vele andere pseudo-instrukties en syntaktische faciliteiten: macro's kunnen *genest* worden, d.w.z. een macro kan aangeroepen worden binnen een macro-definitie. Door van deze faciliteit gebruik te maken, kan een macro zichzelf veranderen met een geneste definitie! Een eerste aanroep produceert een expansie, terwijl verdere aanroepen een gewijzigde expansie produceren van dezelfde macro.

VOORWAARDELIJK ASSEMBLEREN

Voorwaardelijk assembleren is een andere voorziening waarin door de meeste 6502 assemblers nog niet voorzien is. Een voorwaardelijke assembleerfaciliteit stelt de programmeur in de gelegenheid om de spe-

ciale instrukties "IF", gevolgd door een expressie, dan (optioneel) "ELSE", en beeindigd met "ENDIF" te gebruiken. Als de expressie die volgt op de "IF" waar is, worden de instrukties tussen de "IF" en de "ELSE", of die tussen de "IF" en de "ENDIF" (als er geen "ELSE" is), geassembleerd. Als de IF gevolg door een ELSE gebruikt wordt, wordt een van de twee blokken instrukties geassembleerd, afhankelijk van de waarde van de expressie, die getest wordt.

Met voorwaardelijk assembleren kan de programmeur programma's ontwerpen voor een varieteit van gevallen, en dan voorwaardelijk de kodesegmenten assembleren die nodig zijn voor en bepaalde toepassing. Zo kan bijvoorbeeld een industriele gebruiker programma's ontwerpen die een varierend aantal verkeerslichten bij een kruispunt sturen, volgens verschillende algoritmes. Hij ontvangt dan de specificaties van de plaatselijke verkeersdeskundige, die aangeven hoeveel verkeerslichten er gebruikt moeten worden en volgens welke algoritmes ze moeten werken. De programmeur hoeft dan slechts de parameters in zijn programma in te voeren en voorwaardelijk te assembleren. De voorwaardelijke assemblage heeft dan een "op maat gesneden" programma tot gevolg, dat alleen die routines bevat, die voor de oplossing van het probleem nodig zijn.

Voorwaardelijk assembleren is dus vooral van belang voor het genereren van programma's in een industriele omgeving, waar vele opties zijn en waar er snel delen van programma's geassembleerd moeten worden, afhankelijk van externe variabelen.

SAMENVATTING

In dit hoofdstuk zijn de verschillende technieken, hardware en software hulpmiddelen, samen met hun voor- en nadelen behandeld, die nodig zijn voor het ontwikkelen van programma's.

Wat de hardware betreft varieert dit van de single-board microcomputer tot het volledige ontwikkelsysteem, en wat software betreft van binaire kodering tot programmeren in hogere talen.

Je zult hier en keuze uit moeten maken, afhankelijk van je doelstellingen en middelen.

11
KONKLUSIE

We hebben nu alle belangrijke aspekten van het programmeren behandeld, van de definities en de grondbeginselen tot de manipulatie van de interne 6502 registers, tot het behandelen van de I/O-apparatuur, en tot de karakteristieken van software ontwikkelhulpen. Wat is de volgende stap? We kunnen twee invalshoeken beschouwen, die van de technologische ontwikkeling, en die van het ontwikkelen van je eigen kennis en vaardigheden. We zullen deze twee punten eens nader bespreken.

TECHNOLOGISCHE ONTWIKKELING

Door de vooruitgang in de MOS integratietechnieken, kunnen steeds complexere chips gemaakt worden. De kosten van het uitvoeren van de processorfunktie worden steeds lager. Het gevolg is dat veel I/O-chips of de randapparatuur kontrolechips nu vaak een eenvoudige processor huisvesten. Dit betekent, dat de meeste LSI chips in het systeem nu *programmeerbaar* worden. Er onstaat nu een interessant dilemma: om het programmeren te vereenvoudigen en het aantal chips in het systeem te verminderen, zijn de nieuwe I/O chips uitgerust met geavanceerde programmeerbare mogelijkheden: veel geprogrammeerde algoritmes zijn nu in de chip geintegreerd. Het resultaat is echter, dat het ontwikkelen van programma's moeilijker is geworden, omdat al deze chips verschillend zijn, en gedetailleerd bestudeerd moeten worden door de programmeur!

Het programmeren van een systeem behelst niet meer alleen het pro-

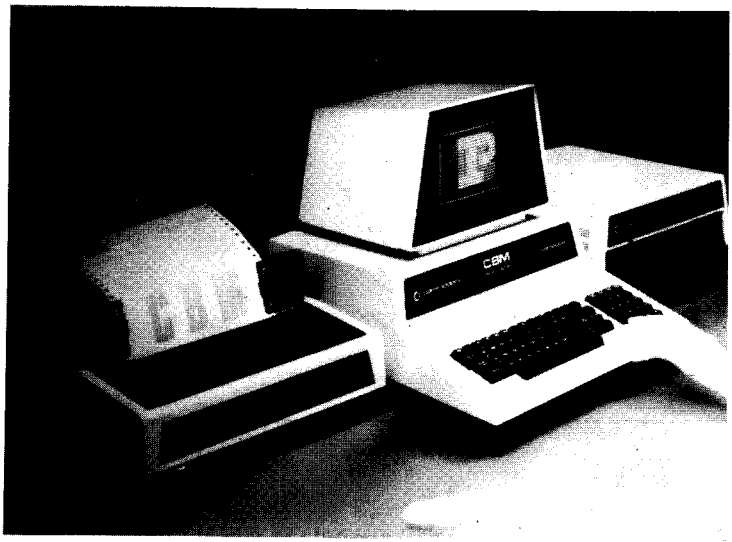


Fig. 11-1: PET is een geïntegreerde eenheid

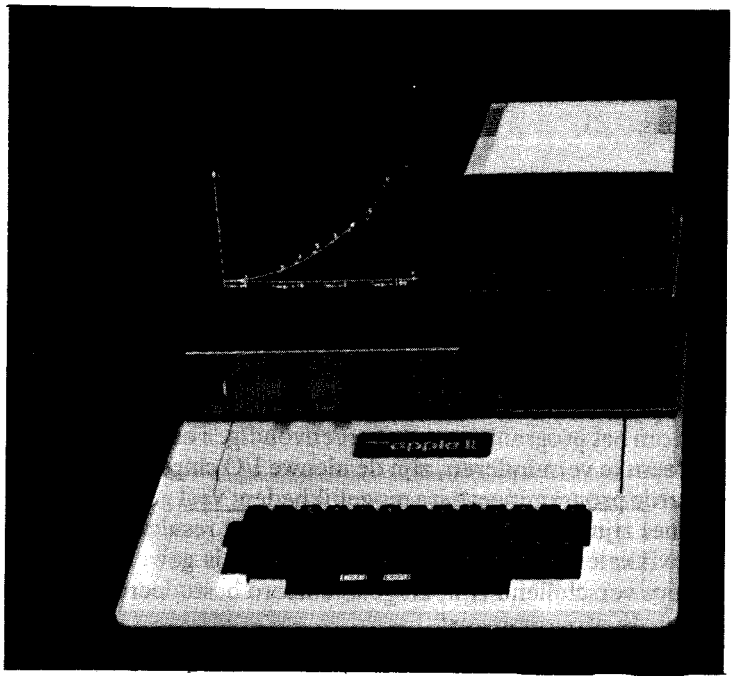


Fig. 11-2: APPLE II gebruikt een gewone TV

grammeren van de microprocessor, maar ook het programmeren van de verschillende andere chips die ermee verbonden zijn. De tijd die nodig is om een chip te leren kennen kan aanzienlijk zijn.

Dit is natuurlijk maar een schijnbaar dilemma. Als deze chips niet beschikbaar waren, zou de mate van ingewikkeldheid van het te realiseren interface en de nodige programmatuur nog groter zijn.

Het nieuwe probleem dat geintroduceerd is, is dat er nu meer dan alleen een processor geprogrammeerd moet worden, en dat de verschillende kenmerken van een chips bestudeerd moeten worden om er doelmatig gebruik van te kunnen maken. We hopen echter dat de in dit boek gepresenteerde technieken en begrippen dit tot een redelijk eenvoudige taak maken.

DE VOLGENDE STAP

Je hebt nu de basistechnieken geleerd die nodig zijn om eenvoudige toepassingen op papier te programmeren. Dat was de doelstelling van dit boek. De volgende stap is het eigenlijke oefenen. Daarvoor is er geen vervanging. Het is onmogelijk om op papier programmeren te leren, en ervaring is een vereiste. Je zou nu in de positie moeten zijn om te beginnen met het schrijven van je eigen programma's. We hopen dat je dat een aangename bezigheid zult vinden.

Voor hen, die een begeleidend boek nuttig vinden, is er een bijbehorend deel in deze serie, het '6502 Applications Book' (ref 302), waarin veel toepassingen gepresenteerd worden, die op een echte microcomputer uitgevoerd kunnen worden.

Verder is er het 'Advanced 6502 Programming' (ref. G402A), waarin programmeertechnieken voor complexe algoritmes gegeven worden. Er is ook een 6502 assembler, geschreven in standaard Microsoft BASIC, verkrijgbaar.

BIJLAGE A

HEXADECIMAL OMZETTINGSTABEL

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393.216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

BIJLAGE B

6502 INSTRUKTIES–ALFABETISCH

ADC	Add with carry	**JSR**	Jump to subroutine
AND	Logical AND	**LDA**	Load accumulator
ASL	Arithmetic shift left	**LDX**	Load X
BCC	Branch if carry clear	**LDY**	Load Y
BCS	Branch if carry set	**LSR**	Logic
BEQ	Branch if result = 0	**NOP**	No operation
BIT	Test bit	**ORA**	Logical OR
BMI	Branch if minus	**PHA**	Push A
BNE	Branch if not equal to 0	**PHP**	Push P status
BPL	Branch if plus	**PLA**	Pull A
BRK	Break	**PLP**	Pull P status
BVC	Branch if overflow clear	**ROL**	Rotate left
BVS	Branch if overflow set	**ROR**	Rotate right
CLC	Clear carry	**RTI**	Return from interrupt
CLD	Clear decimal flag	**RTS**	Return from subroutine
CLI	Clear interrupt disable	**SBC**	Subtract with carry
CLV	Clear overflow	**SEC**	Set carry
CMP	Compare to accumulator	**SED**	Set decimal
CPX	Compare to X	**SEI**	Set interrupt disable
CPY	Compare to Y	**STA**	Store accumulator
DEC	Decrement memory	**STX**	Store X
DEX	Decrement X	**STY**	Store Y
DEY	Decrement Y	**TAX**	Transfer A to X
EOR	Exclusive OR	**TAY**	Transfer A to Y
INC	Increment memory	**TSX**	Transfer SP to X
INX	Increment X	**TXA**	Transfer X to A
INY	Increment Y	**TXS**	Transfer X to SP
JMP	Jump	**TYA**	Transfer Y to A

BIJLAGE C

BINAIRE LIJST VAN DE 6502 INSTRUKTIES

ADC	011bbb01	LDX	101bbb10
AND	001bbb01	LDY	101bbb00
ASL	000bbb10	LSR	01bbbb10
BCC	10110000	NOP	01bbb110
BEQ	11110000	ORA	000bbb01
BIT	0010b100	PHP	01001000
BMI	00110000	PHA	00001000
BNE	11010000	PLA	01101000
BPL	00010000	PLP	00101000
BRK	01010000	ROL	001bbb10
CLC	00011000	ROR	011bbb10
CLD	11011000	RTI	01000000
CLI	01011000	RTS	01100000
CMP	110bbb01	SBC	111bbb01
CPX	1110bb00	SEC	00111000
CPY	1100bb00	SED	11111000
DEC	110bb110	SEI	01111000
DEX	11001010	STA	100bbb01
DEY	10001000	STX	100bb110
EOR	**010bbb01**	STY	100bb100
INC	111bb110	TAX	10101010
INX	11101000	TAY	10101000
INY	11001000	TSX	10111010
JMP	01b01100	TXA	10001010
JSR	00100000	TXS	10011010
LDA	101bbb01	TYA	10011000

Zie Hfdst. 4 voor één definitie van de "bb" veld.

BIJLAGE D

6502–INSTRUKTIESET: HEX EN TIJD

		IMPLIED			ACCUM.			ABSOLUTE			ZERO PAGE			IMMEDIATE			ABS. X			ABS. Y		
MNEMONIC		OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#
ADC	(1)							6D	4	3	65	3	2	69	2	2	7D	4	3	79	4	3
AND	(1)							2D	4	3	25	3	2	29	2	2	3D	4	3	39	4	3
ASL					0A	2	1	0E	6	3	06	5	2				1E	7	3			
BCC	(2)																					
BCS	(2)																					
BEQ	(2)																					
BIT								2C	4	3	24	3	2									
BMI	(2)																					
BNE	(2)																					
BPL	(2)																					
BRK		00	7	1																		
BVC	(2)																					
BVS	(2)																					
CLC		18	2	1																		
CLD		D8	2	1																		
CLI		58	2	1																		
CLV		B8	2	1																		
CMP								CD	4	3	C5	3	2	C9	2	2	DD	4	3	D9	4	3
CPX								EC	4	3	E4	3	2	E0	2	2						
CPY								CC	4	3	C4	3	2	C0	2	2						
DEC								CE	6	3	C6	5	2				DE	7	3			
DEX		CA	2	1																		
DEY		88	2	1																		
EOR	(1)							4D	4	3	45	3	2	49	2	2	5D	4	3	59	4	3
INC								EE	6	3	E6	5	2				FE	7	3			
INX		E8	2	1																		
INY		C8	2	1																		
JMP								4C	3	3												
JSR								20	6	3												
LDA	(1)							AD	4	3	A5	3	2	A9	2	2	BD	4	3	B9	4	3
LDX	(1)							AE	4	3	A6	3	2	A2	2	2				BE	4	3
LDY	(1)							AC	4	3	A4	3	2	A0	2	2	BC	4	3			
LSR					4A	2	1	4E	6	3	46	5	2				5E	7	3			
NOP		EA	2	1																		
ORA								0D	4	3	05	3	2	09	2	2	1D	4	3	19	4	3
PHA		48	3	1																		
PHP		08	3	1																		
PLA		68	4	1																		
PLP		28	4	1																		
ROL					2A	2	1	2E	6	3	26	5	2				3E	7	3			
ROR					6A	2	1	6E	6	3	66	5	2				7E	7	3			
RTI		40	6	1																		
RTS		60	6	1																F9	4	3
SBC	(1)							ED	4	3	E5	3	2	E9	2	2	FD	4	3			
SEC		38	2	1																		
SED		F8	2	1																		
SEI		78	2	1																		
STA								8D	4	3	85	2					9D	5	3	99	5	3
STX								8E	4	3	86	2										
STY								8C	4	3	84	2										
TAX		AA	2	1																		
TAY		A8	2	1																		
TSX		BA	2	1																		
TXA		8A	2	1																		
TXS		9A	2	1																		
TYA		98	2	1																		

(1) *Add 1 to n if crossing page boundary*

(IND. X)			(IND)Y			Z. PAGE, X			RELATIVE			INDIRECT			Z. PAGE, Y			PROCESSOR STATUS CODES									
OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	N	V		B	D	I	Z	C	MNEMONIC
61	6	2	71	5	2	75	4	2										•	•					•	•	ADC
21	6	2	31	5	2	35	4	2										•						•		AND
						16	6	2										•						•	•	ASL
									90	2	2															BCC
									B0	2	2															BCS
									F0	2	2															BEQ
																		M7	M6					•		BIT
									30	2	2															BMI
									D0	2	2															BNE
									10	2	2															BPL
																					1		1			BRK
									50	2	2															BVC
									70	2	2															BVS
																									0	CLC
																						0				CLD
																							0			CLI
																			0							CLV
C1	6	2	D1	5	2	D5	4	2										•						•	•	CMP
																		•						•	•	CPX
																		•						•	•	CPY
						D6	ø6	2										•						•		DEC
																		•						•		DEX
																		•						•		DEY
41	6	2	51	5	2	55	4	2										•						•		EOR
						F6	6	2										•						•		INC
																		•						•		INX
																		•						•		INY
												6C	5	3												JMP
																										JSR
A1	6	2	B1	5	2	B5	4	2										•						•		LDA
															B6	4	2	•						•		LDX
						B4	4	2										•						•		LDY
						56	6	2										0						•	•	LSR
																										NOP
01	6	2	11	5	2	15	4	2										•						•		ORA
																										PHA
																										PHP
																		•						•		PLA
																		•	•	•	•	•	•	•	•	PLP
						36	6	2										•						•	•	ROL
						76	6	2										•						•	•	ROR
																		•	•	•	•	•	•	•	•	RTI
																										RTS
E1	6	2	F1	5	2	F5	4	2										•	•					•	•	SBC
																									1	SEC
																										SED
																										SEI
81	6	2	91	6	2	95	4	2																		STA
															96	4	2									STX
						94	4	2																		STY
																		•						•		TAX
																		•						•		TAY
																		•						•		TSX
																		•						•		TXA
																										TXS
																		•						•		TYA

(2) Add 2 to n if branch within page
Add 3 to n if branch to another page

BIJLAGE E

ASCII OMZETTINGS TABEL

HEX	MSD	0	1	2	3	4	5	6	7
LSD	BITS	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

DE ASCII SYMBOLEN

NUL —Null
SOH —Start of Heading
STX —Start of Text
ETX —End of Text
EOT —End of Transmission
ENQ —Enquiry
ACK —Acknowledge
BEL —Bell
BS —Backspace
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
DC —Device Control
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 —Space (Blank)
DEL —Delete

BIJLAGE F

RELATIEVE SPRONG TABELLEN

FORWARD RELATIVE BRANCH

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

BACKWARD RELATIVE BRANCH TABLE

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

BIJLAGE G:

HEX OPCODE LIJST

LSD / MSD	0	1	2	3	4	5	6	7
0	BRK	ORA-I, X				ORA Ø,P	ASL Ø,P	
1	BPL	ORA-I, Y				ORA Ø,P, X	ASL Ø,P, X	
2	JSR	AND-I, X			BIT Ø,P	AND Ø,P	ROL Ø,P	
3	BMI	AND-I, Y				AND Ø,P, X	ROL Ø,P, X	
4	RTI	EOR-I, X				EOR Ø,P	LSR Ø,P	
5	BVC	EOR-I, Y				EOR Ø,P, X	LSR Ø,P, X	
6	RTS	ADC-I, X				ADC Ø,P	ROR Ø,P	
7	BVS	ADC-I, Y				ADC Ø,P, X		
8		STA-I, X			STY Ø,P	STA Ø,P	STX Ø,P	
9	BCC	STA-I, Y			STY Ø,P, X	STA Ø,P, X	STX Ø,P, Y	
A	LDY-IMM	LDA-I, X	LDX-IMM		LDY Ø,P	LDA Ø,P	LDX Ø,P	
B	BCS	LDA-I, Y			LDY Ø,P, X	LDA Ø,P, X	LDX Ø,P, Y	
C	CPY-IMM	CMP-I, X			CPY Ø,P	CMP Ø,P	DEC Ø,P	
D	BNE	CMP-I, Y				CMP Ø,P, X	DEC Ø,P, X	
E	CPX-IMM	SBC-I, X			CPX Ø,P	SBC Ø,P	INC Ø,P	
F	BEQ	SBC-I, Y				SBC Ø,P, X	INC Ø,P, X	

8	9	A	B	C	D	E	F	LSD / MSD
PHP	ORA-IMM	ASL-A			ORA	ASL		0
CLC	ORA, Y				ORA, X	ASL, X		1
PLP	AND-IMM	ROL-A		BIT	AND	ROL		2
SEC	AND, Y				AND, X	ROL, X		3
PHA	EOR-IMM	LSR-A		JMP	EOR	LSR		4
CLI	EOR, Y				EOR, X	LSR, X		5
PLA	ADC-IMM	ROR-A		JMP-I	ADC	ROR		6
SEI	ADC, Y				ADC, X			7
DEY		TXA		STY	STA	STX		8
TYA	STA, Y	TXS			STA, X			9
TAY	LDA-IMM	TAX		LDY	LDA	LDX		A
CLV	LDA, Y	TSX		LDY, X	LDA, X	LDX, Y		B
INY	CMP-IMM	DEX		CPY	CMP	DEC		C
CLD	CMP, Y				CMP, X	DEC, X		D
INX	SBC-IMM	NOP		CPX	SBC	INC		E
SED	SBC, Y				SBC, X	INC, X		F

I = indirect

Ø,P = zero page

BIJLAGE H:

DECIMAAL TOT BCD OMZETTING

DECIMAL	BCD	DEC	BCD	DEC	BCD
0	0000	10	00010000	90	10010000
1	0001	11	00010001	91	10010001
2	0010	12	00010010	92	10010010
3	0011	13	00010011	93	10010011
4	0100	14	00010100	94	10010100
5	0101	15	00010101	95	10010101
6	0110	16	00010110	96	10010110
7	0111	17	00010111	97	10010111
8	1000	18	00011000	98	10011000
9	1001	19	00011001	99	10011001

INDEX

The SYBEX Library*

Engelstalige microcomputerboeken verkrijgbaar bij onze agent in Nederland.

YOUR FIRST COMPUTER

by Rodnay Zaks 264 pp., 150 illustr., Ref. C200A

The most popular introduction to small computers and their peripherals: what they do and how to buy one.

DON'T (or How to Care for Your Computer)

by Rodnay Zaks 222 pp., 100 illust., Ref. C400

The correct way to handle and care for all elements of a computer system, including what to do when something doesn't work.

INTERNATIONAL MICROCOMPUTER DICTIONARY

140 pp., Ref. X2

All the definitions and acronyms of microcomputer jargon defined in a handy pocket-size edition. Includes translations of the most popular terms into ten languages.

FROM CHIPS TO SYSTEMS: AN INTRODUCTION TO MICROPROCESSORS

by Rodnay Zaks 558 pp., 400 illustr. Ref. C201A

A simple and comprehensive introduction to microprocessors from both a hardware and software standpoint: what they are, how they operate, how to assemble them into a complete system.

INTRODUCTION TO WORD PROCESSING

by Hal Glatzer 216 pp., 140 illustr., Ref. W101

Explains in plain language what a word processor can do, how it improves productivity, how to use a word processor and how to buy one wisely.

INTRODUCTION TO WORDSTAR™

by Arthur Naiman 208 pp., 30 illustr., Ref. W105

Makes it easy to learn how to use WordStar, a powerful word processing program for personal computers.

VISICALC® APPLICATIONS

by Stanley R. Trost 200 pp., Ref. V104

Presents accounting and management planning applications—from financial statements to master budgets; from pricing models to investment strategies.

EXECUTIVE PLANNING WITH BASIC

by X. T. Bui 192 pp., 19 illust., Ref. B380

An important collection of business management decision models in BASIC, including Inventory Management (EOQ), Critical Path Analysis and PERT, Financial Ratio Analysis, Portfolio Management, and much more.

BASIC FOR BUSINESS

by Douglas Hergert 250 pp., 15 illustr., Ref. B390

A logically organized, no-nonsense introduction to BASIC programming for business applications. Includes many fully-explained accounting programs, and shows you how to write them.

FIFTY BASIC EXERCISES

by J. P. Lamoitier 236 pp., 90 illustr., Ref. B250

Teaches BASIC by actual practice, using graduated exercises drawn from everyday applications. All programs written in Microsoft BASIC.