Englisch

Het machinetaal-boek voor de Commodore 64

De machinetaal voor iedereen toegankelijk gemaakt

Commodore Bibliotheek 9

DATA BECKER
NEDERLANDS

Englisch

Het machinetaal-boek voor de Commodore 64

De machinetaal voor iedereen toegankelijk gemaakt

Commodore Bibliotheek 9

DATA BECKER
NEDERLAND$*

Oorspronkelijke titel: Das Maschinensprache Buch zum Commodore 64

Omslagontwerp: Martin van Keulen
Vertaling: ProCread, Groningen
ISBN 90 229 3333 4

De keuze en opbouw van de programma's die in dit boek voorkomen, zijn gebaseerd op hun educatieve waarde. Alle programma's zijn getest op hun juiste werking. De uitgever kan geen enkele verantwoordelijkheid voor eventueel toch optredende fouten aanvaarden.

Boeken en programma's van

DATA BECKER
NEDERLANDS*

worden in de handel gebracht door
A.W. Bruna & Zoons Uitgeversmij b.v.,
Postbus 8411, 3503 RK Utrecht
en
A.W. Bruna & Zoon n.v.
Antwerpsesteenweg 29a, 2630 Aartselaar.

INHOUDSOPGAVE

VOORWOORD

Met het programmeren in machinetaal is het maar vreemd gesteld. Bijna iedereen zou er goed in willen zijn, omdat machinetaal extreem snel en veelzijdig is. Velen proberen het onder het motto: baat het niet, het schaadt ook niet! De meesten geven het snel weer op omdat het toch te ingewikkeld is. Er zijn maar enkelen die het wel lukt en die zijn dan ook te benijden. Met dit boek willen we zoveel mogelijk Commodore 64 gebruikers in staat stellen hun weg in de machinetaal te vinden. Als auteur hebben wij hiervoor Lothar Englisch kunnen inschakelen. Hij heeft tot dusverre niet alleen aan veel van onze boeken meegewerkt, hij geldt ook als een grondige kenner van het operating system van de Commodore en de programmering van alle Commodorecomputers in machinetaal en assembler. De beide populaire en zeer krachtige softwarepakketten PROFIMON 64 en PROFIASS 64 komen uit zijn brein. Deze programma's zijn bij DATA BECKER/NEDERLANDS* verkrijgbaar onder de titel PROFIMAT 64.

Zelf heb ik vroeger dikwijls tevergeefs machinetaal geprobeerd en er vaak de brui aan gegeven. Uiteindelijk heeft het manuscript van dit boek mij geholpen, dat mag niet onvermeld blijven, plus natuurlijk geduld en doorzettingsvermogen. Maar zoals u na het doorwerken van dit boek zeker zult bevestigen, het loont zeker de moeite. Veel plezier met die boek en veel succes met uw eigen machinetaalprogramma's.

Dr. Achim Becker

1 INLEIDING

Als u vandaag de dag een personal of homecomputer zoals de Commodore 64 koopt, beschikken deze machines allemaal over de programmeertaal BASIC. Met BASIC zijn bijna alle voorkomende problemen op te lossen. Het programmeren in BASIC is gemakkelijk te leren. Waarom zou u zich zo nodig nog met machinetaal moeten bezighouden? Is dat niet alleen maar een overblijfsel uit de pionierstijd van de eerste computers?

We vergelijken BASIC met machinetaal. De meesten van u zullen zeker BASIC beheersen en bevestigen dat het programmeren in deze taal niet moeilijk te leren is. Laat u zich echter door dit boek overtuigen dat het programmeren in machinetaal net zo eenvoudig en snel te leren is. U zult ervan kunnen profiteren dat u al BASIC kent. In principe is de manier van werken bij machinetaal niet erg verschillend.

1. Uw Commodore 64 heeft de programmeertaal BASIC. Deze naam is een afkorting van Beginners All Purpose Symbolic Instruction Code, wat ongeveer betekent: symbolische programmmeertaal voor beginners, geschikt voor veel doeleinden. Hoewel het gemakkelijk te leren is, is toch door het grote aantal instructies de taal krachtig, u kunt op een gemakkelijke manier veel bereiken. BASIC behoort tot de zogenaamde hogere programmeertalen waartoe ook FORTRAN, Pascal of COBOL behoren. Deze talen worden aangemerkt als op problemen georiënteerde talen, omdat ze zich richten op mathematische problemen (FORTRAN) of commerciële toepassingen (COBOL). Daartegenover staan de zogenaamde machinegeoriënteerde talen zoals FORTH, die zich richten op de hardware van de computer. Daartoe behoort als extreem geval natuurlijk ook de machinetaal van de processor zelf.

2. BASIC begrijpt u dus goed, anders zou u zich niet wagen aan het programmeren in machinetaal. Uw Commodore 64 werkt met machinetaal en begrijpt niets van BASIC. Hoe komt het dan, zult u terecht vragen, dat hij BASIC-instructies zo gewillig en snel uitvoert? Daarvoor zorgt de in het bedrijfssysteem (operating system) opgenomen BASIC-interpreter. Deze interpreter zorgt voor een kant en klare BASIC-tekst. Gaat het om de uitvoering van het door u geschreven programma, dan moet iedere instructie door uw Commodore 64 worden omgezet in machinetaalinstructies die de computer begrijpt, zodat ze leiden tot het (BASIC) doel. Dat hoeft u eigenlijk verder niet te storen, want uiteindelijk voert de interpreter zijn opgave toch zelf en voor u nauwelijks merkbaar uit. We bekijken door middel van een eenvoudig voorbeeld hoe de

BASIC-interpreter te werk gaat:

```
PRINT "HALLO"
```

Als u deze instructie invoert en op de RETURN-toets drukt, zal deze instructie karakter voor karakter worden gelezen door de interpreter. Zodra het eerste woord gelezen is, wordt dit vergeleken met alle trefwoorden die in het geheugen voorkomen en als instructies dienst doen (FOR, INPUT, IF, NEXT, enzovoort). Als het woord gevonden is, wordt bepaald op welke plaats in de instructietabel het woord staat. Deze positie is nodig om te kunnen springen in de interpreter naar de positie van het PRINT-commando. Dan kan het eigenlijke printcommando uitgevoerd worden. Ook nu wordt weer karakter voor karakter gelezen. Aan het eerste aanhalingsteken merkt de interpreter dat u een tekst wilt afdrukken. Dit gaat karakter na karakter totdat het tweede aanhalingsteken wordt ingelezen. Daaraan herkent de computer het einde van de tekstregel. Dan wordt onderzocht of er nog meer tekst komt. Is dat niet het geval dan is de opdracht uitgevoerd en de interpreter meldt zich weer met READY. Die hele gang van zaken is tamelijk omslachtig: in machinetaal kan dat veel sneller.

1.1 Machinetaal is aanzienlijk sneller

Omdat uw computer BASIC begrijpt moet hij een interpreter hebben die de BASIC-instructies herkent en uitvoert. Deze interpreter is zelf in machinetaal geschreven. Om bijvoorbeeld de instructie POKE 1024, 10 uit te voeren moet eerst de instructie door de interpreter worden herkend. Daarna worden de getallen 1024 en 10 opgehaald zodat in geheugenplaats 1024 het getal 10 gezet kan worden. Dit proces duurt ongeveer 2 milliseconden (0.002 s): niet echt supertraag, zou je zeggen. Hoe ziet het er in machinetaal uit? Twee machinetaalcommando's staan u hiertoe ter beschikking:

```
LDA# 10
STA 1024
```

Deze beide instructies duren 6 microseconden (0.000006 s), meer dan driehonderd keer zo snel! U kunt ervan uitgaan dat een zuiver machinetaalprogramma ca. 10 tot 1000 keer zo snel is als een BASIC-programma voor dezelfde opgave. Bijzonder tijdrovende opgaven zijn bijvoorbeeld omvangrijke wiskundige berekeningen en vooral ook het sorteren van gegevens in bestanden. Bij omvangrijke gegevensbanken kan het zoeken en sorteren in BASIC gemakkelijk uren duren. In die situaties kunt u niet zonder

machinetaal.

Het gaat er niet alleen om dat machinetaal sneller is. Er zijn ook problemen die principieel niet zonder machinetaal op te lossen zijn. Daartoe behoren bijvoorbeeld het programmeren van de in- en uitvoeronderdelen voor de overdracht van informatie of routines die met interrupt uitgevoerd worden, een techniek, die in BASIC niet beschikbaar is. Interrupt is een onderbreking die betekent dat onderdelen van de computer of randapparaten het werk van de computer op een bepaald moment kunnen onderbreken en de computer dwingen dit apparaat passend te bedienen. In het algemeen geldt dat een al dan niet professionele programmeur alle mogelijkheden van de computer alleen maar met machinetaal ten volle kan benutten. Voor de Commodore geldt dat speciaal voor het grafisch gebruik met hoge resolutie en voor de synthesizer. Het geldt ook voor alle zaken die in de reële tijd moeten worden geprogrammeeerd. Een ander belangrijk punt is het gebruik van de beschikbare geheugenruimte. Een goed geschreven machinetaalprogramma neemt vaak tienmaal minder geheugenruimte in beslag dan het voor hetzelfde doel geschreven BASIC-programma. Een BASIC-programma van 1K is niet bijzonder groot, maar voor een machinetaalprogramma is dat een heel aanzienlijke omvang. Dat geldt ook voor de opslag van data. Alleen in machinetaal kunt u een compacte opslag van data bereiken. Ieder element van een tabel neemt in het gunstigste geval twee bytes in beslag als u gebruik maakt van velden voor gehele getallen, of als u alleen getallen tussen nul en honderd moet opslaan. In machinetaal is het geen probleem slechts een byte per element te gebruiken. U bespaart de helft aan geheugenruimte. In machinetaal kunt u de voor ieder probleem optimaal aangepaste datastructuur kiezen. Maar nadelen kent machinetaal ook.

Allereerst moet u het programmeren in machinetaal leren. Met BASIC beheerst u de basisprincipes al. Bovendien gebruiken we geschikte hulpmiddelen waarmee het maken van machinetaalprogramma's net zo gemakkelijk kan zijn als u van BASIC gewend bent. In dit boek vindt u een assembler die u voor het maken van machinetaalprogramma's kunt gebruiken. Een nadeel van machinetaalprogramma's is dat zij principieel alleen maar op machines lopen die hetzelfde processortype hebben. Ook op verschillende computers met dezelfde processor zijn aanpassingen vereist. Daar moet u bij het programmeren al rekening mee houden. Verder is het testen van machinetaalprogramma's zonder hulpmiddelen niet zo gemakkelijk als van programma's in BASIC. Daarvoor stellen we u een simulatieprogramma ter beschikking, dat u erbij helpt, fouten in uw programma te vinden.

Samenvattend kunnen we stellen dat het gebruik van machinetaal volstrekt gerechtvaardigd is. Veel problemen zijn zonder machinetaal onoplosbaar en alleen daarmee kan de computer ten volle benut worden. Soms zult u uw hoofdprogramma in BASIC schrijven en enkele onderdelen in machinetaal formuleren. Deze manier van werken is zeer populair en u komt ze vaak tegen in de vorm van programmeerhulpmiddelen (utility's) of BASIC-uitbreidingen. Deze utility's zijn in machinetaal geprogrammeerd zodat u met BASIC gemakkelijker kunt werken.

Hebt u voor het eerst een machinetaalprogramma geschreven, dan zult u vaststellen dat het helemaal niet zo moeilijk is. Wij hopen dat de inhoud van dit boek u daarbij kan helpen en u inspireert tot het oplossen van uw eigen machinetaalproblemen.

2 DE 6510 MICROPROCESSOR

Voordat we ons met het programmeren in machinetaal bezighouden, moeten we eerst de processor zelf beter leren kennen, want programmeren heeft juist daarop betrekking. Tevens proberen we de basisbegrippen zoals registers, data, adressen en ook bytes en bits uit te leggen. We beginnen met de opbouw van de processor. De microprocessor 6510 behoort tot de familie van de 65xx-processoren die veel voorkomen en die bijvoorbeeld in alle Commodore computers zijn te vinden. De processor heeft intern een aantal registers, waarin alle bewerkingen plaatsvinden. Een microprocessor werkt digitaal, hij kan slechts twee toestanden onderscheiden, die u zich kunt voorstellen als aan-uit of als 1-0. Zo'n schakelaar kan slechts twee verschillende standen innemen. Daarmee kun je niet veel beginnen. Daarom worden acht schakelaars samen in een register ondergebracht. Een schakelaar wordt gekenmerkt als een bit, terwijl de acht schakelaars van het register samen een byte worden genoemd. Een 8-bits register kan als volgt voorgesteld worden:

```
7 6 5 4 3 2 1 0
0 1 1 0 1 0 0 1
```

In de bovenste rij ziet u de nummering van de acht enkele bits, die van rechts naar links de nummers 0 t/m 7 krijgen. Daaronder staat de inhoud van het betreffende bit, een 0 of een 1. Hoewel een bit twee toestanden kan innemen kunnen we met een byte van 8 bits veel meer voorstellen. Dit gaan we nauwkeuriger bekijken en we vergelijken dat met normale decimale getallen.

```
nummering:       3 2 1 0
decimaal getal: 5 7 2 4
```

De plaatsen van de vier getallen zijn weer genummerd, nu van 0 t/m 3. Wat is nu de waarde van dit decimale getal 5724? Elk cijfer heeft een waarde tussen nul en negen en de volgende plaats heeft telkens de tienvoudige waarde:

$$4 + 2*10 + 7*10*10 + 5*10*10*10 = 4 + 20 + 700 + 5000 = 5724$$

Op dezelfde wijze kunt u met de 8-bits registerinhoud te werk gaan. In tegenstelling tot het decimale (tientallige) stelsel noemt men dit het binaire (tweetallige) stelsel, omdat ieder cijfer niet tien waarden maar slechts twee waarden kan aannemen. Vanzelfsprekend heeft de eerstvolgende positie niet de tienvoudige, maar de tweevoudige waarde. Nu kunnen we ook de waarde van ons binaire getal 01101001 berekenen. Wij gaan van rechts naar links:

1*2^0 + 0*2^1 + 0*2^2 + 1*2^3 + 0*2^4 + 1*2^5 + 1*2^6 +0*2^7 =

1 + 0 + 0 + 8 + 0 + 32 + 64 + 0 = 105

De decimale waarde van de registerinhoud is 105. De 8 bitposities kunnen dus de volgende waarden bevatten:

bitpositie	waarde
0	2^0 = 0
1	2^1 = 2
2	2^2 = 4
3	2^3 = 8
4	2^4 = 16
5	2^5 = 32
6	2^6 = 64
7	2^7 = 128

Voorbeeld:

```
Binair getal  : 1     0   1    0   1   1   1   0
Decimaal getal: 128 + 0 + 32 + 0 + 8 + 4 + 2 + 0 = 174

Maximaal getal: 1     1   1    1   1   1   1   1
Decimaal getal: 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
```

In totaal kunnen in een 8-bits register de getallen 0 t/m 255 worden geplaatst. In totaal zijn dit 256=2^8 verschillende waarden. Dit bereik kent u misschien al van de PEEK-functie, maar daar komen we later op terug. Het gebruik van veel nullen en enen is nogal omslachtig, daarom heeft men een tweede getallensysteem ingevoerd dat in nauw verband staat met het binaire systeem, maar veel minder posities bij de presentatie in beslag neemt. Verdeelt men namelijk een 8-bits binair getal in twee 4-bits binaire getallen, dan kan ieder 4-bits getal 16 verschillende waarden aannemen (0 t/m 15). Nu is nog een getallensysteem met 16 cijfers nodig om een 8-bits binair getal door twee cijfers van dit 16-tallig systeem uit te drukken.

```
bit        : 7 6 5 4  3 2 1 0
2-tallig   : 0 1 1 0  1 0 0 1

16-tallig :   6        9
```

Iedere byte wordt in twee halve bytes, ook nibbles genoemd, verdeeld. Omdat deze nibbles de waarden 0 t/m 15 kunnen aannemen en in het decimale systeem slechts de cijfers 0 t/m 9 beschikbaar zijn, moet voor de getallen 10, 11, 12, 13, 14 en 15 iets anders

worden gekozen. Hiervoor heeft men de letters A, B, C, D, E en F gekozen. Dit 16-tallige systeem noemen we het hexadecimale systeem. De vogende tabel ontstaat:

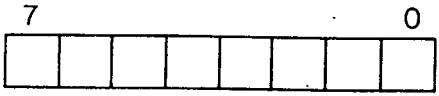
binair	hexadecimaal	decimaal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Onze registerinhoud heeft dus de hexadecimale waarde 69. Om de verschillende getallensystemen te onderscheiden wordt het hexadecimale getal voorafgegaan door een dollarteken ($) en het binaire getal door een procentteken (%). In het voorbeeld is %01101001 = $69 = 105 (decimaal). Achter in dit boek vindt u de omrekeningstabellen voor de getallen 0 t/m 255 decimaal in binaire en hexadecimale getallen. We zullen het meest met hexadecimale getallen werken, omdat deze enerzijds gemakkelijk voor te stellen zijn en anderzijds gemakkelijk in binaire getallen zijn om te zetten. In het hexadecimale systeem kunnen de 8-bits waarden door slechts twee cijfers worden voorgesteld. Na dit korte uitstapje in de getallensystemen zullen we ons nu verder met de processor bezighouden. De 6 registers van de processor:

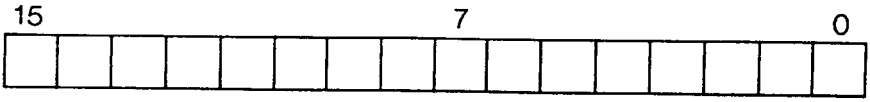
Accumulator

X-Indexregister

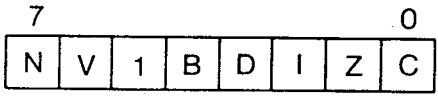
Y-Indexregister



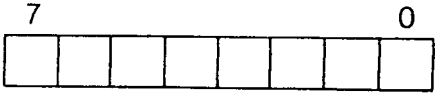
Instructieteller(Program Counter)



Statusregister



Stapelwijzer (Stack Pointer)



Samen zijn er 6 registers, 5 daarvan zijn 8-bits registers en de zesde is een 16-bits register.

Accumulator

Het eerste is de accumulator (A-register), het belangrijkste register van de processor. Het is het werkregister van de processor, waarmee alle bewerkingen plaatsvinden. Dit betekent dat alle rekenkundige en logische bewerkingen en bijna alle vegelijkingsinstructies via de accumulator lopen. De veschillende adresseermogelijkheden van de instructies zijn alleen via de accumulator uitvoerbaar.

X-register

Het X-register is het tweede register van de processor. Bij de verwerking van tabellen wordt dit register in samenspel met de accumulator benut als teller van of als wijzer naar de tabelonderdelen. Daarom wordt dit register ook X-indexregister genoemd.

Y-register

Het Y-register is net als het X-register een indexregister en kan op dezelfde manier worden gebruikt.

Stackpointer

De stapelwijzer, beter bekend als stackpointer, geeft een positie aan in een stapel geheugenplaatsen. De Engelse naam van zo'n stapel is stack. De stack wordt gebruikt voor subroutines of voor kortstondige en snelle opslag van data. De betekenis van de stack en de stackpointer leren we kennen bij het programmeren van subroutines.

Statusregister

Het statusregister tenslotte geeft inlichtingen over het resultaat van de laatste instructie en is de basis voor beslissingen en voorwaardelijke instructies. Van de 8 bits worden 7 gebruikt als zogenaamde vlaggen (Engels: flags). Staat de vlag uit (gezet) dan is de inhoud een 1, is de vlag gestreken (niet gezet) dan is de inhoud een 0. Er zijn bijvoorbeeld spronginstructies die alleen dan uitgevoerd worden als een bepaalde vlag gezet of niet gezet is. Deze vlaggen kunnen waar (=1) of niet waar (=0) zijn en signaleren of aan een bepaalde voorwaarde is voldaan. Het statusregister is als volgt opgebouwd:

```
7 6 5 4 3 2 1 0
N V + B D I Z C
```

De letters zijn afkortingen voor de namen van de vlaggen en hebben de volgende betekenis:

C + Carry

De carryflag laat zien of bij een bewerking een overloop optreedt. Als bijvoorbeeld bij het optellen van de twee getallen het resultaat groter is dan 255 en dit resultaat niet meer in een byte van 8-bits is te zetten, wordt de carryflag gezet (=1).

Z + Zero

De zeroflag wort gezet (=1) als het resultaat van een bewerking nul is.

I + Interrupt Disable

Deze vlag beslist of interrupts (onderbrekingen) in het lopende programma toegestaan zijn. Is de vlag gezet (=1) dan wordt het programma niet onderbroken. Deze vlag interesseert ons op dit moment nog niet.

D + Decimaal

De decimaalvlag bepaalt of optelling of aftrekking op de decimale wijze plaatsvindt. Ook van deze vlag maken we voorlopig geen gebruik.

B + Break

De breakflag wijst op een onderbreking door het BRK-commando en heeft voor ons voorlopig nog geen betekenis.

V + Overflow

De V-flag wordt alleen maar gebruikt als gewerkt wordt met positieve en negatieve getallen. De V-flag(=1) toont dan een overloop aan.

N + Negative

Deze vlag wordt gezet als het resultaat van een bewerking een waarde groter dan 127 krijgt, zodat het zevende bit gezet (=1) wordt. De betekenis negative is een gevolg van het feit dat waarden van getallen groter dan 127=$7F als negatief zijn te interpreteren.

Voorlopig zullen we ons alleen om de carry-, de zero- en de negative-flag bekommeren. Om met een processor te kunnen werken moet hij een programma krijgen dat hij kan uitvoeren. Eveneens moeten data die hij moet verwerken ergens vandaan komen en ook weer worden weggezet. Daarvoor wordt het werkgeheugen van de computer gebruikt. Dit geheugen wordt in 8-bits cellen onderverdeeld, precies even groot als de accumulator en het X-en Y-indexregister. Opdat de processor met dit werkgeheugen kan communiceren moet hij de mogelijkheid hebben om een bepaalde geheugenplaats uit te kiezen. Deze keuze noemt men de adressering

van het geheugen. Iedere geheugenplaats (byte) geeft men een nummer, een adres. Het adres wordt door de processor gezien als een binair getal. Zou dit adres voorgesteld worden door een 8-bits binair getal, dan zouden slechts de adressen 0 t/m 255 bereikt kunnen worden. Voor de meeste toepassingen is dat veel te weinig en daarom heeft men de adresnummers van 16-bits voorzien. Dan is het mogelijk om 2^16=65536 geheugenplaatsen te adresseren. We spreken van een processor met een 16-bits adresbus en een 8-bits databus. De 6510-processor kan 65536 verschillende adressen bereiken, waarvan elk adres (geheugenplaats of byte) een waarde van 0 t/m 255 kan bevatten. Voor het gemak rekent men 2^10=1024 bytes als 1 kilobyte (1K). De processor kan dus 64*1024=65536 bytes adresseren. Bij de Commodore 64 is het gehele bereik benut. Nu kunnen we ook de betekenis van de instructieteller of program counter begrijpen. Die bevat een 16-bits waarde. Deze waarde is het adres van de eerstvolgende instructie die de microprocessor uit het geheugen haalt en vervolgens uitvoert. Een instructie voor de microprocessor kan zo door een getal van 0 t/m 255 worden voorgesteld. Het totale aantal mogelijke instructies is dus 256. Hiermee worden natuurlijk geen BASIC-instructies bedoeld, maar door de processor direkt te begrijpen machinetaalinstructies.

Programcounter

De instructieteller of programcounter is een 16-bits register. De inhoud bepaalt uit welke geheugenplaats (adres) de volgende instructie gehaald wordt. Dit register wordt door de processor zelf bestuurd. Normaal hoeft u zich daarmee niet direkt te bemoeien.

3 INSTRUCTIES EN ADRESSEERMOGELIJKHEDEN VAN DE 6510

Van de 256 verschillende instructiemogelijkheden worden er in de praktijk 151 gebruikt. Veel instructies zijn gelijk, maar onderscheiden zich door de verschillende wijze van adresseren. In totaal zijn er 56 verschillende instructies. Deze instructieset is gemakkelijk te leren. We bespreken ze hieronder kort.

3.1 Laadinstructies

Deze instructies worden gebruikt om data uit het geheugen te halen en weg te zetten in een van de registers van de processor. Omdat de processor drie werkregisters heeft, bestaan er drie verschillende laadinstructies:

LDA laad de accumulator
LDX laad X-indexregister
LDY laad Y-indexregister

De adresseerwijze geeft aan waar de inhoud vandaan komt. De inhoud van het register kan rechtstreeks geplaatst worden, maar ook kan de inhoud van het register gelijk gemaakt worden met de inhoud van een bepaalde geheugenplaats (adres). Ter vergelijking noemen we steeds de BASIC-instructie met dezelfde functie.

Immediate, onmiddellijke adressering

Met LDA #10 wordt het decimale getal 10 onmiddellijk in de accumulator gezet, zonder gebruik te maken van een van de andere adresinhouden. Met deze instructie komt de BASIC-instructie: LET A=10 overeen. Deze instructie wordt gebruikt als een constante die in een register moet worden gezet. Ook bij het X- en Y-register kennen we deze adresseerwijze. Omdat de inhoud van het register uiteraard 8 enen en/of nullen is, wordt gewerkt met de hexadecimale schrijfwijze. Voor het hexadecimale getal wordt het $-teken geplaatst. LDX #$7F betekent: laad onmiddellijk het hexadicimale getal $7F=%0111 1111=127 in het X-indexregister. LDY #$AB betekent: laad onmiddellijk $AB=%1010 1011=171 in register Y. Bij deze manier van adresseren is de waarde die geladen wordt onderdeel van het programma, net als bij de overeenkomende BASIC-instructie. In het geheugen staan de instructie en de te laden waarde in twee opeenvolgende geheugenplaatsen. Als een

machinetaalprogramma op dit adres wordt gestart, haalt de processor de inhoud op van dit adres; deze inhoud wordt geïnterpreteerd als een instructie. De instructiecodes voor de drie instructies zijn:

```
LDA imm =$A9
LDX imm =$A2
LDY imm =$A0
```

Als op het startadres van het programma $A9 staat, weet de processor dat de inhoud van het eerstvolgende adres onmiddellijk in de accumulator moet worden geplaatst. De instructiecode (ook wel Operand-code of Op-code) neemt 1 byte in beslag. Het getal waarmee de instructie wordt uitgevoerd (operand) neemt ook 1 byte in beslag. De gehele instuctie, code en waarde, neemt dus 2 bytes in beslag. De instructieteller (program counter) wordt met 2 opgehoogd en de instructie die op dat adres staat wordt door de processor opgehaald en vervolgens weer uitgevoerd.

Absolute, de absolute adressering

Deze adresseerwijze wordt gebruikt om het register niet met een vaste waarde, maar met de inhoud van een bepaalde geheugenplaats te laden.

LDA $COAF betekent: laad de inhoud van adres $COAF = %11000000 10101111=49327 decimaal in de accumulator. Hoe kan de accumulator met de inhoud van adres $COAF geladen worden? Het adres $COAF is een 16-bits getal en een geheugenplaats kan slechts 8 bits bevatten. Men verdeelt eenvoudig het 16-bits getal in twee stukken van elk 8-bits getallen. De volgende procedure wordt daarbij gevolgd: na de instructie komt eerst het laagste gedeelte van het adres (low byte) en daarna het hoogste deel van het adres (high byte). De instructiecodes voor de absolute adressering zijn:

```
LDA abs=$AD
LDX abs=$AE
LDY abs=$AC
```

Als met de instructie LDA $COAF gestart wordt, moet in de eerste byte $AD (dec. 173) gezet worden, in de volgende byte $AF (dec. 175) en tenslotte $C0 (dec. 192) in de derde byte. De overeenkomstige BASIC-instructie is:

```
A=PEEK(49327), (49327=$COAF)
```

Evenzo kan met het X- en Y-register gewerkt worden. Bij de uitvoering van de instructie haalt de processor eerst de

instructiecode op en weet dan dat het om de absolute adressering gaat, vervolgens wordt lo-byte gehaald en daarna de hi-byte. Dan wordt de inhoud van dit adres gehaald en in de accumulator geladen. Deze instructie is een 3-bytes instructie, in tegenstelling tot de immadiate adressering waar het twee bytes betreft. De processor kent ook nog 1-byte instructies, die alleen betrekking hebben op het interne gebeuren in de processor: er is dan geen operand nodig. We keren nu weer terug naar het statusregister, dat bij de 6510 een belangrijke rol speelt en door bijna alle instructies wordt beïnvloed. Op dit moment beperken we ons tot de carry-, de zero- en de negativeflag. Bij iedere laadinstructie worden de zero- en de negativeflag beïnvloed. De zeroflag wordt gezet als de geladen waarde nul is en wordt teruggezet als de waarde ongelijk is aan nul. De negativeflag wordt gezet als de waarde van de zevende bit 1 is, dus als de waarde groter is dan $7F (dec. 127). Bij kleinere getallen wordt de waarde van het N-bit nul. Een andere manier van adresseren is de zeropage-adressering. Deze adresseringswijze kan worden gebruikt als het adres een nummer van 0 t/m $FF (dec. 0 t/m 255) is. Dit adres neemt 8 bits in beslag. Bij de absolute adressering zijn dan slechts 2 bytes nodig in plaats van 3, 1 byte voor de instructiecode (OP-code) en 1 byte voor het adres. De zeropage-adressering neemt dus minder geheugenplaats in beslag en wordt bovendien door de processor sneller uitgevoerd. Nadeel is natuurlijk dat er een beperkt aantal adressen bereikbaar is. De aanduiding zeropage komt van de verdeling van het geheugen in pagina's van elk 256 bytes. Vier bladzijden van elk 256 bytes bevatten samen 1K=1024 bytes. De 64K van de Commodore kunnen voorgesteld worden door 256 pagina's (0 t/m 255) van elk 256 bytes (0 t/m 255) De voordelen van de zeropage-adressering leiden er toe dat de zeropage bij een 65xx systeem een bijzondere betekenis heeft. De meeste systeemvariabelen van het bedrijfssysteem (operating system) en BASIC-interpreter worden hier opgeslagen. De laadinstructiecodes voor adressering van de zeropage zijn:

```
LDA abs zero page =$A5 (decimaal 165)
LDX abs zero page =$A6 (decimaal 166)
LDY abs zero page =$A4 (decimaal 164)
```

Als de accumulator geladen moet worden met de inhoud van adres $73 (dec. 115), is de instructie:

```
LDA $73
```

Het 2 bytes programma voor deze instructie is: $A5 $73 = %10100101 %01110011 = 165 115. De BASIC-instructie is:

```
A=PEEK(115)
```

Indexed, de absolute geïndexeerde adressering

Bij de geïndexeerde adressering spelen de indexregisters X en Y een belangrijke rol. Deze wijze van adresseren wordt gebruikt bij het programmeren van lussen.

```
LDA $25B8, X
```

De processor laadt nu de accumulator niet met de inhoud van adres $25B8 (dec. 9656), maar telt de inhoud van register X erbij op om het adres te vinden waar de te laden waarde staat. Stel, dat de inhoud van het X-indexregister $35 is, dan wordt de volgende berekening uitgevoerd: $25B8+$0035=$25ED decimaal: 9656+53=9709. De accumulator wordt nu geladen met de inhoud van adres $25ED (dec. 9709). Als dit programmaonderdeel meerdere keren wordt doorlopen met verschillende X-waarden, wordt elke keer een andere waarde geladen. Deze wijze van adresseren is zeer nuttig voor het programmeren van lussen en voor de bewerking van tabellen. Er volgen later meer voorbeelden. De BASIC-instructie luidt:

```
A=PEEK($25B8+X)=PEEK($25B8+$35)=PEEK($25ED)=PEEK(9656+53)
=PEEK(9709)
```

Hierin is X de inhoud van het X-indexregister. In plaats van het X-register kan ook het Y-register als indexregister gebruikt worden, bijvoorbeeld:

```
LDA $25B8, Y
```

Nu wordt de inhoud van het Y-register bij adres $25B8 opgeteld om het adres te vinden van waaruit de inhoud in de accumulator moet worden gezet. Zo krijgt u twee van elkaar onafhankelijke lusvariabelen, bijvoorbeeld voor de programmering van lussen binnen lussen. Deze geïndexeerde adressering wordt ook toegepast voor de zeropage-adressen. We beschikken over

```
zeropage indexed, X
```

en

```
zeropage indexed, Y
```

Natuurlijk hebben we dan weer te maken met een 2-bytes-instructie.

Indirect Indexed, de indirekte geïndexeerde adressering met het Y-register

Deze adresseringswijze is niet zo gemakkelijk te begrijpen, maar maakt echter een flexibele programmering mogelijk. De geïndexeerde adressering kennen we al, hier wordt extra indirekt geadresseerd. Bij deze manier van adresseren speelt de zeropage

weer een grote rol. Bij de indirekte geïndexeerde adressering vormen twee op elkaar volgende geheugenplaatsen een wijzer naar het adres waar de inhoud vandaan gehaald moet worden. De inhoud van het eerste adres geeft de lo-byte en de inhoud van het opvolgende adres de hi-byte aan van het eigenlijke adres. Een voorbeeld zal dit duidelijk maken. We nemen aan dat het zeropage-adres $70 de waarde $20 bevat, het adres $71 bevat de waarde $C8. Deze beide geheugenplaatsen vormen zo een wijzer naar adres $C820. Vervolgens komt hier nog de indexering van het Y-register bij. Stel dat de inhoud van het Y-register $B3 is, dan wordt deze waarde opgeteld bij $C820. Het bewuste adres is dan $C820+$00B3=$C8D3 van waaruit de inhoud gehaald wordt. De vorm van de instructie is:

```
LDA ($70), Y   inhoud adres $70=($70)=$20
               inhoud adres $71=($71)=$C8
               wijst naar adres $C820
               inhoud Y-register=(Y) =$ B8
               adres =$C8D3
               inhoud $C8D3 =($C8D3) =$ 4F
```

De accumulator wordt geladen met $4F. In BASIC ziet dat er zo uit:

```
A=PEEK(PEEK(70)+256*PEEK($71)+Y)
```

De indirekte adressering wordt gekenmerkt door de operand tussen haakjes te zetten om daarmee aan te geven dat het om de inhoud van het betrokken adres gaat. Deze adresseringswijze is zeer krachtig, want met een 2-byte-instructie kan het gehele geheugen worden bereikt. Deze manier geniet de voorkeur bij het werken met tabellen en lussen. Zij is flexibeler dan de eenvoudige geïndexeerde adressering, omdat hier alle adressen mee worden bereikt. Alleen de inhoud van de 2-byte-wijzer in de zeropage hoeft te worden veranderd.

Indexed Indirect, de geïndexeerde, indirekte adressering met het X-register.

In tegenstelling tot de indirekte geïndexeerde adresseerwijze wordt niet met het Y-register, maar met het X-register gewerkt. Ook hier vormt weer de inhoud van twee opvolgende adressen op de zeropage een wijzer naar het eigenlijke adres. Bij de berekening wordt eerst de inhoud van het X-register opgeteld bij dit zeropage adres en vervolgens wordt de inhoud van het zo verkregen adres als wijzer gebruikt. Bijvoorbeeld:

```
LDA($70, X)    inhoud X-register =(X)=$08
               wijzer naar adres $70+$08=$78
               inhoud adres $78 =($78)=$40
```

inhoud adres $79 =($79)=$20
wijst naar adres $2040
inhoud $2040 =($2040)=$A9

De accumulator wordt geladen met $A9

In BASIC ziet dit er als volgt uit:

```
A=PEEK(PEEK($70+X)+256*PEEK($70+X+1))
```

Eerst wordt dus bij de operand de inhoud van het X-register opgeteld om zo de twee adressen te vinden van waaruit gewezen wordt naar het eigenlijke adres. Deze adresseringswijze wordt in tegenstelling tot de boven beschreven indirekte geïndexeerde wijze nauwelijks toegepast. U hoeft hierop dus voorlopig geen acht te slaan. Het overzicht van de adresseermogelijkheden en de bijbehorende instructiecodes (Op-codes) is:

Wijze van adresseren:	LDA	LDX	LDY
immediate	$A9	$A2	$A0
absolute	$AD	$AE	$AC
zeropage	$A5	$A6	$A4
absolute X-indexed	$BD	-	$BC
absolute Y-indexed	$B9	$BE	-
zeropage X-indexed	$B5	-	$B4
zeropage Y-indexed	-	$B6	-
indirect indexed (met Y)	$B1	-	-
indexed indirect (met X)	$A1	-	-

De relatieve adressering en de accumulator-adressering zullen we bij de bijbehorende instructies behandelen.

3.2 Opslaginstructies

Tegenover de laadinstructies staan de opslaginstructies. De inhoud van de accumulator, het X-en Y-indexregister kunnen ergens in het geheugen worden weggezet. Deze instructies zijn:

STA
STX
STY

De inhoud van het register wordt op de door de operand aangegeven plaats weggezet. De adresseermogelijkheden zijn dezelfde als bij de laadinstructies, uitgezonderd natuurlijk de immediate adressering, omdat bij het wegzetten steeds een adres moet worden opgegeven waarin de registerinhoud gestopt moet worden. Bij het wegzetten van de registerinhouden verandert aan de inhoud van het register zelf niets, er is alleen sprake van een kopie. Daarom worden de vlaggen van het statusregister niet beïnvloed door deze instructies. De instructiecodes en adresseermogelijkheden zijn:

wijze van adresseren	STA	STX	STY
absolute	$8D	$8E	$8C
zeropage	$85	$86	$84
absolute X-indexed	$9D	-	-
absolute Y-indexed	$99	-	-
zeropage X-indexed	$95	-	$94
zeropage Y-indexed	-	$96	-
indirect-indexed	$91	-	-
indexed-indirect	$81	-	-

De hiermee overeenstemmende BASIC-instructies kent u zeker: het is het POKE-statement, dat de waarde van een variabele in een bepaalde geheugenplaats zet.

```
STA $8000       POKE $8000, A
STX $C020, Y    POKE $C020+Y, X
STY $F1         POKE $F1, Y
```

De opslaginstructies zijn weer volgens de adresseerwijze 2- of 3-byte instructies. Met de laad-en opslaginstructies hebben we twee belangrijke groepen instructies leren kennen, die de communicatie bewerkstelligen tussen de processor en het aangesloten geheugen.

3.3 Transferinstructies binnen de processor

De 6510 microprocessor heeft instructies tot zijn beschikking die de inhoud van een register in een van de andere registers kopiëren. Daarmee kunt u de inhoud van bijvoorbeeld de accumulator rechtstreeks kopiëren in het X-register en omgekeerd. Dit is bijzonder belangrijk, omdat veel instructies alleen werken met de accumulator. De inhoud van het oorspronkelijke register

blijft bij het kopiëren intact, de waarde wordt alleen in het doelregister gekopieerd. Bij de transferinstructies is op een enkele uitzondering na de accumulator betrokken, een rechtstreeks kopiëren van het X-register in het Y-register of omgekeerd is niet mogelijk. Alle transferinstructies zijn 1-byte instructies, zij hebben geen operand nodig. Door de Op-code is de bewerking eenduidig vastgelegd. We bekijken nu de afzonderlijke instructies. Voor een beter begrip worden weer de overeenkomstige BASIC-instructies ernaast gezet.

TAX X = A

De inhoud van de accumulator wordt in het X-register gekopieerd. De Z-flag en de N-flag worden beïnvloed. De oorspronkelijke inhoud van de accumulator blijft behouden.

TXA A = X

Dit is de omgekeerde instructie van TAX. De inhoud van het X-register wordt in de accumulator gezet, de Z-flag en de N-flag worden beïnvloed. De inhoud van het X-register verandert niet. Evenzo kennen we:

TAY Y=A

TYA A=Y

Deze instructies zijn identiek aan de bovenstaande, alleen betreft het nu het Y-register. De volgende twee transferinstructies betreffen de stackpointer (stapelwijzer). Deze vermelden we hier voor de volledigheid, later behandelen we de betekenis van de stackpointer. De inhoud van de stackpointer kan alleen met het X-register gewisseld worden.

TSX X=SP

De inhoud van de stackpointer wordt in het X-register gezet, overeenkomstig de waarde worden Z-en N-flag gezet of niet gezet. De inhoud van de stackpointer blijft natuurlijk onveranderd.

TXS SP=X

Deze instructie bewerkstelligt het omgekeerde transport. De inhoud van het X-register wordt in de stackpointer gebracht. Bij deze instructie worden geen flags beïnvloed, omdat de stackpointer geen normaal werkregister van de processor is. De inhoud van het X-register blijft natuurlijk behouden. Het overzicht van de transferinstructies en hun Op-code is:

instructie	instructiecode
TAX	$AA
TXA	$8A
TAY	$A8
TYA	$98
TSX	$BA
TXS	$9A

3.4 De rekenkundige instructies

De processor moet aan de slag: hij gaat rekenen. De 6510 kan optellen en aftrekken. Bij de rekenkundige bewerkingen wordt als volgt te werk gegaan: Iedere rekenkundige bewerking heeft twee getallen nodig (operands) die met elkaar worden verbonden en een resultaat opleveren. Het eerste getal moet bij de 6510 in de accumulator staan terwijl het tweede getal uit het geheugen moet worden gehaald. Daarvoor staan weer de verschillende adresseermanieren ter beschikking. Het resultaat van de bewerking wordt steeds in de accumulator gezet. Dit principe geldt ook voor de later te bespreken logische bewerkingen. De vergelijking met de overeenkomstige BASIC-instructies zal dit verduidelijken. Allereerst bekijken we de optelling. Het resultaat van de optelling van de inhoud van een geheugenplaats met die van de accumulator wordt weer in de accumulator gezet.

```
ADC #$3A A=A+$3A
```

Als twee getallen die elk 8 bits in beslag nemen, worden opgeteld zal het vaak voorkomen dat het resultaat groter is dan 255 en dus meer dan 8 bits in beslag neemt. Er is dan sprake van een overloop. De binaire optelling bekijken we aan de hand van twee voorbeelden:

ADC #3A, de accumulator bevat bijvoorbeeld $9E

```
$9E =   %10011110
$3A =   %00111010
```

De optelling ziet er dan zo uit:

```
      10011110
+     00111010
-------------
      11011000 = $D8
```

De binaire optelling verloopt analoog aan de decimale optelling. Er zijn vier verschillende gevallen:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 en 1 onthouden !
```

Het bekende 1 onthouden is het overschot en wordt net als bij de decimale optelling bij de volgende optelling meegeteld. In ons voorbeeld is het resultaat %11011000 of $D8. Het resultaat kan nog in een byte van 8 bits gezet worden. Bij het volgende voorbeeld is dat niet zo:

ADC #$3A, de accumulator bevat nu $E4

```
$E4 = %11100100
$3A = %00111010
```

De optelling ziet er dan zo uit:

```
      11100100
+     00111010
-------------
     100011110 = $11E
```

In dit geval is er een overloop van de achtste bit naar de aanwezige negende bit. Voor dit geval wordt de carrybit uit het statusregister gebruikt. De vlag wordt gezet (bit =1) als er overloop is, en niet gezet (bit=0) als dat niet het geval is. De carryflag wordt daarom ook wel de negende bit van de accumulator genoemd. Als getallen die meer dan 8 bits in beslag nemen opgeteld moeten worden, kunt u de meervoudige optelling gebruiken. Met twee bytes kun je een getal van 0 t/m 65535 voorstellen, een bereik dat voor de meeste doeleinden voldoende is. Om twee van deze getallen op te tellen, telt u na elkaar eerst de onderste 8 bits op en daarna de bovenste 8 bits, precies zoals de processor intern met de 8 bits na elkaar doet. Treedt een overschot van de eerste optelling op, dan moet dit bij de tweede optelling erbij worden genomen. Om dit te vereenvoudigen wordt bij iedere optelling de carryflag mee opgeteld. Voordat opgeteld wordt moet de carryflag op nul worden gezet omdat dan nog overloop heeft plaatsgevonden. Nu kunnen we ook de overeenkomstige BASIC-instructie geven:

```
ADC #3A A=A+$3A+C
```

Een eventueel overschot wordt na iedere optelling zichtbaar gemaakt met de carryflag. Naast de carryflag worden ook de negativeflag en de zeroflag gezet. De N-flag wordt gezet als de zevende bit 1 is en de Z-flag wordt gezet als het resultaat 0 is. Verder wordt ook de V-flag beïnvloed. Deze flag wordt alleen gebruikt als met positieve en negatieve getalllen wordt gewerkt. Daarom laten we die voorlopig buiten beschouwing. De volgende tabel geeft een overzicht van de ADC-instructie en de codes voor de verschillende adresseermogelijkheden:

Adresseringswijze	ADC
immediate	$69
absolute	$6D
zeropage	$65
absolute X-indexed	$7D
absolute Y-indexed	$79
zeropage X-indexed	$75
indirect indexed	$71
indexed indirect	$61

De aftrekking verloopt analoog aan de optelling. De inhoud van de geadresseerde byte wordt nu van de accumulatorinhoud afgetrokken. Bij de optelling kan een overschot voorkomen en bij de aftrekking een tekort. Een tekort wordt door de carryflag met een 1 aangegeven. Als geen tekort optreedt is de carrybit nul. In BASIC kunnen we dit als volgt formuleren:

```
SBC #3A A=A-$3A-(1-C)
```

De binaire aftrekking wordt analoog aan de optelling uitgevoerd. Daarbij kunnen weer vier gevallen worden onderscheiden:

```
0 - 0 = 0
0 - 1 = 1 plus een tekort
1 - 0 = 1
1 - 1 = 0
```

Als de inhoud van de accumulator bijvoorbeeld $7F is, is de binaire weergave:

```
$7F    %01111111
$3A    %00111010
```

Als de carryflag gezet is (geen tekort) ziet de aftrekking er als volgt uit:

```
   01111111
 - 00111010
------------
   01000101
```

Het resultaat is dus %01000101 of $45. Omdat geen tekort optreedt wordt de carryflag weer gezet. Bij het volgende voorbeeld ziet het er iets anders uit. De accumulator bevat nu bijvoorbeeld $1E, de carryflag is gezet.

```
$1E %00011110 30
$3A %00111010 58
```

De aftrekking is binair:

```
   00011110
 - 00111010
------------
   11100100
```

Als resultaat krijgen we %11100100 =$E4. Omdat een tekort is opgetreden is de carryflag op nul gezet. Dit resultaat moet nog geïnterpreteerd worden. Eerst nemen we de normale decimale aftrekking in beschouwing. Het resultaat van 30-58 is het negatieve getal -28. Als resultaat van de binaire aftrekking hebben we $E4 gekregen, dit is decimaal gelijk aan 228. Hebben deze getallen op een of andere manier met elkaar te maken ? Als het resultaat 228 van 256 wordt afgetrokken vinden we 256-228=28. De carrybit is nul, wat aangeeft dat het resultaat als negatief is te beschouwen. Men spreekt van het zogenaamde two's complement van een getal. Als we weten dat het een negatief getal betreft kan de waarde van het negatieve getal worden vastgesteld door alle bits om te keren en aan het resultaat 1 toe te voegen.

```
           11100100
dit wordt  00011011
1 optellen 00000001
```

wordt 00011100 of $1C gelijk aan 28

Bij de rekenkundige bewerkingen moet opgemerkt worden dat voor de optelling de carrybit op nul gezet moet worden en dat een overschot 1 -onthouden door een gezette carryflag- wordt gekenmerkt. Bij de aftrekking moet eerst de carryflag gezet worden, als een tekort optreedt (1 onthouden) is de carrybit nul In dit geval krijgt u een negatief getal in het two's complement.

De tabel voor de SBC-instructiecodes voor de verschillende adresseerwijzen is:

adresseerwijze	SBC
immediate	$E9
absolute	$ED
zeropage	$E5
absolute X-indexed	$FD
absolute Y-indexed	$F9
zeropage X-indexed	$F5
indirect indexed	$F1
indexed indirect	$E1

3.5 De logische instructies

De logische instructies verbinden twee waarden met elkaar; net als bij de rekenkundige instructies moet daarbij een van de operands in de accumulator staan, terwijl de andere overeenkomstig de wijze van adresseren uit het geheugen wordt gehaald. Het resultaat van de bewerking wordt weer in de accumulator gezet. De 6510 kent drie soorten van logische bewerkingen.

De AND-verbinding

Bij de AND-verbinding wordt iedere bit van de accumulator verbonden met de overeenkomstige bit van de operand. Alleen als beide bits een 1 bevatten is het resultaat 1, in de andere gevallen 0.

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

Met het volgende voorbeeld zal de AND-bewerking tussen de inhoud van de accumulator en de operand duidelijk worden. De machinetaalinstructie AND #$37 betekent dat de AND-operatie tussen de inhoud van de accumulator en $37 wordt uitgevoerd. Stel dat de inhoud van de accumulator $5D is. Dan ziet de bewerking er als volgt uit:

```
    $5D 01011101
    $37 00110111
AND ------------
    $15 00010101
```

Als resultaat is de inhoud van de accumulator %00010101 of $15 Dit komt volledig overeen met de BASIC-instructie:

```
A=A AND $37
```

In ons geval is A=$5D, zodat A=$5D AND $37 of A=93 AND 55. Als resultaat wordt $15 verkregen, wat decimaal gelijk is aan 21. Bij de AND-bewerking zijn de N-en Z-flag betrokken. Als het resultaat nul is wordt de Z-flag gezet, resultaten groter dan $7F (127) zetten de N-flag. De volgende tabel geeft de instructiecodes voor de verschillende adresseerwijzen.

adresseerwijze	AND
immedidate	$29
absolute	$2D
zeropage	$25
absolute X-indexed	$3D
absolute Y-indexed	$39
zeropage X-indexed	$35
indirect-indexed	$31
indexed-indirect	$21

De OR-verbinding

Bij de OR-bewerking wordt ook per bit de accumulator met de overeenkomstige bit van de operand verbonden. Is een van beide bits 1 dan is het resultaat 1, als beide bits 0 zijn is het resultaat 0.

```
0  ORA  0 = 0
0  ORA  1 = 1
1  ORA  0 = 1
1  ORA  1 = 1
```

Deze OR-bewerking wordt de zogenaamde inclusive OR-operatie genoemd. Bij de OR bewerking worden weer de Z-en N-flag gezet. De machinetaalinstructie: ORA #$37 betekent dat de bewerking op de inhoud van de accumulator moet worden toegepast. Stel dat deze inhoud weer $37 is, dan is de bewerking als volgt:

```
      $5D 01011101
      $37 00110111
ORA ----------
      $7F 01111111
```

Als resultaat wordt %01111111 of $7F verkregen. De overeenkomstige BASIC-instructie OR is:

```
A=A OR $37
```

De inhoud van A is $5D zodat A=$5D OR $37 of A=93 OR 55. Het resultaat is 127, overeenkomstig $7F.

De tabel van instructiecodes voor de verschillende instructiemethodes is als volgt:

adresseerwijze	ORA
immediate	$09
absolute	$0D
zeropage	$05
absolute X-indexed	$1D
absolute Y-indexed	$19
zeropage X-indexed	$15
indirect-indexed	$11
indexed-indirect	$01

De EXCLUSIVE-OR verbinding

Bij de exclusive-or operatie worden eveneens de overeenkomstige bits met elkaar verbonden. Het resultaat is 1 als de bits verschillend zijn en 0 als de bits gelijk zijn:

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

Bij deze bewerking worden overeenkomstig het resultaat de N-flag en de Z-flag gezet. In BASIC komt de EOR-instructie niet voor. U moet dan met een lus de bits een voor een verbinden. Ons voorbeeld is nu:

```
EOR #$37
```

De accumulator bevat weer $5D. De bewerking ziet er als volgt uit:

```
    $5D   01011101
    $37   00110111
EOR ----------

    $6A   01101010
```

Het resultaat is 01101010 of $6A. Decimaal is dit 106.

De tabel is nu:

adresseerwijze	EOR
immediate	$49
absolute	$4D
zeropage	$45
absolute X-indexed	$5D
absolute Y-indexed	$59
zeropage X-indexed	$55
indirect-indexed	$51
indexed-indirect	$41

De BIT-instructie

De BIT-instructie is iets bijzonders voor de 65XX-processoren. Deze instructie verandert geen registerinhoud maar werkt uitsluitend op de flags. Daarbij gebeurt het volgende: tussen de accumulator en de geadresseerde geheugenplaats wordt een AND-operatie uitgevoerd. Als het resultaat nul is, wordt de Z-flag gezet, in het andere geval gewist. Tevens wordt de zesde bit van de geadresseerde geheugenplaats in de V-bit overgenomen. De zevende bit komt in de N-bit. Zonder de registerinhouden te veranderen kunnen we zo de inhoud van beide bits controleren. Bijvoorbeeld:

```
    BIT $1234
```

Stel dat in de accumulator $10 staat en dat adres $1234 als inhoud $43 heeft, dan levert de AND-bewerking het volgende op:

```
    $10   00010000
    $43   01000011
AND ----------

    $00   00000000
```

De AND-operatie levert nul op, de Z-flag wordt gezet. De V-flag krijgt de waarde van bit 6 van de operand en wordt 1, bit 7 van de operand is 0 zodat de N-flag gewist wordt. Het resultaat is:

Z=1; V=1; N=0

Er zijn twee adresseermanieren:

adresseerwijze	BIT
zeropage	$24
absolute	$2C

3.6 De vergelijkingsinstructies

De groep vergelijkingsinstructies maakt vergelijkingen mogelijk tussen de register- en geheugenplaatsinhouden. Deze instructies veranderen noch de register-, noch de geheugeninhoud, maar beïnvloeden uitsluitend de flags. Op basis van de stand van de flags kunnen beslissingen worden genomen. De bijbehorende instructies worden in het volgende hoofdstuk besproken. Bij de vergelijkingesbevelen wordt de inhoud van de geadresseerde byte afgetrokken van de registerinhoud. De C-, de N- en de Z-flag worden overeenkomend met het resultaat van deze aftrekking al of niet gezet. Er zijn vergelijkingsinstructies voor alle drie werkregisters van de processor.

De CMP-instructie

Deze instructie vergelijkt (to CoMPare) de inhoud van de accumulator met die van een geadresseerde geheugenplaats. De inhoud van de operand wordt afgetrokken van die van de accumulator. Als een tekort optreedt, wordt de C-flag gewist, anders gezet. Is het resultaat nul, dus beide waarden zijn gelijk, dan wordt de Z-flag gezet, anders teruggezet. Is het resultaat groter dan $7F (127), dan wordt de N-flag gezet, anders gewist. Bijvoorbeeld:

```
CMP #$30
```

Stel dat de accumulator $50 bevat. De instructie trekt $30 af van $50. Het resultaat is $20. Er is geen tekort, dus de carryflag wordt gezet. De zeroflag wordt gewist, omdat het resultaat ongelijk is aan nul. De negativeflag wordt eveneens gewist, omdat het getal niet groter is dan $7F. Het resultaat is dus:

C=1; Z=0; N=0

Als de inhoud van de accumulator $30 is, levert de aftrekking de uitkomst nul op. Er treedt geen tekort op, dus de carryflag wordt gezet. De zeroflag wordt gezet en de negativeflag wordt niet gezet, omdat de uitkomst niet groter is dan $7F.

C=1; Z=1; N=0

De inhoud van de accumulator is nu $10. De uitkomst van de aftrekking $10-$30=$F0 en een tekort treedt op. De carryflag wordt gewist, de zeroflag en de negativeflag worden gezet omdat de uitkomst groter is dan $7F.

C=0; Z=0; N=1

Voor praktisch gebruik geldt het volgende:

C = 1 betekent: groter dan of gelijk aan
Z = 1 betekent: gelijk aan
C = 0 betekent: kleiner dan

Voor de beslissing 'groter dan' moeten twee flags worden getest:

C = 1 en Z = 0

Omdat de vergelijkingsinstructies alleen de flags beïnvloeden en niet de registerinhouden zijn deze instructies de basis voor de voorwaardelijke instructies die we in het volgende hoofdstuk zullen leren kennen. De volgende tabel toont de instructiecodes voor de verschillende manieren van adresseren:

adresseerwijze	code
immediate	$C9
absolute	$CD
zeropage	$C5
absolute X-indexed	$DD
absolute Y-indexed	$D9
zeropage X-indexed	$D5

De CPX-instructie

De CMP-instructie is identiek aan de CPX-instructie. Nu wordt de inhoud van de geadresseerde geheugenplaats afgetrokken van het X-register. Ook nu wordt de inhoud van het X-register niet veranderd. Er staan nu maar drie adresseermogelijkheden ter beschikking:

adresseerwijze	code
immediate	$E0
absolute	$EC
zeropage	$E4

De CPY-instructie

Voor de vergelijking met het Y-register staan ook drie adresseerwijzen ter beschikking:

adresseerwijze	code
immediate	$C0
absolute	$CC
zeropage	$C4

3.7 Voorwaardelijke spronginstructies

De instructies waarmee beslissingen kunnen worden genomen zijn de volgende stap. Basis voor deze beslissingen zijn de toestanden van de flags, die al uitvoerig aan de orde zijn geweest. De standen van de volgende vier flags kunnen als beslissingscriterium dienst doen: de Z-flag, de N-flag, de C-flag en de V-flag. Voor iedere flag zijn twee voorwaardelijke spronginstructies: bij de eerste wordt gesprongen als de flag gezet is, bij de tweede wordt gesprongen als de flag juist niet gezet is. De processor moet

weten naar welk adres gesprongen moet worden als aan de gestelde voorwaarde wordt voldaan, daarom hoort bij de instructie een operand. Deze operand geeft dit adres aan. Als operand kan het 16-bitadres de aftakking aangeven. De instructie neemt dan 3 bytes in beslag (opcode + operand). Omdat de voorwaardelijke sprongen meestal over korte afstanden gaan, heeft men een andere adresseerwijze bedacht. Er wordt niet absoluut gesprongen naar een bepaald adres, maar vanaf de stand van de instructieteller wordt het aantal stappen (adressen) waarover gesprongen moet worden, aangegeven. Dan kan met een 8-bits waarde worden volstaan. De totale instructie neemt nu 2 bytes in beslag. Met de 8-bits waarde kunnen 256 getallen worden voorgesteld. Er kan dus over 256 bytes naar voren gesprongen worden. Vaak is het echter gewenst om een stuk naar achteren te springen, bijvoorbeeld naar het begin van een lus. Daarom moeten met eeen 8-bits getal ook negatieve waarden worden voorgesteld. In de SBC-instructie (aftrekking) kunnen ook negatieve getallen voorkomen. Er wordt gekeken naar die getallen waarbij de zevende (meest linkse) bit gezet (=1) is. Deze getallen beschouwen we als negatieve getallen waarvan de waarde met het two's complement wordt bepaald.

%10000000	$80	-128
%10000001	$81	-127
...	...	...
%11111110	$FE	-2
%11111111	$FF	-1
%00000000	$00	0
%00000001	$01	1
%00000010	$02	2
...	...	...
%01111110	$7E	126
%01111111	$7F	127

U ziet, dat aan de hand van de zevende bit wordt beslist of een getal positief of negatief is. De betekenis van de N-flag wordt zo duidelijk. Bij de berekening van de relatieve sprong is het oog gericht op het adres van de volgende instructie achter de voorwaardelijke sprong. Voorbeeld: De spronginstructie staat op adres $C47A en er moet naar $C4BF gesprongen worden.

$C47A adres waar de Op-code van de spronginstructie staat
$C47B adres waar de sprongwaarde ingezet wordt
$C47C adres waar de Op-code van de volgende instructie staat
$C4BF doeladres

Het verschil tussen $C4BF en $C47C is $43. De inhoud van adres $C47B moet dus $43 zijn. Stel dat achterwaarts gesprongen moet

worden naar $C440. De sprongwaarde kan op verschillende manieren worden berekend. De eerste methode omvat het verschil tussen beide adressen, dat nu negatief is:

$C440 - $C47C = $FFC4 met tekort

Van deze waarde hebben we alleen het kleinste deel $C4 nodig als operand voor de spronginstructie. U kunt ook het positieve verschil en van de uitkomst het two's complement nemen:

```
$C47C - $C440 = $3C dit is %00111100
Het two's complement is    %11000011
                       +          1
                       ------------
                          %11000100 = $C4
```

Ook hier is de sprongwaarde $C4.

Welke voordelen heeft de relatieve adressering? In de eerste plaats worden slechts twee in plaats van drie bytes voor de instructie gebruikt. Behalve de besparing aan geheugenruimte is het voordeel dat een 2-bytes instructie door de processor sneller wordt uitgevoerd. Het belangrijkste voordeel komt voort uit de relatieve adressering - relatief ten opzichte van het uitgangsadres. De instructie geeft tenslotte alleen maar aan dat een bepaald aantal bytes naar voren of naar achteren moet worden gesprongen. Als zo'n programma-onderdeel ergens anders in het geheugen wordt gezet, hoeft aan het programma niets te worden veranderd, omdat het verschil ten opzichte van het uitgangsadres onveranderd blijft. Als een sprongadres absoluut zou worden aangegeven, moet dit adres bij elke programmaverschuiving mee worden veranderd. Dit is omslachtig en kan door de relatieve adressering worden vermeden. Een nadeel van de relatieve adressering is het beperkte aantal stappen waarmee gesprongen kan worden. Alleen maar 127 stappen naar voren of 128 naar achteren, gerekend vanaf de instructiebyte 129 naar voren en 126 naar achteren. In de praktijk levert deze beperking geen grote problemen op, omdat er zelden over grotere afstanden gesprongen moet worden. Mocht u de berekening van het relatieve adres omslachtig hebben gevonden, dan kunnen we u geruststellen. Het ging er ons alleen maar om het principe duidelijk te maken. Later zal de assembler deze taak voor zijn rekening nemen, u hoeft dan alleen maar het sprongadres aan te geven, de assembler berekent de sprong. De assembler geeft aan dat de sprong buiten de geoorloofde grens valt.

Sprongen op basis van de zeroflag:

Een sprong, ook wel afsplitsing of vertakking genaamd, wordt in

het Engels branch genoemd. Bij een gezette zeroflag wordt gespongen met de instructie branch on equal, afgekort tot BEQ. Moet juist bij een niet gezette zeroflag gesprongen worden dan is de instructie: branch on not equal, BNE.

Sprongen op basis van de carryflag:

Hier heten de instructies branch on carry set, BCS voor de sprong met gezette carryflag en branch on carry clear, BCC als de flag gewist is.

Sprongen op basis van de negativeflag:

Is de N-flag gezet, dan wordt bij de instructie branch on minus, BMI gesprongen, om bij gewiste N-flag te springen moet de instructie branch on plus, BPL gebruikt worden.

Sprongen op basis van de overflowflag:

Ook de V-flag kan als basis voor de voorwaardelijke sprongen gebruikt worden. De overeenstemmende instructies zijn branch on overflow set, BVS en branch on overflow clear, BVC. Vanwege de geringe betekenis van de V-flag worden deze instructies zelden gebruikt.

De tabel vat alle voorwaardelijke spronginstructies samen.

instructie	instructiecode
BEQ	$F0
BNE	$D0
BCS	$B0
BCC	$90
BMI	$30
BPL	$10
BVS	$70
BVC	$50

3.8 Spronginstructies

In tegenstelling tot de voorwaardelijke spronginstructies (branch-instructies), die we net hebben leren kennen, springt de onvoorwaardelijke spronginstructie rechtstreeks naar een aangegeven adres. Deze instructie hangt van geen enkele voorwaarde af en wordt altijd uitgevoerd. Naast de absolute adresseerwijze is er nog de indirekte adressering. Dit geldt speciaal voor de onvoorwaardelijke spronginstructies. Bij de indirekte sprong wordt niet naar het aangegeven adres gesprongen, maar naar het adres dat daar aangegeven staat. Het eerste adres dient dus als wijzer (pointer) naar het eigenlijke doeladres. Er zijn twee opeenvolgende adressen achter de Op-code beschikbaar als wijzer, waarbij de lo-byte voor de hi-byte van het doeladres komt. Voorbeeld:

JMP ($0302) Dit is een indirekte sprong naar adres $0302

In adres $0302 staat het lo-byte deel van het adres waarheen gesprongen wordt bv. $40. In adres $0303 staat de hi-byte bv. $C8. Er wordt nu gesprongen naar adres $C840. Deze adresseerwijze is alleen bij de JMP-instructie mogelijk. Alle adressen van het hele geheugen kunnen bereikt worden.

De tabel van de instructiecodes is:

adresseerwijze	code
absoluut	$4C
indirekt	$6C

Het bedrijfssysteem (operating system) van de Commodore 64 maakt van deze adressering ruim gebruik. Op de adressen $0300 tot $0330 staan alleen sprongvectoren. Dit heeft het grote voordeel dat door een simpele verandering van de vectoren de deelprogramma's van het operating system en de BASIC-interpreter voor eigen gebruik kunnen worden aangewend.

3.9 De telinstructies

Voor het effectief programmeren van lussen en tellers bezit de 6510 instructies die de register- of geheugeninhouden met 1 verhogen (increment) of met 1 verlagen (decrement). Deze incrementinstructies komen, samen met de voorwaardelijke sprongen, overeen met de NEXT-instructie in BASIC. De decrementinstructie wordt in BASIC met STEP-1 gesimuleerd.

INX

De inhoud van het X-register wordt met 1 opgehoogd. Overeenkomstig de uitkomst worden de N- en Z-flag gezet. In BASIC is de instructie:

```
X = X + 1
```

Als de waarde $FF met 1 wordt verhoogd, wordt op het overschot geen acht geslagen (de carryflag wordt niet gezet). De inhoud wordt 0 en de Z-flag wordt gezet.

INY

Dit is de overeenkomstige instructie voor de verhoging van het Y-register met 1. De beïnvloeding van de flags is analoog. Een instructie om de accumulator met 1 te verhogen heeft de 6510 niet.

INC

Deze instructie verhoogt de inhoud van een geheugenplaats met 1. Daarbij wordt, afhankelijk van het resultaat, weer de Z- en de N-flag gezet. Deze instructie onderscheidt zich van de vorige instructies doordat de inhoud van een geheugenplaats eerst gelezen wordt, daarna met 1 wordt verhoogd en vervolgens weer wordt weggezet (Read-Modify-Write). De instructies die we tot nu toe hebben leren kennen hebben alleen geheugeninhoud gelezen of geschreven, maar nooit beide na elkaar. Bij de INC-instructie wordt de inhoud van de accumulator niet veranderd.

In BASIC zouden we dit als volgt kunnen formuleren:

```
POKE M, PEEK(M)+1
```

Daarbij is M de geadresseerde geheugenplaats.

Voor de vermindering van de register- en geheugeninhouden zijn overeenkomstige instructies beschikbaar:

DEX en DEY

Deze instructies verminderen de inhoud van het X- resp. Y-register. Bij het verlagen van $00 tot $FF wordt de carryflag niet gezet. De Z- en N-flag worden, afhankelijk van de uitkomst, gezet. In BASIC is dit:

```
X = X - 1 en Y = Y - 1
```

DEC

Met deze instructie kan de inhoud van een geheugenplaats met 1 worden verminderd zonder dat de inhoud van de accumulator daarbij verloren gaat. Hierbij geldt hetzelfde als voor de INC-instructie.

De tabel met de instructiecodes is:

instructie	OP-code
INX	$E8
INY	$C8
DEX	$CA
DEY	$88

adresseerwijze	INC	DEC
absoluut	$EE	$CE
zeropage	$E6	$C6
absolute X-indexed	$FE	$DE
absolute Y-indexed	$F6	$D6

3.10 Instructies voor de beïnvloeding van de flags

Buiten de instructies om, die overeenkomstig het resultaat de flags beïnvloeden, kunnen de flags ook direkt door de programmeur gezet of gewist worden. Dit is bijvoorbeeld voor optellingen en

aftrekkingen gewenst. Deze instructies hebben natuurlijk geen operand nodig en zijn daarom 1-bits instructies (implied).

De carryflag

Met de instructie SEC (set carry) wordt de carryflag gezet, met CLC (clear carry) wordt de flag gwist. De beide instructies SEC en CLC moeten voor elke optelling en aftrekking gebruikt worden, omdat anders de uitkomsten verkeerd kunnen zijn.

De decimaalflag

Deze flag beslist of de pocessor de optellingen en aftrekkingen binair (flag gewist) of decimaal (flag gezet) uitvoert. In de decimale mode werkt de processor met BCD-getallen (binary coded decimals). De instructie SED (set decimal flag) zet de flag en de instructie CLD (clear decimal) wist de flag.

De interruptflag

De I-flag beslist of de processor bereid is een interrupt aan te nemen. Is de I-flag gezet met SEI (set interrupt disable), dan wordt geen interrupt aangenomen. Is de flag gewist met CLI (clear interrupt disable), dan kan de processor een interrupt aannemen.

De overflowflag

De overflowflag kan alleen maar gewist worden met de rechtstreekse instructie CLV (clear overflow). De tabel bevat de codes van deze 1-bits instructies.

instructie	code
CLC	$18
SEC	$38
CLD	$D8
SED	$F8
CLI	$58
SEI	$78
CLV	$B8

3.11 De schuifinstructies

De 6510-processor kent nog een groep instructies, waarvoor geen BASIC-equivalent beschikbaar is. Dit zijn de schuifinstructies. Daarbij wordt de bitinhoud van een register of de geadresseerde geheugenplaats een positie naar links of naar rechts verschoven. Is de instructie gericht op de accumulator, dan wordt gesproken van accumulatoradressering. Overeenkomstig de wijze van adresseren betreft het 1-, 2- of 3-bits instructies. Als een geheugenplaats is geadresseerd gaat het net als bij de INC- en DEC-instructie om het inlezen van de inhoud, het bewerken en het neerzetten van het resultaat in hetzelfde adres. De inhoud van de accumulator blijft onveranderd.

ASL

ASL betekent arithmetic shift left, de geadresseerde byte wordt een positie naar links verschoven. In de vrijkomende 0-bit (rechts) komt een 0 te staan. De meest linkse bit, bit 7, komt in de carrybit van het statusregister. We bekijken nu bijvoorbeeld de accumulator: ASL A, stel dat de inhoud van de accumulator $47 is.

```
$47 = %01000111 = 71 verschuiven naar links:
      %10001110   $8E, C=0
```

In ons geval krijgen we als resultaat $8E en de carryflag wordt gewist, omdat de accumulator in bit 7 een 0 bevatte. Als de accumulatorinhoud voor en na het verschuiven wordt vergeleken, blijkt de inhoud van de accumulator verdubbeld te zijn (van 71 naar 142). Dat is logisch, want zo gaat het ook met decimale getallen. Als daar de komma een plaats naar links wordt verschoven, wordt de waarde tienmaal zo groot. Met de ASL-instructie hebben we zo een mogelijkheid gevonden om een getal te verdubbelen. Een ander voorbeeld:

ASL A, de accumulator bevat nu $CD

```
$CD = %11001101 = 205 verschuiven naar links:
      %10011010   $9A, C=1
```

Omdat de carryflag gezet is, is een overschot opgetreden en het resultaat is nu 256+154=410, wat natuurlijk het dubbele van 205 is.

LSR

De LSR-instructie logical shift right stamt van de ASL-instructie. Nu wordt echter naar rechts geschoven met de bitinhouden. Bit 7 wordt 0 en de inhoud van bit 0 wordt in de carrybit geschoven. Een voorbeeld kan dit weer verduidelijken:

LSR A, stel dat de accumulator $CA bevat.

```
$CA    %11001010 = 202 verschuiven naar rechts
       %01100101   $65, C=0
```

Als resultaat wordt $65=101 verkregen met in de carrybit een 0. Zoals u wellicht hebt opgemerkt is het resultaat van een verschuiving naar rechts een halvering van de waarde. De carryflag geeft de inhoud van bit 0 weer. Anders gezegd toont de carrybit ons of bij het delen door twee een rest is opgetreden. Daarmee kan vastgesteld worden of een getal even of oneven is. De carryflag kan getest worden met BCC of BCS. Als met de LSR-instructie een geheugenplaats wordt geadresseerd, blijft de inhoud van de accumulator behouden.

ROL

Met de ROL-instructie (rotate left), kan de inhoud van een register of geheugenplaats cyclisch naar links worden verschoven (roteren). De inhoud van de carrybit wordt in bit 0 geschoven, terwijl de inhoud van bit 7 in de carrybit wordt gezet. Er vindt dus een cyclische verschuiving plaats met 9 bits (8 bits van het register plus de carryflag). Een voorbeeld zal dit verduidelijken:

ROL A, stel dat de accumulator $4B bevat en de carryflag gezet is.

```
$4B    %01001011  C=1 rotatie met C-bit naar links
       %10010111  $97, C=0
```

Alle bits worden een positie naar links verschoven, de carryflag komt in de vrijkomende bit 0 (=1), bit 7 wordt in de carrybit geschoven (=0). Ook bij deze rotatie verdubbelt de inhoud van de accumulator; een eventueel overschot van een eerdere verschuiving kan in de vorm van de carry erbij opgeteld worden.

ROR

De ROR-instructie (rotate right), staat tegenover de ROL-instructie en verschuift de inhoud van een register cyclisch naar rechts. De inhoud van de carrybit wordt in de vrijkomende bit 7 geschoven, de inhoud van bit 0 komt in de carrybit.

ROR A, de accumulator bevat $89 en de carryflag is gewist.

```
$89 = %10001001  C=0 rotatie met C-bit naar rechts
     %01000100  $44, C=1
```

Uit $89 wordt bij het roteren $44 verkregen, de carryflag is gezet en toont een rest bij de deling door twee aan. Bij alle schuifinstructies geldt bovendien dat de Z- en N-flag al dan niet gezet worden al naar gelang het resultaat resp. 0 of groter dan $7F is.

De volgende tabel bevat de instructiecodes voor alle adresseermogelijkheden.

adresseerwijze	ASL	LSR	ROL	ROR
accumulator	$0A	$4A	$2A	$6A
absolute	$0E	$4E	$2E	$6E
zeropage	$06	$46	$26	$66
absolute X-indexed	$1E	$5E	$3E	$7E
zeropage X-indexed	$16	$56	$36	$76

3.12 De subroutine-instructies

Een zeer belangrijke programmeertechniek die u van BASIC al kent is de techniek van de subroutines. In BASIC zijn daarvoor GOSUB, voor het aanroepen van, en RETURN voor het terugkeren uit een subroutine, beschikbaar. Wat is het verschil tussen een normale sprongopdracht zoals GOTO in BASIC en JMP in machinetaal bij het aanroepen van een subroutine? Als een subroutine wordt aangeroepen, moet een processor of de BASIC-interpreter onthouden vanaf welke positie de subroutine wordt aangeroepen, zodat bij de RETURN-instructie weer teruggesprongen kan worden naar het hoofdprogramma. De BASIC-interpreter doet dit automatisch voor ons: ook de 6510 neemt dit werk voor zijn rekening. Desondanks willen we weten hoe dit in zijn werk gaat. Opdat de processor weet naar welk adres hij na de RETURN-instructie terug moet springen, wordt bij het aanroepen van de subroutine het adres vanwaar gesprongen wordt opgeslagen. Daarvoor is een gedeelte van het geheugen gereserveerd, de zogenaamde stack. De stack ligt vast op de adressen $0100 t/m $01ff (256 t/m 511). Omdat de processor moet weten op welk adres van de stack hij het adres

waarnaar teruggesprongen moet worden, moet opslaan is er de stackpointer (stapelwijzer). Dit register van de processor kenden we al. De processor legt het adres (+2) van waaruit gesprongen wordt vast in een lo-byte en een hi-byte. De hi-byte wordt gezet in adres $0100 plus SP. Dan wordt de inhoud van de stackpointer met 1 verminderd en de lo-byte wordt in de stack gezet, op adres $0100+sp. Aansluitend wordt de stackpointer nog een keer met 1 verminderd. Nu wordt naar de subroutine gesprongen. Als de processor bij de RETURN-instructie komt, verloopt het proces in omgekeerde volgorde: de inhoud van de stackpointer wordt met 1 verhoogd en de hi-byte wordt gehaald. De hi-byte wordt nu in de instructieteller gezet. Het volledige adres waarnaar teruggesprongen moet worden staat nu in de instructieteller en de instructie die volgt op de opdracht: GOSUB wordt uitgevoerd, het programma wordt normaal vervolgd.

De procedure in de stack gaat dus als volgt: eerst wordt de waarde in de stack gezet en pas daarna wordt de stackpointer met 1 verlaagd. Bij het terughalen van een byte van de stack wordt eerst de stackpointer verhoogd en daarna de inhoud opgehaald. De stack groeit zo van $01FF naar $0100. Een voorbeeld:

```
$C480 JSR $2000 SP=$FA
```

De inhoud van adres $01FA wordt $C64, de stackpointer wordt SP=SP-1, de inhoud van adres $01F9 wordt $80, de stackpointer wordt SP=SP-1, de stackpointer heeft nu de waarde SP=$F8. Nu wordt naar het adres $2000 gesprongen. Stel dat daar direkt de RETURN-instructie staat:

```
$2000 RTS     SP=$F8
              SP=SP+1  PCL=($01F9)=$82
              SP=SP+1  PCH=($01FA)=$C4
              SP=$FA
```

De instructieteller (PC) bevat nu $C482. Deze waarde wordt nog met 1 verhoogd en wijst dan naar $C483, waar de volgende instructie na het oproepen van de subroutine op adres $C480 staat. De stack werkt volgens het principe "Last in First out", wat betekent dat de waarde die het laatst in de stack wordt geschoven er het eerst weer uitgehaald wordt. Door dit principe is het mogelijk subroutines in subroutines te voegen. Als vanuit een subroutine en tweede subroutine wordt aangeroepen, wordt bij de volgende RETURN-instructie het laatste adres gehaald waarnaar gesprongen moet worden zodat dan verdergegaan wordt met de eerste subroutine. Na de volgende RETURN-instructie wordt met het hoofdprogramma verdergegaan.

Als u de werking van de stack hebt begrepen, kunt u de stack gebruiken voor het tijdelijk opslaan van eigen gegevens. Dit wordt in 3.13 nader besproken.

De tabel bevat de codes voor het aanroepen van subroutines en de terugkeer daaruit.

instructie	Op-code
JSR	$20
RTS	$60

3.13 De stackinstructies

De 6510 heeft de mogelijkheid de inhoud van de accumulator en van het statusregister op de stack te zetten en daar ook weer vandaan te halen. Bij het wegzetten wordt de stackpointer automatisch met 1 verlaagd en bij het ophalen met 1 verhoogd.

PHA

De instructie PHA (push accu) zet de inhoud van de accumulator op de stack en velaagt de stackpointer met 1. De inhoud van de accumulator verandert daarbij niet.

PHP

Met de PHP-instructie (push processor status) wordt het volledige statusregister op de stack gezet en de stackpointer met 1 verminderd. De inhoud van het statusregister blijft onveranderd.

PLA

De PLA-instructie (pull accu) staat tegenover de PHA-instructie. De stackpointer wordt met 1 verhoogd en een byte van de stack wordt in de accumulator gezet. Overeenkomstig de waarde worden Z- en N-flag al dan niet gezet.

PLP

Met de PLP-instructie (pull processor status) kan een byte van de

stack worden gehaald en in het statusregister worden gezet. De instructie staat tegenover PHP. De tabel bevat de instructiecodes.

instructie	code
PHA	$48
PHP	$08
PLA	$68
PLP	$08

3.14 Instructies voor de interruptbehandeling

Deze instructies worden voorlopig niet gebruikt en worden slechts beknopt behandeld. De 6510-processor bezit de mogelijkheid een lopend progamma van buitenaf te onderbreken. Daarvoor moet de zogenaamde interruptleiding (IRQ, interrupt request) van de processor worden geactiveerd. Bij een interrupt gebeurt iets soortgelijks als bij het aanroepen van een subroutine. De processor onderbreekt het werk bij een lopend programma en zet de inhoud van de instructieteller (PC) op de stack. Tevens wordt de inhoud van het statusregister op de stack gezet, omdat de onderbreking van het programma op ieder willekeurig moment kan plaatsvinden. Nu wordt naar een deelprogramma gesprongen. Het eerste adres van dit programma wordt aangegeven door de inhoud van de adressen $FFFE en $FFFF. De instructieteller krijgt de waarde die op deze adressen staat.

Behalve de onderbreking van buitenaf kan een lopend programma vanuit het programma zelf worden onderbroken met de instructie BRK (break). Ook nu worden de instructieteller en de inhoud van het statusregister bewaard op de stack.

Om uit een onderbrekingsprogramma weer in het hoofdprogramma terug te keren is een soortgelijke instructie als RTS bij het gebruik van subroutines beschikbaar. De instructie RTI (return from interrupt) haalt de instructieteller en de inhoud van het statusregister van de stack en gaat verder met het hoofdprogramma zonder dat de flags veranderd zijn. De instucties en de Op-codes zijn:

instructie	Op-code
BRK	$00
RTI	$40

NOP

Tenslotte mag een instructie niet onvermeld blijven, de NOP-instructie (no operation). Deze instructie doet helemaal niets en dient als opvulling van openvallende plaatsen in een programma, omdat het vaak gewenst is de rest van het programma niet te verschuiven. Ook wordt de instructie gebruikt in programmalussen omdat ook deze instructie een zekere verwerkingstijd nodig heeft.

instructie	Op-code
NOP	$EA

4 DE INVOER VAN MACHINETAALPROGRAMMA'S

Nu alle instructies van de processor en hun functies bekend zijn, gaan we ons richten op het maken van machinetaalprogramma's en op de wijze van invoer.

Een machinetaalprogramma bestaat uit een rij instructies en de bijbehorende operanden voor zover die nodig zijn. Als eerste voorbeeld wordt een karakter in het beeldschermgeheugen van de computer gezet. In BASIC luiden de instructies:

```
POKE 1024,1  :REM BEELDSCHERMCODE VOOR DE LETTER A
POKE 55296,7 :REM KLEURCODE VOOR GEEL
```

Als beide instructies worden uitgevoerd, verschijnt in de linker bovenhoek van het scherm de gele letter A.

In machinetaal worden de POKE-instructies vervangen door STA. Met deze instructie kan de inhoud van de accumulator op het aangegeven adres (1024) worden gezet. Eerst wordt de accumulator rechtstreeks (immediate) geladen met de waarde 1.

```
LDA  #1
STA  1024
```

Op dezelfde manier wordt adres 55296 direkt geladen met de waarde 7, via de accumulator.

```
LDA  #7
STA  55296
```

Als deze instructies zo worden ingevoerd, meldt de computer '?SYNTAX ERROR'. De Commodore begrijpt vanuit de positie na het aanzetten van de computer immers alleen BASIC. Er moet dus anders te werk worden gegaan. Een machinetaalprogramma is een reeks instructies en operanden in het geheugen. Eerst moet de instructie omgezet worden in de instructiecode (Op-code). Deze codes zijn in het voorafgaande hoofdstuk steeds vermeld. Ook staat achterin dit boek een volledig overzicht. Voor de LDA-instructie, waarbij de accumulator direkt (immediate) moet worden geladen is de code $A9. Na de LDA-instructie wordt in het volgende adres de operand gezet, dit is hier het getal 1, hexadecimaal $01. Bij de STA-instructie wordt de absolute adresseerwijze gebruikt. De code is $8D. De operand is in dit geval een geheugenadres. Dit adres is een 16-bits adres. In twee bytes komt dit adres te staan 1024=$0400. Na de Op-code staat

eerst de lo-byte $00 op het eerste en de hi-byte $04 op het tweede adres. Ook voor de tweede instructie wordt deze procedure gevolgd. Het adres 55296 is hexadecimaal $D800.

Het programma met hexadecimale getallen wordt:

```
LDA #$01
STA $0400
LDA #$07
STA $D800
```

De lo-byte van $0400 is $00, de hi-byte is $04. De instructie STA $0400 wordt $8D (code) $00 (lo-byte) $04 (hi-byte). LDA #$07 en STA $D800 worden $A9, $07, $8D, $00, $D8.

Het hele programma ziet er zo uit:

$A9, $01, $8D, $00, $04, $A9, $07, $8D, $00, $D8

Deze bytevolgorde moet in het geheugen worden gezet. Waar moet het programma worden geplaatst? We moeten een gebied in het geheugen opzoeken dat niet wordt gebruikt door het operating system of de BASIC-interpreter. Bij de Commodore 64 is een dergelijk gebied beschikbaar op de adressen 49152 t/m 53247, dit is $C000 t/m $CFFF. Dit gebied bevat 4K adressen en is ook voor zeer grote machinetaalprogramma's groot genoeg. Ons programma wordt gezet op de adressen $C000 en verder. Dit kan met een BASIC-programmaatje worden gedaan. De hexadecimale getallen worden omgezet in decimale:

169, 1, 141, 0, 4, 169, 7, 141, 0, 216

```
100 FOR I=0 TO 9
110 READ A:POKE 49152+I, A
120 NEXT I
130 DATA 169, 1, 141, 0, 4, 169, 7, 141, 0, 216
```

Als dit programma met RUN wordt gestart, worden de waarden vanaf adres 49152 in het geheugen gezet. Nu kan het machinetaalprogramma worden uitgevoerd. Daarvoor wordt de BASIC-instructie SYS gebruikt. Met SYS 49152 wordt een machinetaalprogramma met startadres 49152 uitgevoerd. Dit programma wordt beschouwd als een subroutine vanuit BASIC. Pas op, na de uitvoering van de instructie: met STA $D800 wordt de Op-code voor de volgende instructie door de processor opgehaald. Als de inhoud van dit adres niet herkenbaar is, kan de processor in een oncontroleerbare toestand komen en zal meestal onbestuurbaar blijken te zijn. Slechts het uit- en weer aanzetten van de computer biedt dan uitkomst. Het programma is dan echter verloren

en u moet weer van voren af aan beginnen.

Als de processor de 4 instructies heeft uitgevoerd moet hij de controle weer teruggeven aan de BASIC-interpreter omdat dit machinetaalprogramma immers als subroutine wordt geïnterpreteerd. Met alleen de RTS-instructie $60=96 moet het programma worden afgesloten. Aan het eind van het programma moet dan staan:

```
140 POKE 49152+10, 96
```

Na RUN meldt de computer READY, vervolgens wordt het programma gestart met:

```
SYS 49152
```

Na het indrukken van de RETURN-toets verschijnt ogenblikkelijk de gele letter A in de linker bovenhoek van het scherm. De computer meldt: READY.

U bent niet de enige die deze gang van zaken omslachtig vindt. Daarom heeft men naar wegen gezocht om de procedure te automatiseren. Daarvoor hebt u tenslotte uw computer! Er is dus een programma nodig dat de machinetaalinstructies zoals LDA #$01 aanneemt, de code bepaalt en met de bijbehorende operand op de daarvoor bestemde adressen in het geheugen zet. Deze programma's zijn beschikbaar, ze heten assembler. Opdat u meteen met een assembler kunt werken en uw plezier niet vergald wordt met ellenlange omzettingen en speurtochten in tabellen, hebben wij voor u een complete assembler in BASIC geschreven. Voordat de werking van het assemblerprogramma wordt uitgelegd, wordt eerst ingegaan op andere hulpprogramma's die bij het werken in machinetaal van pas kan komen.

In de eerste plaats zijn dat zogenaamde monitorprogramma's. Met een monitor heeft u rechtstreeks toegang tot het geheugen en de registers van de processor. Met de monitor kan de inhoud van een geheugenplaats of register worden gecontroleerd en desgewenst veranderd worden. Bovendien kan vanuit de monitor een machinetaalprogramma worden gestart. Meestal bevatten monitoren commando's om machinetaalprogramma's te laden en te bewaren op cassetteband of diskette. Als u een monitor ter beschikking hebt kunt u het machinetaalprogramma hexadecimaal invoeren. Bij kleine programma's of veranderingen is dit goed te doen. Vaak bevatten monitorprogramma's ook een zogenaamde disassembler. Dit programma doet het omgekeerde van de assembler: de inhouden van de adressen worden omgezet in mnemonische vorm, de symbooltaal voor de instructies. Uit $A9, $01 wordt weer LDA #$01 gevormd. Zo'n disassembler, in BASIC geschreven, staat ook in dit boek. Met een dergelijk programma kunnen niet alleen eigen programma's worden

omgezet, maar kunnen ook delen van het operating system of de BASIC-interpreter in begrijpelijker vorm worden omgezet. Uit deze programma-onderdelen kunt u inspiratie opdoen voor uw eigen programma's.

4.1 De werking en de gebruiksaanwijzing van een assembler

We maken nu een programma in machinetaal, dat de voordelen toont van een assembler. De gehele karakterset van de Commodore 64 moet op het beeldscherm worden gezet. Eerst stellen we vast hoe dat in BASIC gaat. De Commodore 64 kan 256 verschillende tekens op het scherm zetten, omdat de beeldschermcodes van 0 t/m 255 gaan. In BASIC lossen we het probleem met een lus op.

```
100 X=0
110 A=X
120 POKE 1024+X, A:REM BEELDSCHERMCODE
130 A=1
140 POKE 55296+X, A:REM KLEURCODE
150 X=X+1
160 IF X <> 256 THEN 110
170 END
```

Na RUN wordt de gehele karakterset in ongeveer 7 seconden op het scherm gezet. Het programma is met opzet in een vorm geschreven die omzetting in machinetaal vergemakkelijkt. Aan het werk dus maar.

```
100 X=0  LDX #0
```

Voor de variabele X wordt het X-register gebruikt en de waarde van de variabele A wordt weggezet in de accumulator.

```
110 A=X      TXA
```

De inhoud van het X- register wordt in de accumulator gekopieerd. Het X-register verandert niet.

```
120 POKE 1024+X, A  STA 1024, X
```

De inhoud van de accumulator moet in adres 1024+X gezet worden. De geïndexeerde adresseerwijze wordt daarom gebruikt. De operand is $0400=1024, bij de indexed X adresseerwijze wordt automatisch

de inhoud van register X opgeteld bij de operand waardoor het adres gevonden wordt waarin de accumulator gezet moet worden.

```
130 A=1      LDA #1
```

De accumulator wordt direkt met de kleurcode (1=wit) geladen.

```
140 POKE 55296+X, A  STA 55296, X
```

De inhoud van de accumulator wordt in adres 55296+X gezet.

```
150 X=X+1    INX
```

De inhoud van het X-register wordt met 1 verhoogd. De instructie INX increment X doet dit direkt (implied, zonder operand).

```
160 IF X;:256 THEN 110 ??
```

Deze instructie vereist enig overleg. Er moet naar regel 110 teruggesprongen worden als de inhoud van het X-register ongelijk is aan 256. Omdat de grootste waarde die dit register kan bevatten 255 is, zal het register de inhoud $00 krijgen met overschot nadat er 1 bij opgeteld wordt. Dit overschot kan verder buiten beschouwing worden gelaten. Als de inhoud van het register nul wordt, wordt de zeroflag gezet. Is de waarde van het X-register nog ongelijk aan nul dan is de zerobit 0. Met de instructie BNE (branch not zero) wordt getest of de zeroflag al dan niet gezet is. Als de Z-flag niet gezet is wordt gesprongen.

```
160 IF X;:256 THEN 110 BNE regel 110
```

In machinetaal bestaat de instructie 'ga naar regel 110' natuurlijk niet. Er moet gesprongen worden naar de geheugenplaats waar de instructie van regel 110 staat, maar dit adres is nog niet bekend. Eerst moet het startadres van het programma worden gekozen. Adres $C000=49152 is geschikt.

```
100 $C000 LDA #0
110 $C002 TXA
120 $C003 STA $0400, X
130 $C006 LDA #1
140 $C008 STA $D800, X
150 $C00B INX
160 $C00C BNE $C002
170 $C00E RTS
```

De instructieteller is verder gezet al naar gelang de instructie 1, 2 of 3 bytes in beslag neemt (code plus operand). De sprong in regel 160 moet gericht zijn op adres $C002. Voor de laatste keer

nemen we de moeite om het programma in machinecodes om te zetten.

```
100   $C000   A2 00      LDX #0
110   $C002   8A         TXA
120   $C003   9D 00 04   STA $0400,X
130   $C006   A9 01      LDA #1
140   $C008   9D 00 D8   STA $D800,X
150   $C00B   E8         INX
160   $C00C   D0 ??      BNE $C002
170   $C00E   60         RTS
```

De sprong van regel 160 naar 110 moet nog berekend worden. De voorwaardelijke sprong wordt als volgt berekend (zie ook 3.7). Eerst wordt het positieve verschil tussen $C00C+2=$C00E (de instructieteller is inmiddels met 2 verhoogd) en $C002 berekenen. Daarna wordt het two's complement bepaald.

```
   $C00E
-  $C002
--------
   $000C


$0C = %00001100
      %11110011
+           1
   ------------
      %11110100  = $F4
```

De sprong die bij de BNE-instructie hoort in regel 160 is $F4.

Als deze waarde met de andere wordt ingevoerd met een BASIC-programma of via de M-instructie van de stap-voor-stap simulator, kan het eerste machinetaalprogramma worden uitgeprobeerd. Typ in:

```
SYS 49152
```

Een druk op de RETURN-toets toont onmiddellijk de volledige karakterset op het scherm. In BASIC duurde de uitvoering nog 7 seconden, maar nu is slechts een onderdeel van een seconde nodig. Het is een indrukwekkende demonstratie van de snelheid van een machinetaalprogramma. De assembler om het maken van machinetaalprogramma's te versnellen en te vergemakkelijken krijgt nu de aandacht.

5 DE ASSEMBLER

De assembler maak het ons mogelijk om een machinetaalprogramma op dezelfde manier als in BASIC in te voeren. Net zo eenvoudig als in BASIC kunnen regels worden veranderd, ingevoegd en verwijderd. Na het regelnummer kan een merk (label) geplaatst worden en vervolgens de assemblerinstructie met de bijbehorende operand. Opdat u later ook nog uw eigen programma's begrijpt, is het mogelijk commentaar achter een puntkomma te plaatsen. Net als bij het REM-statement in BASIC wordt bij de uitvoering van het programma niet op het commentaar gelet. Een volledige assemblerregel kan er zo uitzien:

```
100 TEKST LDA $70,X ;  STARTWAARDE OPHALEN
```

Dit zogenaamde bron- of sourceprogramma kan op diskette worden weggezet. Dit assemblerprogramma moet ter onderscheiding van de daarmee geproduceerde machinetaalprogramma's aan het eind van de naam ".SRC" krijgen. Nu kan de assembler geladen worden en vervolgens met RUN worden gestart. De naam van het programma dat omgezet of geassembleerd moet worden, wordt opgevraagd. Het programma wordt van de diskette gelezen en omgezet in machinecodes, die rechtstreeks in de bestemde adressen worden gezet. Bovendien kan de assembler desgewenst nog een zogenaamde assemblerlisting maken, die naast de regelnummers en de instructiecodes nog de lijst met hexadecimale codes en operanden geeft. Deze assemblerlisting kan niet alleen op het scherm, maar ook op een aangesloten printer worden uitgevoerd. Bij het assembleren worden ook de adressen van de sprongen automatisch berekend. U hoeft het sprongadres niet absoluut aan te geven maar u kunt volstaan met een merk of label. Het programma uit het vorige hoofdstuk ziet er dan zo uit:

```
100          LDA #0
110  LABEL   TXA
120          STA $0400,X
130          LDA #1
140          STA $D800,X
150          INX
160          BNE LABEL
170          RTS
```

Het adres waarnaar gesprongen moet worden wordt van een naam (label) voorzien. Als de assembler het programma afwerkt en op een label stoot, merkt hij die op met de waarde van de instructieteller. In het voorbeeld heeft de instructieteller in

regel 110 de waarde $C002. Aan deze waarde van de instructieteller kent de assembler de naam LABEL toe. In regel 160 komt hij weer het woord LABEL tegen. De assembler weet nu dat LABEL voor de waarde $C002 van de instructieteller staat. Uit de stand in regel 160 en de waarde van de instructieteller die hoort bij LABEL kan de sprong voor de spronginstructie berekend worden. Terwijl de assembler het programma afwerkt, wordt automatisch de code voor de instructies en hun operanden in het geheugen gezet, zodat het programma na de assemblergang gereed is voor uitvoering. Bij deze gang van zaken kan het voorkomen dat de assembler op een label stoot dat nog niet gedefinieerd is.

Voorbeeld:

```
100           LDA $40
110           BEQ VERDER
120           LDX #$FF
130  VERDER   STX $D840
140           RTS
```

In regel 110 moet worden gesprongen naar het adres waarvan het label de naam VERDER heeft. Deze plaats is nog niet gedefinieerd omdat het programma de naam VERDER nog niet is tegengekomen. Daarom maken we gebruik van een trucje. De assembler gaat eerst een keer door het hele programma en legt de adressen van alle labels vast. Daarna wordt weer van voren af aan begonnen zodat de eigenlijke assemblering kan plaatsvinden. Komt de assembler weer bij regel 110, dan kent hij het adres (de stand van de instructieteller) al waar VERDER staat uit de eerste ronde. De assembler moet een programma dus twee keer doorlopen. Men spreekt ook van een 2-Pass Assembler. Volgens dit principe werkt ook onze assembler. Om te kunnen zien hoever de assembler met zijn werk gevorderd is, laat hij steeds het nummer van de pas afgewerkte regel zien.

Als een assemblerprogramma op de aangegeven manier ingevoerd wordt, doorzoekt de BASIC-interpreter de programmaregels op BASIC-statements en zet deze, indien gevonden, om in 1-byte codes. Daardoor kunnen ook woorden waarin BASIC-statements zitten, zoals ON, TO maar ook = en *, niet meer door de assembler worden begrepen. Ze zijn immers niet als normale tekst in het geheugen gezet.

Daarom moet u eerst het volgende BASIC-programma laden en starten met RUN. Als u nu de programmaregels invoert worden de BASIC-statements niet meer omgezet in de bijbehorende codes, zodat de assembler het woord kan interpreteren. Wilt u later weer een normaal BASIC-programma invoeren, dan wordt met:

```
SYS 53181
```

de oorspronkelijke toestand weer ingesteld.

```
100 REM*****************************
110 REM VOORLADER ASSEMBLERPROGR.**
120 REM*****************************
130 FOR I=53100 TO 53191
140 READ X:POKE I,X:S=S+X:NEXT I
150 DATA 169,119,160,207,141,  2,  3,140,  3,  3, 96, 32
160 DATA  96,165,134,122,132,123, 32,115,  0,170,240,243
170 DATA 162,255,134, 58,144,  6, 32,121,165, 76,225,167
180 DATA  32,107,169,160,  0,162,  0,189,  0,  2,232,201
190 DATA  32,240,248,201, 48,144,  4,201, 58,144,240,153
200 DATA   0,  2,201,  0,240,  7,189,  0,  2,200,232,208
210 DATA 242,200,200,200,200,200, 76,162,164,169,131,160
220 DATA 164,141,  2,  3,140,  3,  3, 96
230 IF S<>11096 THEN PRINT "FOUTEN IN DATA !!":END
240 SYS 53100:PRINT"OK"
```

U moet dit progamma op iedere diskette zetten waarop u ook assembler-sourceprogramma's zet. U moet erop bedacht zijn na de invoer van het assemblerprogramma bovenstaand programma te laden en te RUNnen. Mocht u dit een keer vergeten dan is het voldoende om ook na de uitvoering van dit programma het sourceprogramma te laden en iedere regel door het drukken van de RETURN-toets opnieuw over te nemen en dit programma dan opnieuw weg te zetten. We keren terug naar het voorbeeldprogramma op de vorige bladzijden en voegen aan de regels 100 t/170 nog een regel 180 toe:

```
180 .EN
```

Dit betekent voor de assembler dat uw programma hier wordt afgesloten. Het programma wordt onder de naam TEST .SRC op diskette weggezet. U moet niet vergeten eerst het vorige BASIC-programma uit te voeren. Nu moet de assembler geladen worden en vervolgens gestart. Op het beeldscherm verschijnt:

```
            6510 - ASSEMBLER

SOURCE-FILE-NAAM ? TEST

LISTING J/N ? J

PRINTER J/N ? N
```

Na enige tijd verschijnt PASS 1 op het scherm en de diskdrive gaat lopen. Daarna verschijnen de bewerkte regelnummers 100 t/m 180. Na de tweede pass veschijnt de listing op het beeldscherm.

PASS 2

```
C000 A2 00       100          LDX     #0
C002 8A          110 LABEL    TXA
C003 9D 00 04    120          STA     $0400,X
C006 A9 01       130          LDA     #1
C008 9D 00 D8    140          STA     $D800,X
C00B E8          150          INX
C00C D0 F4       160          BNE     LABEL
C00E 60          170          RTS
                 180          .EN
```

LABEL C002

Dan vraagt de assembler of het vervaardigde machinetaalprogramma op diskette moet worden gezet. Antwoordt u met J.

```
WEGZETTEN OP SCHIJF J/N       ? J
```

Het programma wordt onder de naam TEST .OBJ op schijf gezet (OBJ=objectprogramma). Vervolgens worden gegevens over de gebruikte adressen en eventuele fouten uitgevoerd:

```
C000 / C00F / 000F
SOURCEFILE IS TEST. SRC
 0 FOUTEN
```

De assembler kan verder nog een overzicht van de gebruikte namen voor de labels geven:

```
LABELNAMEN J/N ? J
SORTEREN J/N ? N
```

LABEL C002

Het overzicht van de gebruikte labelnamen kan ook in alfabetische volgorde worden gegeven. Omdat in dit voorbeeldprogramma slechts de labelnaam LABEL is gedefinieerd, is alfabetiseren overbodig. Het gemaakte machinetaalprogramma staat, behalve op schijf onder de naam TEST .OBJ, ook in het geheugen. Het programma is klaar voor de uitvoering en het intypen van:

```
SYS 49152
```

en indrukken van RETURN

levert weer de volledige karakterset op het scherm.

Iedere regel van het sourceprogramma bestaat uit een regelnummer, dan eventueel een labelnaam, vervolgens het instructiewoord (mnemonic) zoals LDA, dan de operand (bij 1-byte instructies is dit natuurlijk niet nodig) en tenslotte na een puntkomma eventueel commentaar. Wij raden u aan van deze mogelijkheid rijkelijk gebruik te maken en precies te beschrijven wat u aan het doen bent. Als u namelijk later het programma weer wilt gebruiken, uitbreiden of veranderen zal dit zeker nuttig blijken te zijn. Ook zijn de programma's voor andere gebruikers veel beter leesbaar. Voor de labelnamen kunt u 1, 2, 3, 4 of maximaal 5 letters gebruiken. Natuurlijk mogen de namen van twee verschillende labels niet gelijk zijn. Behalve de impliciet gedefinieerde labelnamen die we al kenden is het nog mogelijk en zinvol aan labelnamen direkt een waarde toe te kennen en in het programma verderop de naam te gebruiken. Dit maakt de programma's veel overzichtelijker en beter te doorzien. In het voorbeeld wordt het eerste beeldschermadres (1024=$0400) van de naam VIDEO voorzien en adres 55296=$D800 van de naam KLEUR:

```
PASS 2

0400                  70 VIDEO  =     $0400
D800                  80 KLEUR  =     $D800
C000                  90        *=    $C000
C000 A2 00           100        LDX   #0
C002 8A              110 LABEL  TXA
C003 9D 00 04        120        STA   VIDEO,X
C006 A9 01           130        LDA   #1
C008 9D 00 D8        140        STA   KLEUR,X
C00B E8              150        INX
C00C D0 F4           160        BNE   LABEL
C00E 60              170        RTS
                     180        .EN
```

De uitvoer van de gesorteerde labels levert op:

```
KLEUR   D800        LABEL  C002
VIDEO   0400
```

In regel 90 staat weer iets nieuws. Met het sterretje * wordt de instructieteller op het aangegeven startadres $C000 gezet. Voordat met de instructies wordt begonnen moet het startadres worden gegeven, zodat de assembler weet vanaf welke geheugenplaatsen het machinetaalprogramma moet worden gezet. Een tweede vrije plaats in het geheugen is bijvoorbeeld de cassettebuffer vanaf $033C t/m $03FB (828 t/m 1019). Verder kan het gebied van de BASIC-interpreter niet gebruikt worden omdat de assembler dit voor eigen gebruik nodig heeft.

Het voordeel van het gebruik van namen voor direkte labels is onmiskenbaar. Ten eerste kan door een geschikte keuze van de namen de betekenis van een bepaalde geheugenplaats uit de naamgeving worden herkend. Ten tweede is het veel eenvoudiger om een programma te veranderen. Als het beeldschermgeheugen door ons verlegd is naar een andere positie binnen het geheugen, is het voldoende om het adres VIDEO bij het begin van het programma te veranderen. Alle adressen die op dit VIDEO-adres betrekking hebben, worden zo mee veranderd. Het voordeel wordt natuurlijk groter naarmate de adreslabels vaker in het programma voorkomen. U moet er direkt een gewoonte van maken zoveel mogelijk labels te gebruiken.

Er is nog een pseudo-instructie beschikbaar, de assemblerdeclaratie. Deze instructie maakt het mogelijk om waarden naar keuze, dus data of teksten, binnen het machinetaalprogramma te plaatsen. De instructie luidt:

```
.BY
```

Na .BY moet een waarde van 0 t/m 255 volgen, die op de aangegeven plaats in het programma wordt geplaatst. Behalve constanten kunt u ook karakters aangeven. De waarde mag echter niet groter zijn dan 255.

```
.BY 100
.BY $7F
.BY CR
```

Aan deze instructie is nog iets extra's toegevoegd. Vaak is het namelijk gewenst om een 16-bits adres in twee 8-bits waarden te verdelen. Daarvoor worden de operatoren > en < gebruikt. Het groter dan teken geeft de hi-byte (bit 8 t/m 15) aan, het kleiner dan teken de lo-byte (bit 0 t/m 7). Voorbeeld:

```
100  CONST    =    $AB3F
110  .BY  <CONST
120  .BY  >CONST
```

Dit programma-onderdeel plaatst de waarden $3F en $AB naast elkaar in het programma. Deze operatoren kunnen ook bij de onmiddellijke adresseerwijze (immediate) gebruikt worden door het cijferkruisje # te gebruiken.

```
130  LDA  #<CONST
140  LDY  #>CONST
```

Om de adresseerwijze voor de zeropage te gebruiken moet, om dit aan de assembler mee te delen, een sterretje (*) aan de operand

voorafgaan. Als geen * geplaatst is wordt door de assembler de absolute adresseerwijze gebruikt. Bij de geïndexeerde adressering, die alleen met zeropage-adressen werkt, is dit niet gewenst. Bekijkt u het volgende voorbeeld:

```
00B0                100 START  =     $B0
C000 AD B0 00       110        LDA   START
C003 A4 B0          120        LDY   *START
C005 8D 27 00       130        STA   $27
C008 84 60          140        STY   *$60
C00A 24 B0          150        BIT   *START
```

Zonder het sterretje * wordt de absolute 3-bytes instructie genomen, zoals in regel 110 en 130 te zien is. Met het sterretje wordt de zeropage-adresseerwijze gekozen. Deze neemt twee bytes in beslag (regels 120, 140 en 150). En daarmee kent u alle functies van de assembler; u kunt nu volop aan de slag met machinetaalprogramma's. Op de volgende bladzijden vindt u de assemblerlisting en een beschrijving van de onderdelen. Tevens is een lijst van de gebruikte variabelen toegevoegd.

Het assemblerprogramma mag niet klakkeloos ingetypt worden. Bij het intypen kunt u het beste de toegevoegde beschrijving goed lezen, zodat u begrijpt wat de assembler doet en hoe hij dat doet. U kunt daardoor niet alleen iets over de werkwijze van de assembler leren, maar ook over machinetaal en in het bijzonder over de verschillende adresseermethodes.

```
100 REM 6510 ASSEMBLER     12/83
110 PRINTCHR$(147):PRINT:PRINT:PRINT,"6510 - ASSEMBLER":
    PRINT:DG=8
120 INPUT"SOURCE-FILE-NAAM ";SN$
130 IFRIGHT$(SN$,4)=".SRC"THENSN$=LEFT$(SN$,LEN(SN$)-4)
140 DD$="0":REM DRIVENUMMER
150 INPUT"LISTING  J/N      ";A$:IFA$<>"J"THENPM=1:GOTO190
160 PF=4:PG=3
170 INPUT"PRINTER  J/N      ";A$:IFA$="J"THENPG=4
180 OPENPF,PG
190 GOSUB5000
200 A=0:AD=49152:PRINT:PRINT:PA=A
210 PRINT"PASS 1":GOSUB4000:PRINT"PASS 2":FF%=0:FE%=0
220 OP$=DD$+":"+SN$+".SRC"
230 OPEN8,DG,0,OP$
240 GET#8,A$,A$:REM STARTADRES
250 IFPM=1THENPRINTCHR$(145),,ZN$
260 F%=0:IFAD>65535THENPRINT:PRINT:
    PRINT"BEREIK TE HOOG          ":GOTO1000
```

```
270 A=AD:GOSUB3240:PR$=A$+"  ":GOSUB2000:
    IFLEFT$(X$,3)=".EN"THEN1000
280 XX$=LEFT$(X$,1):IFXX$="*"THENPR$="                    ":
    LN$="      "
290 IFXX$="."ORXX$="*"ORXX$="="THENGOSUB2900:GOTO380
300 ONLM%GOTO320
310 SA=OF+AD:PA=AD:LM%=1
320 XX$=LEFT$(X$,3):FORJ=0TONN%:IFXX$=MN$(J)THEN350
330 NEXT
340 FL$(1)="A":A%=1:F%=1:GOSUB1520:GOTO370
350 GOSUB2400:F%=0:IFT%=5ANDT%(J,9)>0THENT%=9:REM RELATIV
360 ONT%+1GOSUB500,600,600,600,600,800,800,800,500,900,
    600,600,800
370 POKEOF+AD,A
380 AD=AD+A%:IFLEFT$(X$,2)="*="THEN400
390 LX=AD
400 REM         ************  UITVOER
410 IFF%=0THENIFFL$(0)=" "ANDFL$(1)=" "ANDFL$(2)=" "THEN430
420 BS%=BS%+1
430 ONPMGOTO250
440 Y$=LEFT$(Y$+"            ",11):FORI=1TO3:
    PRINT#PF,FL$(I);:NEXT
450 PRINT#PF,PR$ZN$LN$"  "LEFT$(X$+"     ",6)Y$" "RM$
460 GOTO250
500 REM 1-BYTE-INSTRUC.
510 A%=1:A=T%(J,T%):IFA<0THENFL$(2)="A":GOTO1510
520 GOSUB3240:PR$=PR$+RIGHT$(A$,2)+"         ":RETURN
600 REM 2-BYTE-INSTRUCT.
610 A=T%(J,T%):IFA<0THENFL$(2)="A":GOTO1500
620 GOSUB3240:PR$=PR$+RIGHT$(A$,2)
630 YY$=YA$:IFLEFT$(YY$,1)="#"THENYY$=MID$(YY$,2)
640 IFLEFT$(YY$,1)="*"THENYY$=MID$(YY$,2)
650 A%=2:IFLEFT$(YY$,1)=">"ORLEFT$(YY$,1)="<"THEN
    YY$=MID$(YY$,2)
660 A$=LEFT$(YY$,1):IFA$="$"ORA$>"/"ANDA$<":"THEN
    A$=YY$:GOTO690
670 SL$=YY$:GOSUB4500
680 A$="$"+HE$
690 GOSUB3100
700 IFLEFT$(YA$,2)="#>"THENA=INT(A/HI)
710 IFLEFT$(YA$,2)="#<"THENA=A-INT(A/HI)*HI
720 IFA>LOTHENFL$(2)="0":F%=1:A=0
730 GOSUB3240:POKEOF+AD+1,AL%:PR$=PR$+" "+RIGHT$
    ("00"+A$,2)+"     "
740 A=T%(J,T%):RETURN
800 REM 3-BYTE-INSTRUC.
810 A%=3
820 A=T%(J,T%)
830 GOSUB3240:PR$=PR$+RIGHT$(A$,2)
```

```
840 A$=LEFT$(YA$,1):IFA$="$"ORA$>"/"ANDA$<":"THEN
    A$=YA$:GOTO870
850 SL$=YA$:GOSUB4500
860 A$="$"+HE$
870 GOSUB3100:GOSUB3240:PR$=PR$+" "+RIGHT$("00"+A$,2)+
    " "+LEFT$(A$,2)+"  "
880 POKEOF+AD+1,AL%:POKEOF+AD+2,AH%
890 A=T%(J,T%):RETURN
900 REM RELATIV
910 A%=2
920 A=T%(J,T%):GOSUB3240:PR$=PR$+RIGHT$(A$,2)
930 A$=LEFT$(Y$,1):IFA$="$"ORA$>"/"ANDA$<":"THENA$=Y$:
    GOTO960
940 SL$=Y$:GOSUB4500
950 A$="$"+HE$
960 GOSUB3100:IFFL$(2)="U"THENA=AD+2
970 DF=A-(AD+2):IFDF<-128ORDF>127THENFL$(3)="R":F%=1:
    DF=0
980 A=DFANDLO:GOSUB3240
990 PR$=PR$+" "+RIGHT$(A$,2)+"      ":POKEOF+AD+1,A:A=
    t%(J,t%):RETURN
1000 PR$="                ":IFF%=0THEN1020
1010 BS%=BS%+1
1020 IFAE<AD+OFTHENAE=AD+OF
1030 ONPMGOTO1060
1040 FORI=0TO3:PRINT#PF,FL$(I);:NEXT
1050 PRINT#PF,PR$ZN$LN$"  "LEFT$(X$+"     ",6)Y$" "RM$
1060 CLOSE8:INPUT"NOTITIE       J/N      ";A$:
     IFA$<>"N"THEN1130
1070 A$=DD$+":"+SN$+".ObJ"
1080 A%=LEN(A$):POKE183,A%:POKE187,351ANDLO:POKE188,351/HI
1090 FORI=1TOA%:POKE350+I,ASc(MID$(A$,I)):NEXT:REM FILENAAM
1100 A=SA:GOSUB3240:POKE251,AL%:POKE252,AH%:REM STARTADRES
1110 A=AE:GOSUB3240:POKE781,AL%:POKE782,AH%:REM EINDADRES
1120 POKE780,251:SYS65496:REM SAVE
1130 A=PA:GOSUB3240:PA$=A$:A=AD:GOSUB3240:AD$=A$:A=AD-PA:
     GOSUB3240
1140 BA$=A$:ONPMGOTO1180
1150 PRINT#PF:PRINT#PF,PA$" / "AD$" / "BA$
1160 PRINT#PF,"SOURCEFILE IS  "SN$+".SRC"
1170 PRINT#PF,BS%" FOUT  ":PRINT#PF
1180 INPUT"SYMBOOLTABEL  J/N ";Z$:IFZ$<>"J"THEN1400
1190 MX=2:IFPG=4THENPRINT#PF,CHR$(12):MX=5
1200 INPUT"GESORTEERD J/N";Z$:IFZ$="J"THEN1300
1210 ONPMGOTO1220
1220 M%=0:P$="":FORI=LL%TOUL%
1230 IFLB$(I)="      "THEN1290
1240 P$=P$+LB$(I)+"  "+HE$(I)+"      ":M%=M%+1
1250 IFM%<>MXTHEN1290
```

```
1260 ONPMGOTO1280
1270 PRINT#PF,P$
1280 P$="":M%=0:IFI>=UL%THEN1400
1290 NEXTI:IFP$<>""THEN1260
1300 HI$=CHR$(127)+CHR$(127)+CHR$(127)+CHR$(127)+CHR$(127):
     F%=0:REM SORT
1310 M%=0:SL$=HI$:FORI=LL%TOUL%:IFLB$(I)="      "THEN1340
1320 IFLB$(I)<SL$THENSL$=LB$(I):M%=I+1
1330 UL%=I
1340 NEXTI:IFF%<MXTHEN1360
1350 F%=0:IFPM=0THENPRINT#PF
1360 IFM%=0THEN1400
1370 ONPMGOTO1390
1380 PRINT#PF,SL$"  "HE$(M%-1)"      ";
1390 LB$(M%-1)="      ":F%=F%+1:GOTO1310
1400 REM
1410 IFPG=4THENPRINT#PF,CHR$(12)
1420 CLOSEPF:END
1500 POKEOF+AD+2,0:REM NOP-VULLER
1510 POKEOF+AD+1,0
1520 A=0:PR$=PR$+NP$(A%):RETURN
1600 IFLEFT$(LN$,1)="."THENI=-1:RETURN
1610 IFMID$(LN$,4,1)<>" "THENI=NN%+1:RETURN
1620 MN$=LEFT$(LN$,3):REM LABEL = MNEMONIC ?
1630 FORI=0TONN%:IFMN$<>MN$(I)THENNEXT
1640 RETURN
2000 GET#8,A$,B$:IFA$+B$=""THEN2290:REM LINKADRES
2010 GET#8,Z1$,Z2$
2020 ZN=ASC(Z1$+CHR$(0))+HI*ASC(Z2$+CHR$(0))
2030 ZN$=RIGHT$("    "+STR$(ZN),5)+" "
2040 GOSUB2300:IFFF%THENRETURN
2050 LN$="":X$="":Y$="":RM$="":X%=0
2060 FORI=0TO3:FL$(I)=" ":NEXTI:IFZ$="*"THEN2190
2070 IFZ$=";"THEN2280
2080 REM LABELNAAM
2090 IFZ$=" "ORFF%THENLN$=LEFT$(LN$+"      ",5):GOTO2120
2100 LN$=LN$+Z$:IFLEN(LN$)=6THENX%=1:FL$(0)="L"
2110 GOSUB2300:GOTO2090
2120 GOSUB1600:IFI<=NN%THENX$=LN$:LN$="      ":GOTO2200
2130 X%=ASC(LN$):IFX%<65ORX%>90THENFL$(0)="S"
2140 REM OPERATION
2150 GOSUB2300:IFFF%THENRETURN
2160 IFZ$<>" "THEN2190
2170 GOTO2150
2180 GOSUB2300:IFFF%THENRETURN
2190 IFZ$<>" "THENX$=X$+Z$:GOTO2180
2200 IFFF%THENRETURN
2210 IFZ$=";"THEN2280
2220 IFZ$<>" "THEN2260:REM OPERAND
```

```
2230 GOSUB2300:IFFF%THENRETURN
2240 GOTO2200
2250 GOSUB2300:IFFF%THENRETURN
2260 IFZ$<>" "THENY$=Y$+Z$:GOTO2250
2270 GOSUB2300:IFFF%THENRETURN:REM OPMERKING
2280 RM$=RM$+Z$:GOTO2270
2290 X$=".EN":RM$="EINDE ?    ":LN$="      ":Y$="":
     ZN$="      ":RETURN
2300 GET#8,Z$:FF%=Z$="":RETURN
2400 REM AARD VAN ADRESSERING ?
2410 IFY$=""THENT%=8:RETURN:REM IMPLIC.
2420 YA$=Y$:IFLEFT$(YA$,1)="("THENYA$=MID$(YA$,2)
2430 IFRIGHT$(YA$,1)=")"THENYA$=LEFT$(YA$,LEN(YA$)-1)
2440 IFRIGHT$(YA$,3)="),Y"THENYA$=LEFT$(YA$,LEN(YA$)-3)
2450 IFRIGHT$(YA$,2)=",Y"ORRIGHT$(YA$,2)=",X"THENYA$=
     LEFT$(YA$,LEN(YA$)-2)
2460 Z$=Y$:K$=LEFT$(Y$,1)
2470 IFZ$="A"THENT%=0:RETURN:REM ACCU
2480 IFK$="#"THENT%=1:RETURN:REM IMMIDIATE
2490 IFK$="("THEN2600:REM INDIRECT
2500 ZP=K$="*":REM ZEROPAGE
2510 Z$=MID$(Y$,2+ZP)
2520 IFLEN(Z$)<2THEN2550
2530 K$=MID$(Z$,LEN(Z$)-1,1)
2540 IFK$=","THEN2570:REM INDEXED
2550 T%=5
2560 T%=T%+3*ZP:RETURN:REM ABSOLUUTBZw. ZEROPAGE
2570 K$=RIGHT$(Z$,1):IFK$="X"THENT%=6:GOTO2560
2580 IFK$="Y"THENT%=7:GOTO2560
2590 T%=-1:RETURN:SYNTAX ERROR
2600 K$=RIGHT$(Z$,1):IFK$=")"THEN2630
2610 IFRIGHT$(Z$,2)<>",Y"THEN2590
2620 T%=11:RETURN
2630 IFMID$(Z$,LEN(Z$)-2,2)=",X"THENT%=10:RETURN
2640 T%=12:RETURN
2700 IFX$="="THEN2730:REM PSEUDO-OPS PASS 1
2710 IFLEFT$(X$,2)="*="THEN2780
2720 A%=0:RETURN
2730 A%=0:IFY$="*"THENRETURN
2740 A%=ASC(LEFT$(LN$,1)):IFA%<65ORA%>90THENRETURN
2750 A$=LEFT$(Y$,1):IFA$<>"$"AND(A$<"0"ORA$>"9")THENRETURN
2760 A$=Y$:GOSUB3100:IFF%THENML$(HC%)=FL$(2):RETURN
2770 GOSUB3240:HE$(HC%)=RIGHT$("0000"+A$,4):RETURN
2780 A%=0:Y1$=LEFT$(Y$,1):IFY1$="$"ORY1$>"/"ANDY1$<":"THEN2800
2790 RETURN
2800 A$=Y$:GOSUB3100:IFF%THENRETURN
2810 HA=A:GOSUB3240:X%=ASC(LEFT$(LN$+CHR$(0),1)):
     IFX%>64ANDX%<91THENHE$(HC%)=A$
2820 RETURN
```

```
2900 IFXX$="="THEN2940:REM PSEUDO-OPS PASS 2
2910 IFLEFT$(X$,2)="*="THEN2990
2920 FL$(1)="S"
2930 A%=0:F%=1:PR$="                    ":RETURN
2940 A%=0
2950 A$=LEFT$(Y$,1)
2960 IFA$<>"*"ANDA$<>"$"AND(A$<"0"ORA$>"9")THENFL$(2)="S":
     GOTO2930
2970 SL$=LN$:F%=0:GOSUB4500:IFF%THENFL$(0)=FL$(2):FL$(2)=" ":
     GOTO2930
2980 PR$=HE$+"              ":RETURN
2990 A%=0:Y1$=LEFT$(Y$,1):IFY1$="$"ORY1$>"/"ANDY1$<":"THEN3010
2991 YZ$=LEFT$(Y$,1):LH%=YZ$=">"ORYZ$=">"ORYZ$="<":
     YA$=MID$(Y$,1-LH%)
2992 YZ$=LEFT$(YA$,1):IFYZ$="$"ORYZ$>"/"ANDYZ$<":"THENHE$=YA$:
     GOTO2994
2993 SL$=YA$:F%=0:GOSUB4500:HE$="$"+HE$:IFF%THENFL$(0)=FL$(2):
     FL$(2)=" "
2994 A$=HE$:GOSUB3100:IFA>LOANDLH%=0THENA=0:FL$(1)="0"
2995 IFLEFT$(Y$,1)=">"THENA=INT(A/HI)
2996 IFLEFT$(Y$,1)="<"THENA=A-INT(A/HI)*HI
2998 POKEAD,A:A%=1:GOSUB3240:PR$=PR$+RIGHT$("00"+A$,2)+"       ":
     RETURN
3000 FL$(2)="S":F%=1:GOTO3030
3010 A$=Y$:GOSUB3100:IFF%THEN3030
3020 AD=A:GOSUB3240:PR$=A$+"  "
3030 PR$=PR$+"           ":RETURN
3100 REM OMZETTEN HEX -> DEC  A$ -> A
3110 Z$=LEFT$(A$,1):IFZ$="$"THENA$=RIGHT$(A$,LEN(A$)-1):GOTO3150
3120 IFZ$<"0"ORZ$>"9"THENFL$(2)="S":F%=1:RETURN
3130 A=VAL(A$):IFA>65535ORA<0THENFL$(2)="0":F%=1
3140 RETURN
3150 A=0:L%=LEN(A$):IFL%>4THENF%=1:FL$(2)="L":RETURN
3200 FORI=1TOL%:AA%=ASC(MID$(A$,I))-48
3210 IFAA%<00RAA%>9THENIFAA%<17ORAA%>22THENF%=1:FL$(2)="S":RETURN
3220 IFAA%>9THENAA%=AA%-7
3230 A=A+AA%*16^(L%-I):NEXT:RETURN
3240 AH%=A/HI:AL%=A-AH%*HI:A$=A$(AH%/16)+A$(AH%AND15)+A$(AL%/16)+
     A$(AL%AND15)
3250 RETURN
4000 DIMLB$(349),HE$(349),ML$(349):HA=AD:REM ADRESBESTAND OPMAKEN
4010 FORI=0TO349:LB$(I)="     ":HE$(I)="0000":ML$(I)=" ":NEXT
4020 OP$=DD$+":"+SN$+".SRC"
4030 OPEN8,DG,0,OP$
4040 GET#8,A$,A$:LL%=349
4050 IFST<>0THENCLOSE8:END
4060 GOSUB2000:PRINTCHR$(145),ZN$:IFLN$=""ORLEFT$(LN$,1)=" "
     THEN4210
4070 X%=ASC(LEFT$(LN$,1)):IFX%<65ORX%>90THEN4210
```

```
4080 GOSUB4100:GOTO4130
4090 LN$=LEFT$(LN$+"     ",5):REM HASH-CODE VORMEN
4100 HC=0:FORI=1TO5
4110 HC%=ASC(MID$(LN$,I,1)):HC=HC+(HC%/10-INT(HC%/10))*10^(6-I):
     NEXTI
4120 HC%=(HC/307-INT(HC/307))*300:RETURN
4130 A=HA:GOSUB3240
4140 IFLB$(HC%)<>"      "THEN4180
4150 LB$(HC%)=LN$:HE$(HC%)=A$:IFHC%>UL%THENUL%=HC%
4160 IFHC%<LL%THENLL%=HC%
4170 GOTO4210
4180 IFLB$(HC%)=LN$THENML$(HC%)="M":GOTO4210
4190 HC%=HC%+1:IFHC%<350THEN4140
4200 PRINT"SYMBOOLTABEL VOL  ":CLOSE8:END
4210 IFX$=".EN"THENCLOSE8:RETURN
4220 XX$=LEFT$(X$,1):IFXX$="."ORXX$="*"ORXX$="="THENGOSUB2700:
     HA=HA+A%:GOTO4060
4230 F%=0:XX$=LEFT$(X$,3):FORJ=0TONN%:IFXX$<>MN$(J)THENNEXT:
     GOTO4270
4240 GOSUB2400
4250 IFT%(J,T%)>=0THEN4280
4260 IFT%=5ANDT%(J,9)>=0THENT%=9:GOTO4280
4270 F%=1:HA=HA+1:GOTO4060
4280 HA=HA+L%(T%):GOTO4060
4500 REM          *********  LABEL ZOEKEN
4510 X%=ASC(LEFT$(SL$,1)):IFX%<650RX%>90THENFL$(2)="S":F%=1:
     HE$="0000":RETURN
4520 IFLEN(SL$)>5THENFL$(2)="L"
4530 SV$=LN$:LN$=SL$:GOSUB4090:SL$=LN$:LN$=SV$
4540 IFLB$(HC%)="      "ORHC%>UL%THENFL$(2)="U":F%=1:HE$="0000":
     RETURN
4550 IFLB$(HC%)<>SL$THEN4580
4560 HE$=HE$(HC%):IFML$(HC%)<>" "THENFL$(2)=ML$(HC%)
4570 RETURN
4580 HC%=HC%+1:GOTO4540
4590 Y1$="":Y2$="":I=1:REM OMZETTEN VAN Y$ IN Y1$ EN Y2$
4600 IFMID$(Y$,I,1)<>","THENY1$=Y1$+MID$(Y$,I,1)
4610 IFI>LEN(Y$)THENF%=1:RETURN
4620 IFMID$(Y$,I,1)<>","THENI=I+1:GOTO4600
4630 I=I+1:IFI>LEN(Y$)THENF%=1:RETURN
4640 Y2$=Y2$+MID$(Y$,I,1):IFI=LEN(Y$)THENF%=0:RETURN
4650 I=I+1:GOTO4640
5000 READNN%:HI=256:LO=255
5010 DIMA$(15),MN$(NN%),T%(NN%,12),L%(12),FL$(3),NP$(3)
5020 FORI=0TO15:READA$(I):NEXT
5030 NP$(1)="00         ":NP$(2)="00 00      ":NP$(3)="00 00 00  "
5040 FORI=0TO12:READL%(I):NEXT
5050 FORJ=0TONN%:READMN$(J):FORJJ=0TO12:READA$:IFA$="-1"THENA=-1:
     GOTO5070
```

```
5060 A=0:FORI=1TO2:X=ASC(RIGHT$(A$,I))-48:X=X+(X>9)*7:
     A=A+X*16^(I-1):NEXT
5070 T%(J,JJ)=A:NEXT:NEXT:RETURN
6000 DATA 55 :REM AANTAL MNEMONICS
6010 DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
6020 DATA 1,2,2,2,2,3,3,3,1,2,2,2,3
7000 DATA ADC,-1,69,65,75,-1,6D,7D,79,-1,-1,61,71,-1
7010 DATA AND,-1,29,25,35,-1,2D,3D,39,-1,-1,21,31,-1
7020 DATA ASL,0A,-1,06,16,-1,0E,1E,-1,-1,-1,-1,-1,-1
7030 DATA BCC,-1,-1,-1,-1,-1,-1,-1,-1,-1,90,-1,-1,-1
7040 DATA BCS,-1,-1,-1,-1,-1,-1,-1,-1,-1,B0,-1,-1,-1
7050 DATA BEq,-1,-1,-1,-1,-1,-1,-1,-1,-1,F0,-1,-1,-1
7060 DATA BMI,-1,-1,-1,-1,-1,-1,-1,-1,-1,30,-1,-1,-1
7070 DATA BIT,-1,-1,24,-1,-1,2C,-1,-1,-1,-1,-1,-1,-1
7080 DATA BNE,-1,-1,-1,-1,-1,-1,-1,-1,-1,D0,-1,-1,-1
7090 DATA BPL,-1,-1,-1,-1,-1,-1,-1,-1,-1,10,-1,-1,-1
7100 DATA BRK,-1,-1,-1,-1,-1,-1,-1,-1,00,-1,-1,-1,-1
7110 DATA BVC,-1,-1,-1,-1,-1,-1,-1,-1,-1,50,-1,-1,-1
7120 DATA BVS,-1,-1,-1,-1,-1,-1,-1,-1,-1,70,-1,-1,-1
7130 DATA CLC,-1,-1,-1,-1,-1,-1,-1,-1,18,-1,-1,-1,-1
7140 DATA CLD,-1,-1,-1,-1,-1,-1,-1,-1,D8,-1,-1,-1,-1
7150 DATA CLI,-1,-1,-1,-1,-1,-1,-1,-1,58,-1,-1,-1,-1
7160 DATA CLV,-1,-1,-1,-1,-1,-1,-1,-1,B8,-1,-1,-1,-1
7170 DATA CMP,-1,C9,C5,D5,-1,CD,DD,D9,-1,-1,C1,D1,-1
7180 DATA CPX,-1,E0,E4,-1,-1,EC,-1,-1,-1,-1,-1,-1,-1
7190 DATA CPY,-1,C0,C4,-1,-1,CC,-1,-1,-1,-1,-1,-1,-1
7200 DATA DEC,-1,-1,C6,D6,-1,CE,DE,-1,-1,-1,-1,-1,-1
7210 DATA DEX,-1,-1,-1,-1,-1,-1,-1,-1,CA,-1,-1,-1,-1
7220 DATA DEY,-1,-1,-1,-1,-1,-1,-1,-1,88,-1,-1,-1,-1
7230 DATA EOR,-1,49,45,55,-1,4D,5D,59,-1,-1,41,51,-1
7240 DATA INC,-1,-1,E6,F6,-1,EE,FE,-1,-1,-1,-1,-1,-1
7250 DATA INX,-1,-1,-1,-1,-1,-1,-1,-1,E8,-1,-1,-1,-1
7260 DATA INY,-1,-1,-1,-1,-1,-1,-1,-1,C8,-1,-1,-1,-1
7270 DATA JMP,-1,-1,-1,-1,-1,4C,-1,-1,-1,-1,-1,-1,6C
7280 DATA JSR,-1,-1,-1,-1,-1,20,-1,-1,-1,-1,-1,-1,-1
7290 DATA LDA,-1,A9,A5,B5,-1,AD,BD,B9,-1,-1,A1,B1,-1
7300 DATA LDX,-1,A2,A6,-1,B6,AE,-1,BE,-1,-1,-1,-1,-1
7310 DATA LDY,-1,A0,A4,B4,-1,AC,BC,-1,-1,-1,-1,-1,-1
7320 DATA LSR,4A,-1,46,56,-1,4E,5E,-1,-1,-1,-1,-1,-1
7330 DATA NOP,-1,-1,-1,-1,-1,-1,-1,-1,EA,-1,-1,-1,-1
7340 DATA ORA,-1,09,05,15,-1,0D,1D,19,-1,-1,01,11,-1
7350 DATA PHA,-1,-1,-1,-1,-1,-1,-1,-1,48,-1,-1,-1,-1
7360 DATA PHP,-1,-1,-1,-1,-1,-1,-1,-1,08,-1,-1,-1,-1
7370 DATA PLA,-1,-1,-1,-1,-1,-1,-1,-1,68,-1,-1,-1,-1
7380 DATA PLP,-1,-1,-1,-1,-1,-1,-1,-1,28,-1,-1,-1,-1
7390 DATA ROL,2A,-1,26,36,-1,2E,3E,-1,-1,-1,-1,-1,-1
7400 DATA ROR,6A,-1,66,76,-1,6E,7E,-1,-1,-1,-1,-1,-1
7410 DATA RTI,-1,-1,-1,-1,-1,-1,-1,-1,40,-1,-1,-1,-1
7420 DATA RTS,-1,-1,-1,-1,-1,-1,-1,-1,60,-1,-1,-1,-1
```

```
7430 DATA SBC,-1,E9,E5,F5,-1,ED,FD,F9,-1,-1,E1,F1,-1
7440 DATA SEC,-1,-1,-1,-1,-1,-1,-1,-1,38,-1,-1,-1,-1
7450 DATA SED,-1,-1,-1,-1,-1,-1,-1,-1,F8,-1,-1,-1,-1
7460 DATA SEI,-1,-1,-1,-1,-1,-1,-1,-1,78,-1,-1,-1,-1
7470 DATA STA,-1,-1,85,95,-1,8D,9D,99,-1,-1,81,91,-1
7480 DATA STX,-1,-1,86,-1,96,8E,-1,-1,-1,-1,-1,-1,-1
7490 DATA STY,-1,-1,84,94,-1,8C,-1,-1,-1,-1,-1,-1,-1
7500 DATA TAX,-1,-1,-1,-1,-1,-1,-1,-1,AA,-1,-1,-1,-1
7510 DATA TAY,-1,-1,-1,-1,-1,-1,-1,-1,A8,-1,-1,-1,-1
7520 DATA TSX,-1,-1,-1,-1,-1,-1,-1,-1,BA,-1,-1,-1,-1
7530 DATA TXA,-1,-1,-1,-1,-1,-1,-1,-1,8A,-1,-1,-1,-1
7540 DATA TXS,-1,-1,-1,-1,-1,-1,-1,-1,9A,-1,-1,-1,-1
7550 DATA TYA,-1,-1,-1,-1,-1,-1,-1,-1,98,-1,-1,-1,-1
```

5.1 Beschrijving van de 6510-assembler

100-190

Uitvoer van de titel, invoer van het sourceprogramma onder de naam SN$. Gevraagd wordt of een assemblerlisting uitgevoerd moet worden en zo ja of dat op de printer (devicenummer PG=4) of op het scherm (PG=3) moet gebeuren. De PM-flag beslist of een listing al dan niet uitgevoerd wordt. Aansluitend wordt gesprongen naar subroutine 5000. In deze subroutine worden de velden voor de variabelen in het geheugen vrij gemaakt (DIMensionering). Tevens worden de volledige lijst mnemonics (symbolen voor de machinetaalinstructies) en de codes die behoren bij de verschillende adresseermanieren, ingelezen.

200-460

Hoofdlus van het programma. In regel 210 wordt pass-1 als subroutine van het hoofdprogramma uitgevoerd (GOSUB 4000). Het sourceprogramma wordt ingelezen en onderzocht op labels. Na pass-1 wordt het programma opnieuw ingelezen van de diskette. Als geen listing wordt gewenst worden in regel 250 alleen de regelnummers ZN$ uitgevoerd. Aansluitend worden aan het einde de instructies X$=".EN" en de pseudo-instructies onderzocht en apart behandeld. De variabele PR$ (afdrukstring), die het geassembleerde deel bevat, wordt gevormd. In regel 320 wordt getest of de instructie (bv. XX$=LDA) in het bestand als een van de 56 geldige instructies voorkomt. Als dit niet het geval is, wordt de foutflag gezet en voor de instructiecode een nul geplaatst. 00 is de code voor BRK. In regel 350 wordt gesprongen

naar GOSUB 2400 om de wijze van adresseren te bepalen. (T% geeft de adresseermanieren 0t/m 12 aan). Bij de relatieve adressering wordt net zo gecorrigeerd. In regel 360 wordt, afhankelijk van de adresseerwijze gesprongen naar de subroutines 500 (1-byte), 600 (2-bytes), 800 (3-bytes) of 900 (relative). In regel 5070 (van subroutine 5000) zijn voor 13*56 instructiecodes A$ (hexadecimaal) de decimale getallen A=T%(J, JJ) bepaald. De waarde van A wordt in de subroutines 500, 600, 800 en 900 bepaald en in regel 370 in het geheugen gezet op adres OF+AD. In regel 380 wordt de instructieteller opgehoogd. De regels 400 t/m 460 zorgen voor de printopdrachten. Bij een fout wordt de teller BS% in regel 420 verhoogd. In regel 450 tenslotte wordt de volledige assemblerregel uitgevoerd.

500-520

Behandeling van de 1-byte instructies. De variabele A% krijgt als waarde het aantal bytes en met hulp van het veld T% wordt de instructiecode vastgesteld. Een waarde kleiner dan nul betekent dat deze instructie de betrokken adresseerwijze niet heeft. In dit geval wordt naar 1510 gesprongen, waar een nulbyte geschreven wordt. Is de waarde groter dan nul, dan wordt de Op-code die in het array T% decimaal was weggezet, weer in de hexadecimale code omgezet. In de afdrukstring PR$ wordt dit ingebouwd.

600-740

Behandeling van de 2-bytes instructies. In regel 610 wordt weer de Op-code vastgesteld, die in regel 620 in de afdrukstring PR$ wordt ingebouwd. Daarna wordt getest op de immediate adresseerwijze, aangegeven met #, en op de operatoren > en <. Eveneens wordt herkend of het gaat om de zeropage adresseerwijze, aangegeven met *. Regel 660 stelt vast of het om een numerieke (hexadecimaal of decimaal) waarde, dan wel om een alfanumerieke(letters en/of cijfers) waarde gaat. In het laatste geval betreft het een labelnaam. In regel 670 wordt de waarde van de labelnaam bepaald (subroutine 4500). De waarde van de operand wordt in de hexadecimale schrijfwijze omgezet. Regels 700 en 710 wijzigen de waarde overeenkomstig de operatoren < en >. Regel 720 test tenslotte of de waarde niet groter dan 255 is. Nu kan de waarde in de afdrukstring en in het geheugen gezet worden.

800-890

De behandeling van de 3-bytes instructies gaat analoog aan die van de 2-bytes instructies. De waarde van de operand wordt nu in de volgorde lo-byte, hi-byte in het geheugen gezet (gePOKEd) en in de afdrukstring ingebouwd.

900-990

Bij de relatieve adresseerwijze wordt eerst de waarde van de

operand bepaald. In regel 970 wordt de waarde van de sprong bepaald. Tevens wordt getest of het adres binnen het geheugengebied van de Commodore 64 ligt. Als dat niet het geval is, wordt dat door een fout R(range) aangegeven en sprong op nul gezet. In regel 980 wordt een AND-bewerking met 255 uitgevoerd om de negatieve waarde met behulp van het two's complement te bepalen. Nu wordt de waarde in het geheugen gezet en in de afdrukstring ingebouwd.

1000-1420

Hier wordt naartoe gesprongen als de assemblergang is afgelopen. De laatste regel wordt uitgevoerd. Gevraagd wordt of de geproduceerde codes op diskette moeten worden gezet. Als dit het geval is worden de filenaam en begin- en eindadres op de betreffende geheugenplaatsen weggezet en de SAVE-routine van het operating system met SYS opgeroepen. Het geheugengebied en het aantal gebruikte geheugenplaatsen worden uitgevoerd. U kunt ook bepalen of u een al dan niet gealfabetiseerde labeltabel wilt hebben. Dit gebeurt in de regels 1200-1400. Daarmee is de assemblergang beëindigd.

5.2 Gebruikte subroutines

Nu volgen de subroutines die door het bovenstaande programma worden aangeroepen en die bijvoorbeeld getallen omzetten.

1500-1520

Deze routine plaatst een of twee nulbytes als een fout bij het assembleren wordt ontdekt.

1600-1640

Deze subroutine test of een labelnaam overeenkomt met een instructie. Als dit het geval is krijgt de variabele I de waarde tussen 0 en NN% (0 en 55) die hoort bij de betrokken instructie. Hiermee wordt beslist of het gaat om een labelnaam dan wel om een instructie.

2000-2300

Deze routine wordt gebruikt om een programmaregel van de diskette te lezen en de variabelen voor regelnummer ZN$, label LN$, instructie X$, operand Y$ en commentaar RM$ de overeenkomstige waarde toe te kennen. De routine 2300 haalt daarvoor een byte van

de diskette in de variabele Z$. De flag FF% wordt gezet als het een 0-byte betreft (herkenning van het einde van een regel). De beide eerste bytes, die het schakeladres bevatten, worden niet gebruikt. Zijn beide echter nul, dan is het programma-einde bereikt en wordt een .EN-aanduiding gemaakt. Anders wordt uit de volgende twee bytes het regelnummer gehaald. De karakters die nu volgen worden opgeteld in een variabele, totdat een spatie of het regeleinde is bereikt. Als een puntkomma wordt gevonden, wordt het volgende commentaar in de REMark-string RM$ gezet.

2400-2640

Deze routine deelt de adresseerwijze van een instructie mee. Daarbij wordt getest of het gaat om een linker haakje (een rechter haakje) een komma, een X of een Y. De direkte adresseerwijze (immediate) wordt herkend aan het #-teken en de verwijzing naar het zeropage-adres aan * (variabele ZP, regel 2510). Het resultaat van deze routine is de adresseerwijze in de variabele T%, die een waarde van 0 t/m 12 toegekend krijgt. In subroutine 5000 was het gehele veld met in totaal T%(55, 12), dus 56*13 mogelijke instructies met adresseermethodes vastgelegd. Een negatieve waarde geeft aan dat de betrokken adresseerwijze niet beschikbaar is.

2700-2820

In deze routine worden de pseudo-instructies =, *= en BY verwerkt. De subroutine wordt in pass-1 opgeroepen en bijvoorbeeld voor het definiëren van labels gebruikt.

2900-2998

Deze routine behandelt dezelfde instructies als de vorige, maar voor pass-2. Deze routine zet bijvoorbeeld de code van de .BY-instructie in het geheugen evenals alle instructies die de afdrukstring PR$ samenstellen.

3000-3030

Deze routine roept de volgende routines op voor de omzetting van getallen en wordt gebruikt om aan het begin van de afdrukstring de stand van de instructieteller te zetten.

3100-3230

Een hexadecimaal getal in A$ wordt omgezet in een decimaal getal A.

3240-3250

Uit een decimaal getal A wordt het hexadecimale getal A$ gevormd. Na het oproepen van deze routine staan in AL% en AH% nog de waarde van lo-byte en hi-byte ter beschikking.

4000-4280

Dit deelprogramma voert de volledige pass-1 uit. De labels worden voornamelijk gezocht; hun waarden worden toegekend. Van deze routine worden ook de lopende regelnummers uitgevoerd (regel 4060). Om de labels later sneller te kunnen terugvinden worden ze voorzien van een deelcode (hashcode-procedure) en weggezet in veld LB$(). De hexadecimale waarde wordt in veld HE$() gezet. Uit de wijze van adresseren moet de lengte van de bijbehorende instructie meegedeeld worden, zodat aan het label de correcte waarde kan worden toegekend. Ook moet op dubbele labels getest worden. Het veld L%() bevat voor iedere adresseerwijze de overeenkomstige lengte. De instructieteller kan nu voldoende worden verhoogd.

4500-4650

Deze routine wordt in pass-2 opgeroepen en geeft de waarde van een label dat in SL$ wordt gezet. Als het label niet wordt gevonden, wordt een foutflag gezet. Wordt het label wel gevonden dan wordt in HE$ de hexadecimale waarde gezet.

5000-5070

In deze routine worden de waarden van de variabelen die in de DATA-statements staan in het geheugen gezet.

6000-7550

Hier staan alle DATA. In regel 6000 het aantal instructies, in regel 6010 de hexadecimale getallen die nodig zijn voor de omzettingen, in regel 6020 het aantal bytes vooor elk van de 13 adresseermanieren en tenslotte vanaf regel 7000 de volledige instructieset met de bijborende codes.

5.3 Betekenis van de belangrijkste variabelen

SN$

Deze variabele bevat de naam van het programma. Het achtervoegsel .SRC wordt weggelaten. Het door de assembler gemaakte machinetaalprogramma wordt onder dezelfde naam, maar met het achtervoegsel .OBJ weggezet.

DD$

Drivenummer

PG

Nummer van het randapparaat. Het beeldscherm is 3 en de printer 4.

PM

Er vindt geen uitvoer op de printer plaats als de flag gezet is.

A

Waarde van het actuele adres.

AD

Actuele stand van de instructieteller tijdens het assembleren.

ZN$

Lopend regelnummer.

LN$

Labelnaam.

X$

Instructiewoord.

Y$

Operand.

RM$

Commentaar.

PR$

Afdrukstring, volledige regel, geassembleerde deel eerst en dan de regel van het sourceprogramma.

TZ

Adresseerwijze.

OF

Sprong bij het wegzetten van de gevonden code in het geheugen, indien niet gebruikt is OF=0.

AZ

Lengte van de instructie.

A$

Hexadecimale schrijfwijze van actueel adres A.

SL$
Zoeklabel, moet bij de oproep van subroutine 4500 de naam van het te zoeken label bevatten.

HE$
Hexadecimale voorstelling van het label.

LO
Constante 255.

HI
Constante 256.

DF
Adresverschil bij de relatieve adressering.

BS%
Teller voor fouten in de statements.

MX
Aantal labels per regel voor de uitvoer van de labelnamentabel.

HI$
Grootste labelnaam bij de sortering op alfabet.

MN$
Mnemonic (instructiewoord).

Z$
Karakter van de diskette.

NN%
Aantal mnemonics.

X%
ASCII-code.

ZP
Flag voor de adressering op zeropage.

HA
Waarde van een label (in pass-1).

F%, FF%
Foutflags.

HC, HC%
Deelcodes (hashcodes).

FL$(3)
Foutcodes.

LB$(349)
Tabel van de labels.

HE$(349)
Tabel van de bijbehorende hexadecimale codes van de labels.

T%(55, 12)
Tabel van de Op-codes en de adresseermethodes. De eerste index geeft het aantal instructies aan, de tweede de adresseerwijze.

MN$(55)
Tabel van de instructiewoorden in alfabetische volgorde.

6 DE STAP-VOOR-STAP SIMULATOR VOOR DE 6510

In dit hoofdstuk presenteren we u een programma, dat zowel bij het testen van uw eigen programma's als bij het opsporen van fouten zeer nuttig kan zijn. Dit BASIC-programma simuleert de werking van de 6510-processor. Als u dit programma met RUN start, veschijnen naast de registeraanduidingen ook de registerinhouden op het scherm in de volgende vorm:

```
  PC    AC XR YR SR SP  NV-BDIZC

0000    00 00 00 20 FF  00100000
```

De betekenis van de aanduidingen kent u al:

PC	Instructieteller of program-counter
AC	Accumulator
XR	X-Indexregister
YR	Y-Indexregister
SR	Statusregister
SP	Stackpointer

Het statusregister is uitgesplitst:

N	Negativeflag
V	Overflowflag
-	Niet gebruikt, altijd 1
B	Breakflag
D	Decimalflag
I	Interruptflag
Z	Zeroflag
C	Carryflag

Onder de aanduidingen staan de inhouden van de registers. Bovenstaande waarden zijn de beginwaarden bij het starten van het programma. Door een bepaalde toets in te drukken kunt u de inhoud van een register veranderen. De cursor licht op en u kunt de oude waarde van het register met een geldig hexadecimaal getal veranderen. Daarna moet op de RETURN-toets gedrukt worden om de verandering door te voeren, de nieuwe waarde wordt in de oude regel gezet en de invoerregel wordt gewist. De registers en de flags kunnen door het indrukken van de volgende lettertoetsen worden veranderd:

P - Instructieteller

De oude inhoud van de instructieteller verschijnt. Na de vervanging door een nieuwe waarde wordt deze in het register gezet. De instructie die op deze plaats staat wordt omgezet van machinetaal naar assembleertaal (disassembleren). De registers die op de instructie betrekking hebben veranderen mee.

A-X-Y - Registers

De inhoud van de registers wordt getoond en kan worden veranderd. Na het indrukken van RETURN verschijnt de nieuwe waarde onder de registeraanduiding.

S - Stackpointer

De inhoud van de stackpointer verschijnt in de invoerregel. Na de vervanging door een nieuwe waarde en een druk op RETURN verschijnt de nieuwe waarde van de stackpointer onder de registeraanduiding. Het statusregister kan niet direkt worden veranderd. Dit kan wel door de aparte flags te veranderen. Overeenkomstig de stand van de flags wordt de waarde van het statusregister bepaald en automatisch onder de aanduiding SR gezet. De inhouden van de bits worden omgekeerd door het indrukken van een lettertoets:

N - Negative

V - Overflow

B - Break

D - Decimal

I - Interrupt

Z - Zero

C - Carry

Als bijvoorbeeld op de Z-toets wordt gedrukt, verandert de inhoud van de Z-bit van het statusregister, de waarde van het statusregister verandert mee. De belangrijkste functie van de simulator kunt u met de spatiebalk bewerkstelligen. Door het indrukken van deze toets wordt de instructie op het adres waarnaar de instructieteller wijst, in gedisassembleerde vorm onder de registeraanduidingen gezet. Zoals de naam simulator al zegt, wordt deze instructie niet direkt door de processor uitgevoerd maar wordt via BASIC nagegaan wat het gevolg is. Tevens worden de registerinhouden en de flags op dezelfde manier beïnvloed als ook door de processor zelf zou worden gedaan. Na het indrukken van de

spatiebalk veranderen dienovereenkomstig de registerinhouden en de flags. Vervolgens ziet u onder de registeraanduidingen de volgende instructie in gedisassembleerde vorm.

We bekijken een concreet voorbeeld: een programmadeel uit het operating system. De instructieteller PC wordt op adres $A81D gezet. Op het scherm staat:

```
   PC     AC XR YR SR SP  NV-BDIZC
  A81D    00 00 00 20 FF  00100000

A81D 38        SEC
```

Als nu op de spatiebalk wordt gedrukt om de instructie te simuleren krijgen we:

```
   PC     AC XR YR SR SP  NV-BDIZC
  A81E    00 00 00 21 FF  00100001

A81E A5 2B     LDA $2B
```

De instructie SEC (set carry) wordt uitgevoerd, de carryflag wordt gezet en het statusregister krijgt de inhoud $21. De instructieteller wordt met 1 verhoogd naar $A81E en de inhoud van dit adres en het volgende wordt getoond: A5 en 2B. De inhoud van adres $2B op de zeropage wordt in de accumulator gezet. Door het drukken op de spatiebalk wordt de instructie gesimuleerd:

```
   PC     AC XR YR SR SP  NV-BDIZC
  A820    01 00 00 21 FF  00100001

A820 E9 01     SBC #$01
```

De accumulator wordt met de inhoud van $2B geladen, die de waarde $01 bevat. Let op dat de Z- en de N-flag niet gezet worden, omdat de geladen waarde ongelijk is aan nul en niet groter dan $7F. De instructieteller staat nu op $A820, twee plaatsen verder; daar staat de instructie SBC (subtract with carry). Van de inhoud van de accumulator moet direkt (immediate) $01 afgetrokken worden. De geïnverteerde waarde van de carry-flag is nul. Van de accumulator wordt dus 1 afgetrokken.

```
   PC     AC XR YR SR SP  NV-BDIZC
  A822    00 00 00 23 FF  00100011

A822 A4 2C     LDY $2C
```

De accumulator heeft nu de waarde 0 gekregen, wat betekent dat de zeroflag gezet wordt. De carryflag is gezet. Daaraan ziet u dat

bij de aftrekking geen tekort is opgetreden. De volgende instructie op adres $A822 is LDY $2C. Na de uitvoering is de situatie:

```
   PC     AC XR YR SR SP  NV-BDIZC
 A824     00 00 08 21 FF  00100001

A824 B0 01      BCS $A827
```

Het Y-register krijgt de waarde $08, de inhoud van het zeropage-adres $2C. Het resultaat is ongelijk nul, de zeroflag wordt gewist. De instructieteller gaat naar adres $A824, er moet gesprongen worden met sprong 1 indien de carryflag gezet is (branch on carry set) Daar de carrybit de waarde 1 heeft is voldaan aan de voorwaarde en wordt gesprongen naar $A827:

```
   PC     AC XR YR SR SP  NV-BDIZC
 A827     00 00 08 21 FF  00100001

A827 85 41      STA $41
```

Er is dus gesprongen naar $A827, door de spronginstructie is aan de stand van de flags niets veranderd. De volgende instructie zet nu de inhoud van de accumulator in het zeropage-adres $41.

```
   PC     AC XR YR SR SP  NV-BDIZC
 A829     00 00 08 21 FF  00100001

A829 84 42      STY $42
```

Bij de STA-instructie zijn de flags niet veranderd. Ook de volgende instructie verandert niets aan de flags.

```
   PC     AC XR YR SR SP  NV-BDIZC
 A82B     00 00 08 21 FF  00100001

A82B 60         RTS
```

Hier wordt de simulatie afgebroken. Het grote voordeel van deze simulatie is dat u precies kunt zien wat er bij iedere instructie gebeurt. U kunt voor de uitvoering van iedere instructie een willekeurige register- of flaginhoud veranderen om te zien hoe de processor daarop reageert. U kunt natuurlijk ook na de uitvoering van de instructie de inhoud van de instructieteller veranderen en bijvoorbeeld weer terugzetten op de laatste instructie. Die kunt u met een veranderde flag -of registerinhoud weer opnieuw uitvoeren tot de instructie zo werkt als u wilt. Buiten de al vermelde commando's voor de stap-voor-stap simulator staat u nog een aantal mogelijkheden ter beschikking.

Wilt u een instructie niet uitvoeren, maar alleen de instructieteller op de volgende instructie zetten, dan kunt u dit met de cursor down-toets doen. Want tijdens de simulatie kunnen instructies voorkomen zoals STA of INC, die belangrijke variabelen van het operating system veranderen. Gebeurt dit niet geheel volgens plan, dan kan het gebeuren dat u de controle over de computer verliest. Daarom worden instructies die de geheugeninhoud ergens veranderen niet uitgevoerd. Is het voor het volgen van het programma gewenst dat de instructie toch wordt uitgevoerd (bv. als een geheugenplaats als teller wordt gebruikt), dan kunt u aangeven dat deze instructies daadwerkelijk uitgevoerd moeten worden. Als u op de E-toets drukt verschijnt de regel: ECHTE SIMULATIE ? J op het scherm. Drukt u nu op RETURN dan worden met onmiddellijke ingang alle instructies waarmee geheugeninhouden worden veranderd ook uitgevoerd. Verandert u de J in een N, dan schakelt u deze optie weer uit.

Door het indrukken van de letter M hebt u nog een ander commando tot uw beschikking. Met dit commando kunt u de inhoud van een bepaalde geheugenplaats bekijken en zo nodig veranderen. De veranderingen worden alleen dan uitgevoerd als, voorafgaand aan de M-toets, de E-toets voor de echte simulatie is ingedrukt.

Het volgende voorbeeld moet de werking van de stack verduidelijken. Daarvoor willen we een BREAK-instructie uitvoeren. Eerst wordt gekozen voor de echte simulatie zodat de inhoud van een geheugenplaats op 00, code voor de BREAK-instructie, kan worden gezet. Start u de simulator met RUN, druk de E-toets in zet dan de instructieteller op $0002. U krijgt dan het volgende plaatje:

```
   PC     AC XR YR SR SP  NV-BDIZC
 0002     00 00 00 20 FF  00100000

0002 00        BRK
```

Om de gang van zaken later beter te kunnen volgen, worden in de accumulator en het X- en Y-register verschillende waarden gezet:

```
   PC     AC XR YR SR SP  NV-BDIZC
 0002     22 44 88 20 FF  00100000

0002 00        BRK
```

Als op adres $0002 niet de BRK-instructie staat, drukt u op de M-toets en verandert de waarde van adres $0002 in $00. Daarmee wordt de BRK-instructie aangegeven. Als u nu op de spatiebalk drukt ontstaat:

```
   PC     AC XR YR SR SP  NV-BDIZC
 FF48     22 44 88 34 FC  00110100

FF48 48        PHA
```

De B- en de I-flag zijn gezet. De stackpointer werd met 3 verminderd doordat de instructieteller, 2 bytes, en het statusregister op de stack werden gezet. De instructieteller wordt op de stand gezet die wordt aangegeven met de inhoud van de adressen $FFFE en $FFFF. Deze inhoud wijst naar $FF48. Op dit adres staat de instructie PHA die de inhoud van de accumulator op de stack zet.

```
   PC     AC XR YR SR SP  NV-BDIZC
 FF49     22 44 88 34 FB  00110100

FF49 8A        TXA
```

De inhoud van de accumulator wordt op de stack gezet en de stackpointer wordt met 1 verminderd. Met TXA wordt de inhoud van X in A gezet:

```
   PC     AC XR YR SR SP  NV-BDIZC
 FF4A     44 44 88 34 FB  00110100

FF4A 48        PHA
```

De flags blijven onveranderd omdat in de accu geen nul of een waarde groter dan $7F staat. Met PHA wordt de accumulatorinhoud weer op de stack gezet.

```
   PC     AC XR YR SR SP  NV-BDIZC
 FF4B     44 44 88 34 FA  00110100

FF4B 98        TYA
```

De stackpointer wordt met 1 verminderd en de inhoud van register Y in de accumulator gezet. De N-flag wordt gezet omdat de waarde groter dan $7F is.

```
   PC     AC XR YR SR SP  NV-BDIZC
 FF4C     88 44 88 B4 FA  10110100

FF4C 48        PHA
```

De accumulatorinhoud wordt op de stack gezet. Deze procedure heeft als doel alle registerinhouden op de stack te bewaren, zodat reconstructie van de oorspronkelijke inhoud kan plaatsvinden als met de RTI-instructie naar het hoofdprogramma wordt

teruggesprongen.

```
   PC      AC XR YR SR SP  NV-BDIZC
  FF4D     88 44 88 B4 F9  10110100

FF4D BA         TSX
```

Nu springen we naar de plaats waar de registerinhouden weer worden opgehaald. Eerst zetten we alle registerinhouden willekeurig op nul om het proces te verduidelijken. Zet u de instructieteller op $EA81.

```
   PC      AC XR YR SR SP  NV-BDIZC
  EA81     00 00 00 B4 F9  10110100

EA81 68         PLA
```

De waarde van de stack wordt opgehaald en in de accumulator gezet:

```
   PC      AC XR YR SR SP  NV-BDIZC
  EA82     88 00 00 B4 FA  10110100

EA82 A8         TAY
```

Deze waarde wordt nu in het Y-register gezet:

```
   PC      AC XR YR SR SP  NV-BDIZC
  EA83     88 00 88 B4 FA  10-10100

EA83 68         PLA
```

De volgende waarde wordt van de stack gehaald.

```
   PC      AC XR YR SR SP  NV-BDIZC
  EA84     44 00 88 34 FB  00110100

EA84 AA         TAX
```

Iedere keer als een waarde van de stack wordt gehaald, wordt de stackpointer met 1 verhoogd. Nu wordt de inhoud van register X gelijk gemaakt aan die van A.

```
   PC      AC XR YR SR SP  NV-BDIZC
  EA85     44 44 88 34 FB  00110100

EA85 68         PLA
```

De inhoud van de accumulator wordt van de stack gehaald:

```
   PC     AC XR YR SR SP  NV-BDIZC
  EA86    22 44 88 34 FC  00110100

EA86 40        RTI
```

U ziet dat alle registers weer hun oorspronkelijke inhoud hebben en dat de stackpointer ook weer de waarde heeft die hij voor de BRK-instructie had. Belangrijk is dat de registerinhouden in omgekeerde volgorde van de stack worden gehaald. Dit first in-last out principe kenmerkt deze procedure. Na de uitvoering van de RTI-instructie is de inhoud van de registers:

```
   PC     AC XR YR SR SP  NV-BDIZC
  0005    22 44 88 20 FF  00010000

0005 91 B3     STA($B3),Y
```

Het statusregister heeft weer de oorspronkelijke inhoud teruggekregen en de instructieteller is op het volgende adres gezet na de BRK-instructie.

De simulator is een prima hulpmiddel om uw programma's te testen. U kunt stap voor stap zien of de processor datgene doet met uw programma's wat u verwacht. Op het opsporen van fouten, in machinetaalprogramma's immers een groot probleem, hebt u nu grip gekregen. Speciaal de beginners, die de adresseermethodes nog niet zo precies kennen of bij het zetten van een flag nog problemen hebben, zullen dit op prijs stellen. Op de volgende bladzijden vindt u de listing van het programma; daarachter een korte beschrijving van de afzonderlijke routines en een lijst met de belangrijkste variabelen.

```
100 PRINT"▮▮▮▮▮       DE STAP-VOOR-STAP-S
IMULATOR"
110 PRINT"      -----------------------
---
120 PRINT"     ┌──────┬───────────────────────┬────
────┐
130 PRINT"     | PC  | AC XR YR SR SP  INV-
BDIZC|"
140 PRINT"     |     |                 |
     |"
150 PRINT"     └──────┴───────────────────────┴────
────┘
160 FF=255:HI=256:UL=2↑16:SC=2↑15-1:SP=F
F
170 DIMMN$(FF),OP(FF),AD(FF),SP(FF),H$(1
5)
180 FORJ=0TO15:READH$(J):NEXT
190 FORJ=0TOFF:READMN$(J),OP(J),AD(J):NE
XT
200 REM REGISTERINHOUD TONEN
210 PRINT"▮▮▮▮▮▮▮▮▮▮▮▮";
215 IFPC>=ULTHENPC=PC-UL
220 A=PC:GOSUB2290:PRINT"▮▮▮";
230 A=AC:GOSUB2320:PRINT"▮▮";
240 A=XR:GOSUB2320:PRINT"▮▮";
250 A=YR:GOSUB2320:PRINT"▮▮";
255 GOSUB900:REM SR
260 A=SR:GOSUB2320:PRINT"▮▮";
270 A=SP:GOSUB2320:PRINT"▮▮▮";
280 PRINTCHR$(48+N);
290 PRINTCHR$(48+V);
300 PRINT"1";
310 PRINTCHR$(48+B);
320 PRINTCHR$(48+D);
330 PRINTCHR$(48+I);
340 PRINTCHR$(48+Z);
350 PRINTCHR$(48+C)
360 PRINT"▮▮▮▮▮                            ▮▮▮▮▮▮▮
▮▮▮▮▮▮▮▮▮▮▮▮▮▮";
400 GETT$:IFT$=""THEN400
405 IFT$=" "THEN1100:REM SIMULATIE
410 IFT$="P"THENPRINT"PC  ";:A=PC:GOSUB2
290:INPUT"▮▮▮▮▮▮▮";A$:GOSUB2380:PC=A:GOTO1000
420 IFT$="A"THENT$="AC":A=AC:GOSUB540:AC
=A:GOTO200
430 IFT$="X"THENT$="XR":A=XR:GOSUB540:XR
=A:GOTO200
```

```
440 IFT$="Y"THENT$="YR":A=YR:GOSUB540:YR
=A:GOTO200
450 IFT$="S"THENT$="SP":A=SP:GOSUB540:SP
=A:GOTO200
460 IFT$="N"THENN=1-N:GOTO200
470 IFT$="V"THENV=1-V:GOTO200
480 IFT$="B"THENB=1-B:GOTO200
490 IFT$="D"THEND=1-D:GOTO200
500 IFT$="I"THENI=1-I:GOTO200
510 IFT$="Z"THENZ=1-Z:GOTO200
520 IFT$="C"THENC=1-C:GOTO200
525 IFT$="M"THENS=P:E=P:PC=P:GOTO1010
527 IFT$="M"THEN3000
528 IFT$="E"THEN3100
530 GOTO400
540 PRINTT$"   ";:GOSUB2320:INPUT"■■■■";A
$:GOTO2380
900 SR=N*128+V*64+32+B*16+D*8+I*4+Z*2+C:
RETURN
910 N=SGN(SRAND128):V=SGN(SRAND64):B=SGN
(SRAND16):D=SGN(SRAND8)
920 I=SGN(SRAND4):Z=SGN(SRAND2):C=SRAND1
:RETURN
980 N=SGN(ACAND128):Z=1-SGN(AC):REM FLAG
S
990 PC=PC+1+L
1000 S=PC:E=PC
1010 PRINT"■■■■■■■■■■":GOSUB2040:GOTO200
1100 A=OP(PEEK(PC)):L=0:IFA=0THEN990
1110 ONAGOTO1200,1210,1220,1230,1240,125
0,1260,1270,1280,1290,1300,1310,1320,1330
1115 A=A-14
1120 ONAGOTO1340,1350,1360,1370,1380,139
0,1400,1410,1420,1430,1440,1450,1460,1470
1125 A=A-14
1130 ONAGOTO1480,1490,1500,1510,1520,153
0,1540,1550,1560,1570,1580,1590,1600,1610
1135 A=A-14
1140 ONAGOTO1620,1630,1640,1650,1660,167
0,1680,1690,1700,1710,1720,1730,1740,1750
1150 GOTO200
1200 IFDTHEN1205:REM ADC
1201 GOSUB1900:V=1-SGN(ACAND128):AC=AC+O
P+C:C=-(AC>FF)
1202 AC=ACANDFF:N=SGN(ACAND128):V=VANDN:
GOTO980
1205 GOSUB1900:AC=VAL(H$(AC/16)+H$(ACAND
15)):OP=VAL(H$(OP/16)+H$(OPAND15))
1206 AC=AC+OP+C:C=-(AC>99):IFAC>99THENAC
```

```
=AC-100
1207 A$=MID$(STR$(AC),2):GOSUB2390:AC=A:
GOTO980
1210 REM AND
1211 GOSUB1900:AC=ACANDOP:GOTO980
1220 REM ASL
1221 IFAD(PEEK(PC))=4THENAC=AC*2:C=-(AC>
FF):AC=ACANDFF:GOTO980
1222 GOSUB1900:A=OP*2:C=-(A>FF):A=AANDFF
:GOSUB1850
1223 N=SGN(OPANDFF):Z=1-SGN(OP):GOTO990
1230 REM BCC
1231 FL=1-C:GOTO1800
1240 REM BCS
1241 FL=C:GOTO1800
1250 REM BEQ
1251 FL=Z:GOTO1800
1260 REM BIT
1261 GOSUB1900:N=SGN(OPAND128):V=SGN(OPA
ND64):Z=1-SGN(OPANDAC):GOTO990
1270 REM BMI
1271 FL=N:GOTO1800
1280 REM BNE
1281 FL=1-Z:GOTO1800
1290 REM BPL
1291 FL=1-N:GOTO1800
1300 REM BRK
1301 PC=PC+2:IFPC>=ULTHENPC=PC-UL
1302 PH=INT(PC/HI):PL=PC-PH*HI:SP(SP)=PH
:SP=SP-1ANDFF:SP(SP)=PL:SP=SP-1ANDFF
1303 B=1:I=1:GOSUB900:SP(SP)=SR:SP=SP-1A
NDFF:PC=PEEK(UL-2)+HI*PEEK(UL-1):GOTO1000
1310 REM BVC
1311 FL=1-V:GOTO1800
1320 REM BVS
1321 FL=V:GOTO1800
1330 REM CLC
1331 C=0:GOTO990
1340 REM CLD
1341 D=0:GOTO990
1350 REM CLI
1351 I=0:GOTO990
1360 REM CLV
1361 V=0:GOTO990
1370 REM CMP
1371 GOSUB1900:A=AC-OP
1372 N=SGN(AAND128):Z=-(A=0):C=-(A>=0):G
OTO990
1380 REM CPX
```

```
1381 GOSUB1900:A=XR-OP:GOTO1372
1390 REM CPY
1391 GOSUB1900:A=YR-OP:GOTO1372
1400 REM DEC
1401 GOSUB1900:A=OP-1ANDFF:GOSUB1850
1402 GOTO1442
1410 REM DEX
1411 XR=(XR-1)ANDFF:GOTO1452
1420 REM DEY
1421 YR=(YR-1)ANDFF:GOTO1462
1430 REM EOR
1431 GOSUB1900:AC=(ACOROP)ANDNOT(ACANDOP
)
1432 GOTO980
1440 REM INC
1441 GOSUB1900:A=OP+1ANDFF:GOSUB1850
1442 N=SGN(AAND128):Z=1-SGN(A):GOTO990
1450 REM INX
1451 XR=(XR+1)ANDFF
1452 Z=1-SGN(XR):N=SGN(XRAND128):GOTO990
1460 REM INY
1461 YR=(YR+1)ANDFF
1462 Z=1-SGN(YR):N=SGN(YRAND128):GOTO990
1470 REM JMP
1471 GOSUB1900:PC=AD:GOTO1000
1480 REM JSR
1481 A=PC+2:PH=INT(A/HI):PL=A-PH*HI:SP(S
P)=PH:SP=SP-1ANDFF:SP(SP)=PL:SP=SP-1ANDFF
1482 PC=PEEK(PC+1)+PEEK(PC+2)*HI:GOTO100
0
1490 REM LDA
1491 GOSUB1900:AC=OP:GOTO980
1500 REM LDX
1501 GOSUB1900:XR=OP:GOTO1452
1510 REM LDY
1511 GOSUB1900:YR=OP:GOTO1462
1520 REM LSR
1521 IFAD(PEEK(PC))<>4THEN1524
1522 AC=AC/2
1523 C=-(AC<>INT(AC)):AC=ACANDFF:GOTO980
1524 GOSUB1900:A=OP/2:C=-(A<>INT(A)):A=A
ANDFF:GOSUB1850
1525 GOTO1442
1530 REM NOP
1531 GOTO990
1540 REM ORA
1541 GOSUB1900:AC=ACOROP:GOTO980
1550 REM PHA
1551 SP(SP)=AC:SP=SP-1ANDFF:GOTO990
```

```
1560 REM PHP
1561 GOSUB900:SP(SP)=SR:SP=SP-1ANDFF:GOT
0990
1570 REM PLA
1571 SP=(SP+1)ANDFF:AC=SP(SP):GOTO980:RE
M FLAGS ZETTEN
1580 REM PLP
1581 SP=(SP+1)ANDFF:SR=SP(SP):GOSUB910:G
OTO990
1590 REM ROL
1591 IFAD(PEEK(PC))=4THENAC=AC*2+C:GOTO1
523
1592 GOSUB1900:A=OP*2+C:C=-(A>FF)
1593 A=AANDFF:GOSUB1850
1594 GOTO1442
1600 REM ROR
1601 IFAD(PEEK(PC))=4THENAC=AC/2+128*C:G
OTO1523
1602 GOSUB1900:A=OP/2+128*C:C=-(A<>INT(A
)):GOTO1593
1610 REM RTI
1611 SP=SP+1ANDFF:SR=SP(SP):GOSUB910:GOT
01621
1620 REM RTS
1621 SP=SP+1ANDFF:A=SP(SP):SP=SP+1ANDFF:
PC=A+SP(SP)*HI:GOTO990
1630 IFDTHEN1635:REM SBC
1631 GOSUB1900:V=SGN(ACAND128):AC=AC-OP-
1+C:C=-(AC>=0)
1632 AC=ACANDFF:N=SGN(ACAND128):V=VAND1-
N:GOTO980
1635 GOSUB1900:AC=VAL(H$(AC/16)+H$(ACAND
15)):OP=VAL(H$(OP/16)+H$(OPAND15))
1636 AC=AC-OP+C-1:C=-(AC>=0):IFAC<0THENA
C=AC+100
1637 A$=MID$(STR$(AC),2):GOSUB2390:AC=A:
GOTO980
1640 REM SEC
1641 C=1:GOTO990
1650 REM SED
1651 D=1:GOTO990
1660 REM SEI
1661 I=1:GOTO990
1670 REM STA
1671 GOSUB1900:A=AC:GOSUB1850
1672 GOTO990
1680 REM STX
1681 GOSUB1900:A=XR:GOSUB1850
1682 GOTO990
```

```
1690 REM STY
1691 GOSUB1900:A=YR:GOSUB1850
1692 GOTO990
1700 REM TAX
1701 XR=AC:GOTO1452
1710 REM TAY
1711 YR=AC:GOTO1462
1720 REM TSX
1721 XR=SP:GOTO1452
1730 REM TXA
1731 AC=XR:GOTO980
1740 REM TXS
1741 SP=XR:GOTO990
1750 REM TYA
1751 AC=YR:GOTO980
1800 REM BRANCH-INSTRUCTIE
1810 IFFL=0THENL=1:GOTO990
1820 GOSUB1985:GOTO1000
1850 REM POKE
1870 IFAD<HIORAD>HI+FFTHEN1880
1875 SP(AD-HI)=A:RETURN
1880 IFESTHENPOKEAD,A
1885 RETURN
1900 REM OPERAND HALEN
1910 A=AD(PEEK(PC))
1920 ONAGOSUB1930,1935,1940,1945,1950,19
55,1960,1965,1970,1975,1980,1985,1990
1925 IFAD<HIORAD>HI+FFTHENRETURN
1927 OP=SP(AD-HI):RETURN
1930 AD=0:RETURN:REM GEIMPLICEERD
1935 AD=PC+1:OP=PEEK(AD):L=1:RETURN:REM
#
1940 AD=PEEK(PC+1):OP=PEEK(AD):L=1:RETUR
N:REM ZEROPAGE
1945 AD=0:RETURN:REM A
1950 AD=PEEK(PC+1)+HI*PEEK(PC+2):OP=PEEK
(AD):L=2:RETURN: REM ABSOLUUT
1955 AD=PEEK(PC+1)+XRANDFF:OP=PEEK(AD):L
=1:RETURN
1960 AD=PEEK(PC+1)+YRANDFF:OP=PEEK(AD):L
=1:RETURN
1965 AD=PEEK(PC+1)+HI*PEEK(PC+2)+XR:OP=P
EEK(AD):L=2:RETURN:REM ABSOLUUT,X
1970 AD=PEEK(PC+1)+HI*PEEK(PC+2)+YR:OP=P
EEK(AD):L=2:RETURN:REM ABSOLUUT,Y
1975 AD=PEEK(PEEK(PC+1))+HI*PEEK(PEEK(PC
+1)+1ANDFF)+YR:OP=PEEK(AD):L=1:RETURN
1980 AD=PEEK(PC+1)+XRANDFF:AD=PEEK(AD)+H
I*PEEK(AD+1):OP=PEEK(AD):L=1:RETURN
```

```
1985 A=PEEK(PC+1):A=A+HI*(A>127)+2+PC
1986 PC=INT(A/HI)*HI+((A+(A>SC)*UL)ANDFF
):RETURN:REM RELATIEF
1990 AD=PEEK(PC+1)+HI*PEEK(PC+2):AD=PEEK
(AD)+HI*PEEK(AD+1):OP=PEEK(AD):RETURN
2040 FORP=STOE:PRINT" ";
2050 A=P:GOSUB2290:REM ADRES
2060 PRINT" ";:A=PEEK(P):GOSUB2320:PRINT
" ";:J=PEEK(P):OP=AD(J)
2070 ONOPGOSUB2350,2360,2360,2350,2370,2
360,2360,2370,2370,2360,2360,2360,2370
2080 PRINT" ";MN$(J)" ";
2090 ONOPGOSUB2110,2120,2130,2140,2150,2
160,2170,2180,2190,2200,2210,2220,2240
2100 PRINT"           ":NEXTP
2105 IFP>=ULTHENP=P-UL
2110 RETURN
2120 PRINT"#";:GOSUB2330:P=P+1:RETURN
2130 GOSUB2330:P=P+1:RETURN
2140 PRINT" A";:RETURN
2150 GOSUB2260:P=P+2:RETURN
2160 GOSUB2330:P=P+1:PRINT",X";:RETURN
2170 GOSUB2330:P=P+1:PRINT",Y";:RETURN
2180 GOSUB2260:P=P+2:PRINT",X";:RETURN
2190 GOSUB2260:P=P+2:PRINT",Y";:RETURN
2200 PRINT"(";:GOSUB2330:P=P+1:PRINT"),Y
";:RETURN
2210 PRINT"(";:GOSUB2330:P=P+1:PRINT",X)
";:RETURN
2220 A=PEEK(P+1):A=A+HI*(A>127)+2+P
2230 A=INT(A/HI)*HI+((A+(A>SC)*UL)ANDFF)
:PRINT"$";:GOSUB2290:P=P+1:RETURN
2240 PRINT"(";:GOSUB2260
2250 PRINT")";:P=P+2:RETURN
2260 PRINT"$";
2270 A=PEEK(P+1)+HI*PEEK(P+2)
2280 REM HEXADRES A
2290 HB=INT(A/HI):A=A-HI*HB
2300 PRINTH$(HB/16)H$(HBAND15);
2310 REM HEXBYTE A
2320 PRINTH$(A/16)H$(AAND15);:RETURN
2330 PRINT"$";
2340 A=PEEK(P+1):GOTO2320
2350 PRINT"     ";:RETURN
2360 GOSUB2340:PRINT"    ";:RETURN
2370 GOSUB2340:PRINT" ";:A=PEEK(P+2):GOT
O2320
2380 IFASC(A$)=42THENEND
2390 A=0:FORJ=1TOLEN(A$):X=ASC(RIGHT$(A$
```

```
,J))-48:X=X+(X>9)*7:A=A+X*(16↑(J-1)):NEXT:RETURN
3000 PRINT:PRINT"■■":PRINT"ADRES   :    $$
***■■■■■■■";:INPUTA$:GOSUB2380
3010 PRINT"■",,:AD=A:OP=PEEK(A):GOSUB192
5:A=OP:GOSUB2320:INPUT"■■■■■";A$:GOSUB2380
3020 GOSUB1850:PRINT"■
          ":IFAD=PCTHEN1000
3030 GOTO200
3100 INPUT"ECHTE SIMULATIE    J■■■■";ES$:E
S=ES$="J":GOTO200
10000 DATA0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,
F
10010 DATA"BRK",11,1,"ORA",35,11,"???",0
,1
10020 DATA"???",0,1,"???",0,1,"ORA",35,3
10030 DATA"ASL",3,3,"???",0,1,"PHP",37,1
10040 DATA"ORA",35,2,"ASL",3,4,"???",0,1
10050 DATA"???",0,1,"ORA",35,5,"ASL",3,5
10060 DATA"???",0,1,"BPL",10,12,"ORA",35
,10
10070 DATA"???",0,1,"???",0,1,"???",0,1
10080 DATA"ORA",35,6,"ASL",3,6,"???",0,1
10090 DATA"CLC",14,1,"ORA",35,9,"???",0,
1
10100 DATA"???",0,1,"???",0,1,"ORA",35,8
10110 DATA"ASL",3,8,"???",0,1,"JSR",29,5
10120 DATA"AND",2,11,"???",0,1,"???",0,1
10130 DATA"BİT",7,3,"AND",2,3,"ROL",40,3
10140 DATA"???",0,1,"PLP",39,1,"AND",2,2
10150 DATA"ROL",40,4,"???",0,1,"BIT",7,5
10160 DATA"AND",2,5,"ROL",40,5,"???",0,1
10170 DATA"BMI",8,12,"AND",2,10,"???",0,
1
10180 DATA"???",0,1,"???",0,1,"AND",2,6
10190 DATA"ROL",40,6,"???",0,1,"SEC",45,
1
10200 DATA"AND",2,9,"???",0,1,"???",0,1
10210 DATA"???",0,1,"AND",2,8,"ROL",40,8
10220 DATA"???",0,1,"RTI",42,1,"EOR",24,
11
10230 DATA"???",0,1,"???",0,1,"???",0,1
10240 DATA"EOR",24,3,"LSR",33,3,"???",0,
1
10250 DATA"PHA",36,1,"EOR",24,2,"LSR",33
,4
10260 DATA"???",0,1,"JMP",28,5,"EOR",24,
5
10270 DATA"LSR",33,5,"???",0,1,"BVC",12,
12
```

```
10280 DATA"EOR",24,10,"???",0,1,"???",0,
1
10290 DATA"???",0,1,"EOR",24,6,"LSR",33,
6
10300 DATA"???",0,1,"CLI",16,1,"EOR",24,
9
10310 DATA"???",0,1,"???",0,1,"???",0,1
10320 DATA"EOR",24,8,"LSR",33,8,"???",0,
1
10330 DATA"RTS",43,1,"ADC",1,11,"???",0,
1
10340 DATA"???",0,1,"???",0,1,"ADC",1,3
10350 DATA"ROR",41,3,"???",0,1,"PLA",38,
1
10360 DATA"ADC",1,2,"ROR",41,4,"???",0,1
10370 DATA"JMP",28,13,"ADC",1,5,"ROR",41
,5
10380 DATA"???",0,1,"BVS",13,12,"ADC",1,
10
10390 DATA"???",0,1,"???",0,1,"???",0,1
10400 DATA"ADC",1,6,"ROR",41,6,"???",0,1
10410 DATA"SEI",47,1,"ADC",1,9,"???",0,1
10420 DATA"???",0,1,"???",0,1,"ADC",1,8
10430 DATA"ROR",41,8,"???",0,1,"???",0,1
10440 DATA"STA",48,11,"???",0,1,"???",0,
1
10450 DATA"STY",50,3,"STA",48,3,"STX",49
,3
10460 DATA"???",0,1,"DEY",23,1,"???",0,1
10470 DATA"TXA",54,1,"???",0,1,"STY",50,
5
10480 DATA"STA",48,5,"STX",49,5,"???",0,
1
10490 DATA"BCC",4,12,"STA",48,10,"???",0
,1
10500 DATA"???",0,1,"STY",50,6,"STA",48,
6
10510 DATA"STX",49,7,"???",0,1,"TYA",56,
1
10520 DATA"STA",48,9,"TXS",55,1,"???",0,
1
10530 DATA"???",0,1,"STA",48,8,"???",0,1
10540 DATA"???",0,1,"LDY",32,2,"LDA",30,
11
10550 DATA"LDX",31,2,"???",0,1,"LDY",32,
3
10560 DATA"LDA",30,3,"LDX",31,3,"???",0,
1
10570 DATA"TAY",52,1,"LDA",30,2,"TAX",51
,1
```

```
10580 DATA"???",0,1,"LDY",32,5,"LDA",30,
5
10590 DATA"LDX",31,5,"???",0,1,"BCS",5,1
2
10600 DATA"LDA",30,10,"???",0,1,"???",0,
1
10610 DATA"LDY",32,6,"LDA",30,6,"LDX",31
,7
10620 DATA"???",0,1,"CLV",17,1,"LDA",30,
9
10630 DATA"TSX",53,1,"???",0,1,"LDY",32,
8
10640 DATA"LDA",30,8,"LDX",31,9,"???",0,
1
10650 DATA"CPY",20,2,"CMP",18,11,"???",0
,1
10660 DATA"???",0,1,"CPY",20,3,"CMP",18,
3
10670 DATA"DEC",21,3,"???",0,1,"INY",27,
1
10680 DATA"CMP",18,2,"DEX",22,1,"???",0,
1
10690 DATA"CPY",20,5,"CMP",18,5,"DEC",21
,5
10700 DATA"???",0,1,"BNE",9,12,"CMP",18,
10
10710 DATA"???",0,1,"???",0,1,"???",0,1
10720 DATA"CMP",18,6,"DEC",21,6,"???",0,
1
10730 DATA"CLD",15,1,"CMP",18,9,"???",0,
1
10740 DATA"???",0,1,"???",0,1,"CMP",18,8
10750 DATA"DEC",21,8,"???",0,1,"CPX",19,
2
10760 DATA"SBC",44,11,"???",0,1,"???",0,
1
10770 DATA"CPX",19,3,"SBC",44,3,"INC",25
,3
10780 DATA"???",0,1,"INX",26,1,"SBC",44,
2
10790 DATA"NOP",34,1,"???",0,1,"CPX",19,
5
10800 DATA"SBC",44,5,"INC",25,5,"???",0,
1
10810 DATA"BEQ",6,12,"SBC",44,10,"???",0
,1
10820 DATA"???",0,1,"???",0,1,"SBC",44,6
10830 DATA"INC",25,6,"???",0,1,"SED",46,
1
```

```
10840 DATA"SBC",44,9,"???",0,1,"???",0,1
10850 DATA"???",0,1,"SBC",44,8,"INC",25,
8
10860 DATA"???",0,1
```

6.1 Programmabeschrijving van de stap-voor-stap simulator

100-190

Opbouw van het overzicht van de registermonitor. Initialisatie van de variabelen en velden.

200-360

Overzicht van de registerinhoud. De inhoud van het processorregister wordt weergegeven in hexadecimale getallen. Bij de flags gebeurt dit via de CHR$-functie, die afhankelijk van de waarde 0 of 1 oplevert.

400-530

De invoer via het toetsenbord wordt geïnterpreteerd. Drukt u op de spatiebalk dan springt het programma naar regel 1100 waar de echte simulatie begint. Bij de registers wordt naar een invoerroutine gesprongen die de oude waarde aangeeft en de invoer van een nieuwe verwacht. Bij de flags wordt de toestand omgekeerd. Is de toets CURSOR OMLAAG ingedrukt dan springt het programma naar de disasemblerroutine en wordt de volgende instructie aangegeven.

900-920

Berekening van de waarde van statusregister SR op grond van de afzonderlijke flags en omgekeerd.

980

Sprong als de N- en Z-flags worden gezet.

990

Sprong als de programmawijzer wordt verhoogd.

1000-1010

Disassembleren van de volgende instructie.

1100-1150

Stap-voor-stap simulatie. Welke routine wordt aangesproken is afhankelijk van de instructie.

1200-1751

Simulatieroutines voor alle 6510-instructies. De alfabetische volgorde van de instructies bepaalt die van de routines. Na afloop springen de routines naar regel 990 waar de verhoging van de programmawijzer afhangt van de lengte der instructies. Bij de sprong naar regel 980 worden bovendien de N- en Z-flags ingesteld overeenkomstig de waarde in de accu.

1800-1820

Hier worden alle branchinstructies afgehandeld nadat de waarde van de flag in de variabele FL is gezet.

1850-1885

Deze routine dient om waarden in het geheugen te schrijven. Het stackbereik ($100-$1FF) wordt afzonderlijk behandeld. De POKE's worden alleen uitgevoerd als de echte simulatie werd gewenst (variabele ES).

1900-1990

Deze routine haalt de operanden die bij de instructies horen. Ze verschillen per adresseringswijze. Is de routine opgeroepen dan staat het adres van de operanden in AD, de waarde zelf in OP.

2040-2370

Deze routine is een disassembler die na elke stap wordt opgeroepen om de volgende regel te disassembleren. In regel 2070 worden de bytes van de operanden uitgevoerd. Ze verschillen per adresseringswijze. Regel 2090 doet dit nog een keer waarbij de presentatie wordt aangepast aan de wijze van adresseren. Bevat een geheugenadres geen geldige instructiecode dan worden er drie vraagtekens uitgevoerd. De volgende routines voeren dan de eerder aangesproken opdrachten uit evenals de transformatie van decimaal naar hexadecimaal.

2380-2390

Hier wordt elke invoer omgerekend van decimaal naar hexadecimaal. Toetst u een sterretje in, dan wordt het programma beëindigd.

3000-3030

Deze routine geeft een overzicht van de geheugeninhoud en dient om er wijzigingen in aan te brengen. Veranderingen zijn alleen mogelijk als u de echte simulatie heeft gekozen.

3100
Hier kunt u beslissen of u de echte simulatie wilt of niet.

10000-10860
Vergezeld van hun nummers en adresseringswijzen staan in deze regel de instructies die in het begin in de velden worden gezet.

6.2 De belangrijkste variabelen

FF
constante 255

HI
constante 256

UL
constante 65536

SC
constante 32767

MN$(255)
tabel van 6510-mnemonics

OP(255)
tabel van de instructienummers voor de stap-voor-stap simulatie

AD(255)
dit veld bevat voor elk veld de adresseringswijze

SP(255)
in dit veld staat de stack

H$(15)
veld met hexadecimale getallen

PC
programmawijzer

AC
accumulator

XR
X-register

YR
Y-register

SR
statusregister

SP
stackpointer

N
flag voor negatief

V
overflowflag

B
breakflag

D
flag voor decimaal

I
interruptflag

Z
zeroflag

C
carryflag

T$
ingedrukte toets

L
lengte van operanden

ES
flag voor echte simulatie

OP
operand

7 MACHINETAALPROGRAMMA'S OP DE COMMODORE 64

Bij het programmeren van de Commodore 64 is machinetaal zeer geschikt voor grafische toepassingen. De grafische mogelijkheden komen dan goed tot hun recht. Uitgaande van een mogelijke oplossing van een probleem in BASIC zal geprobeerd worden om de overeenkomstige oplossing in machinetaal te formuleren. Bovendien zullen we langzamerhand alle programmeertechnieken en mogelijkheden van de processor leren kennen en benutten.

Juist de programmering voor grafische doeleinden laat zien waar de kracht van het programmeren in machinetaal ligt. In BASIC is dit meestal een verzameling van ondoorzichtige PEEKs en POKE's. Tevens zullen we zien hoe machinetaalprogramma's met BASIC-programma's worden gecombineerd en hoe de kracht van beide talen het best wordt benut. Op de functies van de VIDEO-chip gaan we alleen in als dat voor de oplossing van het probleem nodig is. Als u nader op de hardware en het operating system van de Commodore 64 wilt ingaan, en dat zal vroeg of laat zeker het geval zijn, kunt u daarover alle nodige informatie vinden in 64 INTERN van DATA BECKER NEDERLANDS*.

Voordat u begint met het eerste voorbeeld bekijken we hoe vanuit BASIC machinetaalprogramma's worden gebruikt en hoe de parameters tussen beide programma's worden uitgewisseld. Voor het aanroepen van machinetaalprogramma's dient in de eerste plaats de SYS-instructie. Daarbij wordt het startadres van het machinetaalprogramma aangegeven. Als het machinetaalprogramma de instructie RTS bereikt, wordt teruggesprongen in BASIC. Het BASIC-programma wordt vervolgens afgewerkt. Als een machinetaalroutine een bepaalde opdracht moet uitvoeren, zoals het wissen van een hires-scherm (high resolution of hoog oplossend vermogen van het scherm), zijn geen verdere parameters nodig en de simpele SYS-instructie voert de opdracht uit. Moet echter op het scherm een punt worden gezet dan moet aan het machinetaalprogramma meegedeeld worden welk punt bedoeld wordt. Daarvoor zijn verschillende methodes. Bij de zogenaamde brievenbusmethode worden de parameters in een of meerdere geheugenplaatsen gedeponeerd. Van te voren is vastgesteld welke adressen als brievenbus dienst doen. In ons voorbeeld kan het ene adres het X-coördinaat en het andere adres het Y-coördinaat bevatten. Vanuit BASIC kan dit met twee POKE-instructies. Het machinetaalprogramma kan ook de waarden van de coördinaten uit deze geheugenplaatsen halen en verwerken. De BASIC-interpreter biedt echter nog een speciale methode. Als de SYS-instructie wordt uitgevoerd bestaat de mogelijkheid om bepaalde

registerinhouden aan de processor over te dragen. Omdat vanuit BASIC niet direkt teruggegrepen kan worden op de registerinhouden van de processor zijn daarvoor vier adressen gereserveerd. Bij het oproepen van het machinetaalprogramma wordt de inhoud van de volgende vier adressen overgedragen aan de vier registers, voordat verder gegaan wordt met de machinetaalroutine:

```
780   =>   accumulator
781   =>   X-register
782   =>   Y-register
783   =>   statusregister
```

Als de machinetaalroutine met een bepaalde accumulatorinhoud gestart moet worden, is het voldoende om voor de SYS-instructie de waarde met een POKE-instructie in adres 780 te zetten. De adressen 781 en 782 zijn voor het X- en Y-register beschikbaar. Bij het statusregister is enige voorzichtigheid geboden. U moet erop letten dat de decimal- of de interruptflag niet slordig wordt gezet, aangezien dat aanleiding tot complicaties kan geven. Als de computer aangezet wordt is de inhoud van de vier geheugenplaatsen nul, als u niet let op het statusregister zijn na het oproepen van het machinetaalprogramma alle flags gewist. Na het afwerken van de SYS-instructie worden de inhouden van de vier registers gezet in de vier geheugenplaatsen. De inhouden van de vier registers blijven dus in BASIC beschikbaar. Op deze manier is het mogelijk om drie of vier 8-bitswaarden tussen BASIC-en machinetaalprogramma's en omgekeerd, uit te wisselen. Voor de meeste toepassingen is dat ruimschoots voldoende, zodat ook wij daar gebruik van maken. Moeten meer parameters worden overgedragen dan moet u op dezelfde wijze als boven beschreven bepaalde geheugenplaatsen daarvoor bestemmen.

Er is nog een tweede methode die bijzonder comfortabel is. Daarbij worden routines van de BASIC-interpreter gebruikt. Bereikt de BASIC-interpreter bijvoorbeeld een instructie als POKE 780, 10 dan roept hij, na het herkennen van de POKE-instructie, een algemeen toepasbare routine op die een parameter uit het BASIC-programma haalt. Deze routine kan niet alleen, zoals in ons voorbeeld, van constanten gebruik maken, maar ook van willekeurige ingewikkelde uitdrukkingen zoals POKE A+7. 5*Z%(INT(SIN(X)*1000)), EXP(X). Van deze routine kan nuttig gebruik worden gemaakt en we hebben daarmee ook de mogelijkheid van dergelijke gecompliceerde uitdrukkingen gebruik te maken. Later zal op deze routine en het gebruik ervan ingegaan worden. Eerst gaan we parameters direkt aan de registers doorgeven.

Voordat we ons storten op het programmeren van grafiek nog een paar fundamentele opmerkingen daarover. Met de hoogoplossende (Hires=high resolution) grafiek kunt u zich richten op elk

afzonderlijk punt van het scherm afzonderlijk. Bij het gebruik van de normale karakterset heeft u in totaal 40*25=1000 posities tot uw beschikking. Elk hokje waarin een karakter geplaatst kan worden bestaat uit 8*8=64 beeldpunten. Bij de hoogoplossende grafiek staan dus 320*200=64000 beeldpunten ter beschikking. Zowel bij het normale gebruik als bij het hires-gebruik wordt het scherm afgebeeld in een zogenaamd video-RAM. Bij normaal schermgebruik is voor ieder hokje een byte nodig, bij het hires-schermgebruik is voor ieder beeldpunt een bit nodig. (0=beeldpunt uit; 1=beeldpunt aan). Voor normaal gebruik zijn dus 40*25=1000 bytes nodig, voor hires-gebruik 200*320=64000 bits of 8000 bytes. Voor normaal gebruik ligt het video-RAM van adres 1024 t/m 2023 ($0400 t/m $07E8). Dit startadres van het video-RAM kan worden veranderd door een andere waarde te zetten in het video-controleregister, adres 53272 ($D018). Het scherm kan in stappen van 1K vezet worden naar $0000, $0400, $0800, $0C00. . . .: Bij het hires-gebruik is een geheugengebied van 8K nodig. Het startadres van dit gebied kan weer door het zetten van een bepaalde waarde in het video-controleregister 53272. Daarbij moet worden bedacht dat het controleregister slechts 16K kan adresseren. Welk 16K-gebied het betreft wordt door een bepaalde waarde in een I/O register aangegeven. Eerst gaan we nu een geschikt 8K-gebied voor ons hires-scherm uitzoeken.

Op het eerste gezicht ligt het voor de hand om 8K van het BASIC-gebied te kiezen. Maar we programmeren immers in machinetaal! Het gehele geheugenbied van de Commodore 64 bestaat uit vrij te gebruiken RAM-gebied. Ook onder het ROM van het operating system, adres $E000 t/m $FFFF (57344 t/m 65535) ligt een vrij te gebruiken RAM-gebied. Dit gebruiken we als grafiek-RAM. Vanuit BASIC is dit gebruik niet mogelijk, omdat het lezen en schrijven van dit geheugen alleen maar mogelijk is door de BASIC-interpreter en het operating system uit te schakelen. Het normale video-RAM wordt als kleuren-RAM gebruikt. Aangezien het video-RAM in hetzelfde 16K gebied moet liggen als het grafiek-RAM ($C000-$FFFF) kiezen we gemakshalve $C000 t/m $C3FF; dit wordt immers ook door BASIC niet benut. Omdat voor het kleuren-RAM slechts 1K beschikbaar is, is meteen duidelijk dat niet elk beeldpunt een eigen kleur kan krijgen. De kleur geldt voor een hokje van 8*8 punten, dezelfde grootte die de karakters hadden.

Verschillende hulpprogramma's gaan we nu in machinetaal schrijven. Deze routines kunnen later in BASIC gebruikt worden. De eerste routine moet de omschakeling van de normale karakterinstelling naar de hires-instelling bewerkstelligen. Het hires-scherm moet in het boven beschreven geheugengebied worden gelegd. Dit heeft bovendien het voordeel dat door de kleur van de hires-punten de normale beeldscherminhoud niet verstoord wordt en bij het terugschakelen behouden blijft.

Eerst wordt de omschakeling in BASIC geformuleerd.

```
100 V = 53248    : REM STARTADRES VIDEOCONTROLLER
110 V1 = V+17    : REM SCHAKELAAR GRAFISCHE MODE
120 V2 = V+24    : REM BEPAALD VIDEORAMADRES
130 CIA = $DD00 : REM BEPAALD 16K GEBIED
140 POKE V1,59
150 POKE V2,8
160 POKE CIA,0
170 END
```

Als dit in machinetaal moet worden omgezet moet eerst het startadres van het machinetaalprogramma worden bepaald. Omdat het gebied $C000 t/m $C3FF door het kleuren-RAM bezet is, kiezen we als startadres $C400. De omzetting van de BASIC-instructies in die voor machinetaal zullen geen problemen opleveren.

```
100 VIDEO = 53248 ; VIDEOCONTROLLER
110 V1 = 53265    ; ADRES VOOR GRAFISCHE MODE
120 V2 = 53272    ; ADRES VOOR VIDEORAMADRES
130 CIA = $DD00   ; 16K-SELECTIE
140 *= $C400      ; BEGIN VAN ROUTINE
150 LDA #59
160 STA V1
170 LDA #8
180 STA V2
190 LDA #0
200 STA CIA
210 RTS
220 .EN
```

Na het assembleren verschijnt de volgende listing:

```
D000                    100 VIDEO  =    53248
D011                    110 V1     =    53265
D018                    120 V2     =    53272
DD00                    130 CIA    =    $DD00
                        140        *=   $C400
C400  A9 3B             150 IN     LDA  #59
C402  8D 11 D0          160        STA  V1
C405  A9 08             170        LDA  #8
C407  8D 18 D0          180        STA  V2
C40A  A9 00             190        LDA  #0
C40C  8D 00 DD          200        STA  CIA
C40F  60                210        RTS
```

```
220         .EN
```

Voordat deze routine wordt uitgeprobeerd moet eerst de omschakeling naar het normale gebruik worden geprogrammeerd. De registers moeten dan weer met de oorspronkelijke waarden worden geladen. Gemakshalve wordt dit deel achter het vorige gezet.

```
100 VIDEO = 53248 ; VIDEOCONTROLLER
110 V1 = 53265    ; ADRES VOOR GRAFISCHE MODE
120 V2 = 53272    ; ADRES VOOR VIDEORAMADRES
130 CIA = $DD00   ; 16K - SELECTIE
140 *= $C400      ; BEGIN VAN ROUTINE
150 IN LDA #59
160 STA V1
170 LDA #8
180 STA V2
190 LDA #0
200 STA CIA
210 RTS
220 ; UITSCHAKELEN
230 UIT LDA #27
240 STA V1
250 LDA #21
260 STA V2
270 LDA #3
280 STA CIA
290 RTS
300 .EN
```

Na het assembleren wordt tevens de lijst labels uitgevoerd. De labels IN en UIT zijn gedefinieerd om later over de adressen te beschikken waarnaar, volgens de SYS-instructie, gesprongen moet worden.

```
D000                    100 VIDEO  =     53248
D011                    110 V1     =     53265
D018                    120 V2     =     53272
DD00                    130 CIA    =     $DD00
                        140        *=    $C400
C400  A9 3B             150 IN     LDA   #59
C402  8D 11 D0          160        STA   V1
C405  A9 08             170        LDA   #8
C407  8D 18 D0          180        STA   V2
C40A  A9 00             190        LDA   #0
C40C  8D 00 DD          200        STA   CIA
C40F  60                210        RTS
C410                    220                    ;UITSCHAKELEN
C410                    230 UIT    LDA   #27
C412  8D 11 D0          240        STA   V1
```

```
C415  A9 15        250        LDA   #21
C417  8D 18 D0     260        STA   V2
C41A  A9 03        270        LDA   #3
C41C  8D 00 DD     280        STA   CIA
C41F  60           290        RTS
                   300        .EN
```

C400 / C420 / 0020

```
SOURCEFILE IS VOORBEELD 1.SRC
 0  FOUTEN
UIT      C410      CIA       DD00      IN       C400
VIDEO    D000      V1        D011      V2       D018
```

Voordat de routine wordt uitgeprobeerd berekenen we eerst de decimale startadressen IN en UIT $C400=50176 en $C410=50192. Met het volgende BASIC-programma wordt de routine getest:

```
100 SYS 50176 : REM HIRES INSCHAKELEN
110 GET A$    : IF A$="" THEN 110
120 SYS 50192 : REM HIRES UITSCHAKELEN
```

Het programma schakelt naar het grafisch gebruik met hoogoplossend vermogen om, wacht dan op het indrukken van een willekeurige toets en keert dan terug naar het scherm voor normaal karaktergebruik.

Na het starten van het programma verschijnt een veelkleurige chaos op het scherm. Na het indrukken van een toets wordt weer naar de normale toestand teruggekeerd. Wat u gezien heeft waren toevallige waarden die na het inschakelen van het hires-scherm in het geheugen stonden, na het inschakelen van het anders ongebruikte RAM-gebied. Onze volgende opdracht is om het hires-scherm en het bijbehorende kleuren-RAM te wissen. In BASIC wordt dit met een POKE-lus gedaan.

Om alle punten in het grafiekgeheugen te wissen moet iedere betrokken bit de waarde nul hebben en als gevolg daarvan ook iedere byte. De lus moet de adressen $E000 - $FFFF omvatten (eigenlijk t/m $FF3F omdat niet 8192 maar 8000 bytes gewist moeten worden).

```
FOR I = 57344 TO 65535 : POKE I,0 :NEXT I
```

Dit is gemakkelijk en snel geformuleerd, maar het grote nadeel is dat deze lus ongeveer 30 seconden duurt en dat is voor veel toepassingen veel te lang. In machinetaal is het proces veel sneller.

Een lus hebben we al geprogrammeerd toen de karakterset van onze computer op het scherm werd gezet. Deze lus betrof 256 bytes, het gebied dat met de inhoud van de indexregisters X en Y kon worden bestreken. We moeten echter 8000 bytes wissen. We proberen het met twee genestelde lussen. In BASIC gaat dat als volgt:

```
AD = 57344
FOR X = 0 TO 31
FOR Y = 0 TO 255
POKE AD+Y, 0
NEXT Y
AD = AD+256
NEXT X
```

Het gebied van de 8192 bytes is verdeeld in 32 onderdelen (pagina's) van elk 256 bytes. Met de Y-lus worden 256 bytes gewist, dan wordt het basisadres met 256 vergroot. Dit gebeurt 32 keer, de lusvariabele X telt het aantal omlopen. Zoals u ziet worden hier steeds gehele bladzijden van elk 256 gewist, wat bij onze toepassing niet stoort. De teveel gewiste 192 bytes hebben immers geen speciale functie. We proberen nu een formulering in machinetaal:

```
100 AD = $E000
110 LDA #0 ; ACCUMULATOR WISSEN
120 LDX #0
130 LDY #0
140 STA AD,Y
150 INY
160 BNE SYMB1
170 ; AD = AD + $100
180 INX
190 ; IS X AL 31 ?
200 ; ZO NIET DAN TERUG NAAR REGEL 130
210 .EN
```

Met onze huidige kennis kan onze poging er zo uitzien. Het merkteken SYMB1 hebt u misschien al geplaatst in regel 140. Maar de verhoging van adres AD werkt natuurlijk niet zo. In adres 140 wordt de geïndexeerde adresseerwijze gebruikt, de inhoud van de accumulator wordt gezet in adres AD + inhoud Y-register. Een andere adresseerwijze is voor ons doel geschikter, namelijk de indirekte geïndexeerde. Daarbij wordt het actuele adres gehaald uit een 2-bytes wijzer in de zeropage met daarbij de inhoud van het Y-register. Deze wijzer kan later met $100 verhoogd worden, zoals in regel 170 verlangd wordt. De normale geïndexeerde adresseerwijze kan slechts 256 bytes omvatten. De vraag aan het X-register of de inhoud al 31 is levert geen probleem op. Bij een inhoud kleiner dan 31 wordt teruggesprongen naar regel 130. Het

programma kan er zo uitzien:

```
100 AD = $E000
110 LDA #0 ; ACCUMULATOR WISSEN
120 LDX #0
130 SYMB2 LDY #0
140 SYMB1 STA (AD),Y
150 INY
160 BNE SYMB1
170 ; AD = AD + $100
180 INX
190 CPX #32
200 BNE SYMB2
210 .EN
```

Bij de indirekte geïndexeerde adresseerwijze moet het adres een 2-byteswijzer op de zeropage zijn en niet zoals boven een absoluut adres. We kunnen bijvoorbeeld $FA en $FB gebruiken. Deze wijzer moet eerst met het adres $E000 geladen worden: de lo-byte ($00)in $FA en de hi-byte ($E0) in $FB. Nu is ook de verhoging van dit adres met $100 geen probleem meer, omdat eenvoudig de hi-byte met 1 verhoogd wordt. Tenslotte wordt nog de RTS-instructie toegevoegd. Het kant en klare programma is dan:

```
90 * = $C420
100 LDA #<$E000
102 STA $FA
104 LDA #>$E000
106 STA $FB
110 LDA #0 ; ACCUMULATOR WISSEN
120 LDX #31
130 SYMB2 LDY #0
140 SYMB1 STA (AD),Y
150 INY
160 BNE SYMB1
170 INC $FB
180 INX
190 CPX #32
200 BNE SYMB2
205 RTS
210 .EN
```

De programmastart is, in regel 90, op $C420 gezet, de eerstvolgende vrije geheugenplaats na de vorige routines om het hires-scherm aan en uit te schakelen. Het programma wordt in het geheugen gezet en geassembleerd:

```
00FA              90 AD     =     $FA
00FB              95 AD1    =     $FB
```

```
C420                    97          *=    $C420
C420  A9 00             100         LDA   #<$E000
C422  8D FA 00          102         STA   AD
C425  A9 E0             104         LDA   #>$E000
C427  8D FB 00          106         STA   AD1
C42A  A9 00             110         LDA   #0
C42C  A2 00             120         LDX   #0
C42E  A0 00             130 SYMB2   LDY   #0
C430  91 FA             140 SYMB1   STA   (AD),Y
C432  C8                150         INY
C433  D0 FB             160         BNE   SYMB1
C435  EE FB 00          170         INC   AD1
C438  E8                180         INX
C439  E0 20             190         CPX   #32
C43B  D0 F1             200         BNE   SYMB2
C43D  60                205         RTS
                        210         .EN
```

```
C420 / C43E / 001E

SOURCEFILE IS VOORBEELD 2.SRC

0  FOUTEN

AD      00FA      AD1     00FB      SYMB1  C430      SYMB2  C42E
```

Dit programma functioneert, maar we zullen zien dat enkele onderdelen korter en eleganter opgelost kunnen worden. Ten eerste kan bij de adressen AD en AD1 de adresseerwijze van de zeropage worden gebruikt. Dit wordt gedaan door het vooraf plaatsen van een ster. Het laden van de accumulator in regel 110 is niet nodig omdat dit al plaats vond in regel 100. Regel 100/102 wordt daarom gewisseld met regel 104/106. Bovendien laten we de X-lus van 32 naar 0 lopen, zodat de test in regel 190 kan vervallen. Verder kan het Y-register simpeler met nul laden door de accumulatorinhoud daarin te kopiëren. Met deze verbeteringen wordt het programma nog een stuk korter.

```
00FA                    90 AD     =     $FA
00FB                    95 AD1    =     $FB
C420                    97        *=    $C420
C420  A9 E0             100       LDA   #>$E000
C422  85 FB             102       STA   *AD1
C424  A9 00             104       LDA   #<$E000
C426  85 FA             106       STA   *AD
C428  A2 20             110       LDX   #32
C42A  A8                120 SYMB2 TAY
```

```
C42B  91 FA      130 SYMB1  STA   (AD),Y
C42D  C8         140        INY
C42E  D0 FB      150        BNE   SYMB1
C430  E6 FB      160        INC   *AD1
C432  CA         170        DEX
C433  D0 F5      180        BNE   SYMB2
C435  60         190        RTS
                 200        .EN
```

C420 / C436 / 0016

SOURCEFILE IS VOORBEELD 2.SRC

0 FOUTEN

AD 00FA AD1 00FB SYMB1 C42B SYMB2 C42A

Het programma wordt daardoor natuurlijk ook sneller. Met het volgende programma worden de drie routines uitgeprobeerd:

```
100 SYS 50176 : REM HIRES-SCHERM IN
110 GET A$ : IF A$="" THEN 110
120 SYS 51208 : REM SCHERM WISSEN
130 GET A$ : IF A$="" THEN 130
140 SYS 50192 : REM NORMAAL SCHERM
```

Na het starten met RUN wordt op grafisch gebruik overgeschakeld. Na het indrukken van een toets wordt het scherm gewist in onderdelen van een seconde. In BASIC duurde dit nog 30 seconden! Door nogmaals te drukken wordt weer teruggeschakeld naar de normale toestand. Het zal ons nu niet moeilijk vallen om een overeenkomstige routine voor het in orde brengen van het kleur-RAM te ontwerpen.

Het kleur-RAM beslaat de 1000 bytes van het gebied $C000 t/m C3FF. Iedere byte beslaat een groep van 8 bij 8 punten. De inhoud van een byte bepaalt de achtergrondkleur (niet gezette punten) en de voorgrondkleur (gezette punten). De vier onderste bits (0 t/m 3) bepalen de kleur van de achtergrond. De kleur van de voorgrond wordt bepaald door de inhoud van de vier bovenste bits (4 t/m 7). Als bijvoorbeeld de waarde $10 in een adres gezet wordt, is de inhoud van de laagste nibble 0 (zwart) en de inhoud van de hoogste nibble 1 (wit). De achtergrond is zwart en de kleur van de gezette punten wit. In BASIC gaat dit met: POKE AD, AK+16*PK. In adres AD (schermhokje) is de achtergrondkleur AK en de kleur van de gezette punten PK. Probeert u nu zelf een routine in machinetaal te schrijven om de kleuren vast te leggen. Als startadres wordt

$C440 gekozen.

```
00FA                    90 AD      =     $FA
00FB                    95 AD1     =     $FB
C440                    97         *=    $C440
C440  A0 C0            100         LDY   #>$C000
C442  84 FB            102         STY   *AD1
C444  A0 00            104         LDY   #<$C000
C446  84 FA            106         STY   *AD
C448  A2 04            110         LDX   #4
C44A  91 FA            130 SYMB1   STA   (AD),Y
C44C  C8               140         INY
C44D  D0 FB            150         BNE   SYMB1
C44F  E6 FB            160         INC   *AD1
C451  CA               170         DEX
C452  D0 F6            180         BNE   SYMB1
C454  60               190         RTS
                       200         .EN
```

```
C440 / C455 / 0015
SOURCEFILE IS VOORBEELD 3.SRC
 0  FOUTEN

AD      00FA      AD1     00FB      SYMB1  C44A
```

Uw programma kan er ongeveer zo uitzien. Ten opzichte van het vorige programma is nog een kleine verbetering aangebracht. De sprong terug in regel 180 kan ook naar label SYMB1 in regel 130, omdat het Y-register op dit moment de waarde nul bevat. Het opnieuw laden met nul bij SYMB2 is dus overbodig. Voordat u het programma start op adres $C440 (50240) moet de accumulator geladen worden met de waarde van de betrokken kleuren. Zoals gezegd is het adres van de accumulator 780. Met: POKE 780, 0+16*1 = POKE 780, 16 wordt de achtergrond zwart en worden de gezette punten wit.

Het zetten en wissen van punten is ons volgende onderwerp. Na de drie routines voor het voorbereiden (initiëren) van het hires-scherm komen we nu bij de belangrijkste routine. Met deze routine moeten bepaalde punten aan of uit worden gezet. Er zijn hiervoor vele programmeertechnieken mogelijk, zoals het gebruik van logische bewerkingen. Eerst moeten we weten in welke volgorde de 64000 bits van de 8000 bytes geordend zijn. We bekijken de volgende tabel. De overeenkomst met het normale schermgebruik van 40 karakters op 25 regels wordt zo duidelijk.

	kolom 0	kolom 1	....	kolom 39
	0	8		312
	1	9		313
	2	10		314
Regel 0	3	11	....	315
	4	12		316
	5	13		317
	6	14		318
	7	15		319
	320	328		632
	321	329		633
	322	330		634
Regel 1	323	331	....	635
	324	332		636
	325	333		637
	326	334		638
	327	335		639
...	....	....	....	....
	7680	7688		7992
	7681	7689		7993
	7682	7690		7994
Regel 24	7683	7691		7995
	7684	7692		7996
	7685	7693		7997
	7686	7694		7998
	7687	7695		7999

Het beeldscherm blijft in 25 regels van 40 kolommen verdeeld. Voor ieder hokje staan 8 bytes ter beschikking. De acht bits van een byte staan elk voor een beeldschermpunt. De bit met de hoogste waarde (bit 7) is de meest linkse punt op het scherm.

```
bit     7 6 5 4 3 2 1 0
-----------------------
inhoud  1 0 0 0 1 1 0 0
```

Heeft een byte de inhoud %10001100 = $8C dan betekent dat, dat de eerste, de vijfde en de zesde punt, van linksuit gezien, gezet zijn. Om een betere greep op iedere punt te hebben wordt iedere punt door een horizontale X- en een verticale Y-coördinaat

voorgesteld. De X-coördinaat heeft het bereik 0 t/m 319 en de Y-coördinaat 0 t/m 199. Er moet dus een procedure worden gevonden om iedere punt apart met een X-Y-coördinaat te bereiken. Uitgaande van een bepaalde X en Y moet vastgesteld worden om welke byte en welke bit het gaat. Eerst proberen we dit in BASIC. De X-posities 0 t/m 7, 8 t/m 15, 16 t/m 23 enz. liggen binnen dezelfde byte in het grafiek-RAM. Bij deze getallengroepen zijn de onderste drie bits verschillend terwijl de bovenste vijf gelijk zijn. De onderste drie bits hebben dus voor de bepaling van de byte geen betekenis en kunnen gewist worden. Dit kunnen we bereiken met de AND-functie, die als resultaat een 1 geeft als beide bits 1 zijn. Als een van beide bits van de vergelijking nul is, is het resultaat nul. De volgende AND-bewerking wordt gebruikt:

```
X  AND  %11111000
```

De onderste drie bits worden door deze bewerking gewist, onafhankelijk van de waarde van de X-coördinaat. Probeer maar eens in BASIC:

```
PRINT X AND 248
```

Hoe kan de bytepositie uit de Y-coördinaat worden bepaald? Als de waarde van de Y-coördinaat 0 t/m 7 is, kan deze waarde direkt toegevoegd worden aan de door de X-coördinaat bepaalde byte. Is de Y-coördinaat groter dan 7 dan moet 40 keer de waarde eraan toegevoegd worden, waarbij weer de onderste drie bits buiten beschouwing blijven. Bij de omzetting in machinetaal moeten we erop bedacht zijn dat de X-coördinaat de waarde van 255 kan overschrijden waardoor deze waarde twee bytes nodig heeft. De X-coördinaat wordt in een XL- en een XH-gedeelte gesplitst. De waarde van XH is dan 0 of 1, al naar gelang de waarde kleiner dan 256 of groter dan 255 is. De maximale waarde is 319 = $13F. De byte wordt bepaald door:

```
PRINT XH*256+(XL AND 248)+(Y AND 7)+40*(Y AND 248)
```

Deze formule moet in machinetaal worden omgezet. Eerst stellen we vast in welke registers de waarden moeten worden overgedragen.

```
Y  =>  Register Y-coördinaat
X  =>  Register XH-coördinaat
A  =>  XL-coördinaat
```

De Y-coördinaat wordt in het Y-register opgeslagen terwijl voor de X-coördinaat de twee bytes van de andere registers gebruikt worden. Voor de som is een tweede 16-bits opslagplaats nodig.

```
100 XL = $FA
110 XH = $FB
120 SUML = $FC
130 SUMH = $FD
140 *= $C460
150 STA *XL
160 STX *XH ; X-COÖRDINAAT VASTLEGGEN
170 TYA ; Y-COÖRDINAAT
180 AND #$F8 ;
```

De term 40*(Y AND 248) proberen we nu te berekenen. Omdat de processor geen instructies kent voor de vermenigvuldiging moeten we het anders doen. Door het opschuiven naar links van elke bit wordt de inhoud verdubbeld zoals. Het getal 40 kan in verdubbelingen opgesplitst worden:

```
A * 40  =>  A *2 *2 +A *2 *2 *2
```

Eerst krijgen we de dubbele waarde, daarna de viervoudige en door de oorspronkelijke waarde erbij op te tellen de vijfvoudige. Drie verdubbelingen leveren uiteindelijk de veertigvoudige waarde op.

```
190 STA *$FE  ; WAARDE ONTHOUDEN
200 STA *SUML
210 LDA #0
220 STA *SUMH ; HI-BYTE WISSEN
230 ASL *SUML ; VERSCHUIVING NAAR LINKS
240 ROL *SUMH ; BIJ HI-BYTE LETTEN OP CARRY
250 ASL *SUML
260 ROL *SUMH ; NOGMAALS VERSCHUIVEN
```

Aangezien de verschuiving plaatsvindt over 16 bits moet bij de hi-byte de ROL-instructie worden gebruikt, omdat hier een eventueel overschot (van het eruitgeschoven bit 7) in de hi-byte moet worden overgenomen. Nu kan de optelling van de oorspronkelijke waarden plaatsvinden.

```
270 CLC ; OVERSCHOT WISSEN
280 LDA *SUML
290 ADC *$FE
300 STA *SUML ; RESULTAAT OPSLAAN
310 LDA *SUMH
320 ADC #0
330 STA *SUMH
```

De optelling met nul bij de SUMH gebeurt op dezelfde grond als boven: als bij de optelling van SUML een overschot in de carry staat wordt deze meegeteld. Het resultaat moet nu nog drie maal verdubbeld worden.

```
340 ASL *SUML
350 ROL *SUMH
360 ASL *SUML
370 ROL *SUMH ; DRIE KEER VERDUBBELEN
380 ASL *SUML
390 ROL *SUMH
```

Daarmee is de eerste en moeilijkste term al gedaan. Vervolgens gaan we de tweede term optellen.

```
400 TYA ; Y-COORDINAAT IN DE ACCUMULATOR
410 AND #7
420 CLC
430 ADC *SUML
440 STA *SUML
450 LDA *SUMH
460 ADC #0
470 STA *SUMH
```

Ook hier is weer een 16-bits optelling met meerekening van de carry uitgevoerd. De optelling van de X-waarde AND 248 gaat als volgt:

```
480 CLC
490 LDA *XL
500 AND #$F8
510 ADC *SUML
520 STA *SUML
530 LDA *XH
540 ADC *SUMH
550 STA *SUMH
```

Omdat het grafiekgeheugen niet op adres 0 begint maar bij $E000 moet deze waarde nog toegevoegd worden:

```
560 CLC
570 LDA #<$E000
580 ADC *SUML
590 STA *SUML
600 LDA #>$E000
610 ADC *SUMH
620 STA *SUMH
```

Eindelijk hebben we dan in SUML/SUMH het adres van de opgeroepen byte uit het grafiek-RAM. Tenslotte moet nog de bitpositie binnen de byte worden bepaald. Dit kan met de onderste drie bits gedaan worden. De volgende tabel geeft de bitpositie aan:

```
X          bit
0    =>     7
1    =>     6
2    =>     5
3    =>     4
4    =>     3
5    =>     2
6    =>     1
7    =>     0
```

De bitwaarden zijn omgedraaid. Hiervoor kan de EOR (exclusive-or)-instructie gebruikt worden.

```
630 LDA *XL
640 AND #7
650 EOR #7
```

Hiermee hebben we het byte-adres en het bitnummer verkregen. Uit het nummer moet moet de bij deze positie behorende waarde bepaald worden. De waarde 1 (bitpositie 0) wordt zo vaak naar links verschoven tot de positie wordt aangegeven.

```
660 TAX ; POSITIE NAAR X
670 LDA #1
680 SHIFT DEX
690 BMI OK
700 ASL ; BIT NAAR LINKS VERSCHUIVEN
710 BNE SHIFT
720 OK ...
```

Het aantal keren dat naar links is verschoven wordt door de teller in het X-register aangegeven. In regel 690 wordt getest of de teller al negatief is. Is dit het geval dan zijn we klaar, anders wordt 1 naar links geschoven. De sprong in regel 710 wordt steeds uitgevoerd waar het resultaat van het verschuiven altijd ongelijk is aan nul. Nu hebben we in de accumulator de bitpositie en kunnen die overdragen aan het berekende adres. Om te voorkomen dat een andere gezette punt gewist wordt moet de betrokken bit toegevoegd worden. Daarvoor kennen we de logische bewerking OR. Met de OR-instructie wordt de oorspronkelijke waarde toegevoegd en het resultaat wordt weer in het betrokken adres gezet. Hiervoor wordt weer de indirekte geïndexeerde adresseerwijze gebruikt. De adreswijzer op de zeropage is al berekend. Aangezien deze adresseerwijze alleen maar gaat via het Y-register, moet dit register eerst gewist worden.

```
720 OK LDY #0
730 ORA (SUML),Y
740 STA (SUML),Y
750 RTS
```

De nieuwe punt wordt zo aangezet en we zijn klaar. Een kleinigheid moeten we nog in beschouwing nemen. Met de OR-functie in regel 730 wordt de inhoud van een adres in het gebied van $E000 t/m $FFFF gelezen. De Commodore 64 geeft normaal de inhoud van het ROM-gebied, tenzij we hem meedelen dat uit het RAM-geheugen de waarde van een adres gehaald moet worden. Op adres 1 kunnen we het ROM-geheugen vervangen door het onderliggende RAM-geheugen. Tijdens het gebruik van dit RAM-gebied moeten interrupts verhinderd worden omdat de interruptroutines zich op dit moment in het niet bereikbare ROM-gebied bevinden. Aan het eind wordt de oorspronkelijke geheugenindeling weer hersteld en zijn interrupts weer mogelijk.

```
730 LDX #$34 ; INSCHAKELING RAM
740 SEI ; VERHINDERING INTERRUPTS
750 STX *1
760 ORA (SUML),Y
770 STA (SUML),Y ; PUNT ZETTEN
780 LDX #$37 ; INSCHAKELING ROM
790 STX *1
800 CLI ; INTERRUPTS MOGELIJK
810 RTS
820 .EN
```

De assemblerlisting:

```
00FA                100 XL    =    $FA
00FB                110 XH    =    $FB
00FC                120 SUML  =    $FC
00FD                130 SUMH  =    $FD
C460                140       *=    $C460
C460  08            145       PHP
C461  85 FA         150       STA   *XL
C463  86 FB         160       STX   *XH      ; X-COORDINAAT MERKEN
C465  98            170       TYA            ; Y-COORDINAAT
C466  29 F8         180       AND   #$F8
C468  85 FE         190       STA   *$FE
C46A  85 FC         200       STA   *SUML
C46C  A9 00         210       LDA   #0
C46E  85 FD         220       STA   *SUMH
C470  06 FC         230       ASL   *SUML
C472  26 FD         240       ROL   *SUMH
C474  06 FC         250       ASL   *SUML
C476  26 FD         260       ROL   *SUMH
```

```
C478  18        270          CLC           ; OVERSCHOT WISSEN
C479  A5 FC     280          LDA  *SUML
C47B  65 FE     290          ADC  *$FE
C47D  85 FC     300          STA  *SUML
C47F  A5 FD     310          LDA  *SUMH
C481  69 00     320          ADC  #0
C483  85 FD     330          STA  *SUMH
C485  06 FC     340          ASL  *SUML
C487  26 FD     350          ROL  *SUMH
C489  06 FC     360          ASL  *SUML
C48B  26 FD     370          ROL  *SUMH
C48D  06 FC     380          ASL  *SUML
C48F  26 FD     390          ROL  *SUMH
C491  98        400          TYA           ; Y-COORDINAAT
C492  29 07     410          AND  #7
C494  18        420          CLC
C495  65 FC     430          ADC  *SUML
C497  85 FC     440          STA  *SUML
C499  A5 FD     450          LDA  *SUMH
C49B  69 00     460          ADC  #0
C49D  85 FD     470          STA  *SUMH
C49F  18        480          CLC
C4A0  A5 FA     490          LDA  *XL
C4A2  29 F8     500          AND  #$F8
C4A4  65 FC     510          ADC  *SUML
C4A6  85 FC     520          STA  *SUML
C4A8  A5 FB     530          LDA  *XH
C4AA  65 FD     540          ADC  *SUMH
C4AC  85 FD     550          STA  *SUMH
C4AE  18        560          CLC
C4AF  A9 00     570          LDA  #<$E000
C4B1  65 FC     580          ADC  *SUML
C4B3  85 FC     590          STA  *SUML
C4B5  A9 E0     600          LDA  #>$E000
C4B7  65 FD     610          ADC  *SUMH
C4B9  85 FD     620          STA  *SUMH
C4BB  A5 FA     630          LDA  *XL
C4BD  29 07     640          AND  #7
C4BF  49 07     650          EOR  #7
C4C1  AA        660          TAX
C4C2  A9 01     670          LDA  #1
C4C4  CA        680 SHIFT    DEX
C4C5  30 03     690          BMI  OK
C4C7  0A        700          ASL  A
C4C8  D0 FA     710          BNE  SHIFT
C4CA  A0 00     720 OK       LDY  #0
C4CC  A2 34     730          LDX  #$34
C4CE  28        735          PLP
C4CF  78        740          SEI
```

```
C4D0  86 01      750          STX   *1
C4D2  90 04      760          BCC   WIS
C4D4  11 FC      770          ORA   (SUML),Y
C4D6  B0 04      780          BCS   OK2
C4D8  49 FF      790 WIS      EOR   #$FF
C4DA  31 FC      800          AND   (SUML),Y
C4DC  91 FC      810 OK2      STA   (SUML),Y
C4DE  A2 37      820          LDX   #$37
C4E0  86 01      830          STX   *1
C4E2  58         840          CLI
C4E3  60         850          RTS
                 860          .EN
```

De routines worden samengebracht in het volgende BASIC-programma:

```
100 SYS 50176 : REM GRAFIEK INSCHAKELEN
110 SYS 50208 : REM GRAFIEK WISSEN
120 POKE 780,16 : REM ZWART/WIT
130 SYS 50240 : REM KLEUR INITIALISEREN
140 FOR X = 0 TO 319
150 POKE 780, X AND 255 : REM X-LO
160 POKE 781, X / 256   : REM X-HI
170 POKE 782, X * 0.625 : REM Y
180 SYS 50272 : REM PUNT PLAATSEN
190 NEXT
200 GET A$ : IF A$ = "" THEN 200
210 SYS 50192 : REM UITSCHAKELEN
```

Als u dit programma RUNt wordt van links boven naar rechts onder een lijn getrokken. Door op een willekeurige toets te drukken komt u weer in de normale karaktermodus terug. Een gezette punt kan weer worden gewist door de betrokken bit nul te maken. Eerst bekijken we nog eens hoe de punt werd gezet.

```
Oorspronkelijke byte-inhoud     % 01001000
Punt zetten in bit 4            % 00010000
                                ----------
Resultaat met ORA               % 01011000
```

Door de OR-bewerking werd de punt gezet. Als dezelfde bit gewist moet worden, gebruiken we de AND-bewerking:

```
Oorspronkelijke byte-inhoud     % 01011000
Punt wissen in bit 4            % 00010000
                                ----------
Resultaat met AND               % 00010000
```

Hier is iets mis gegaan: alle bits behalve die van de te wissen punt zijn gewist. Bij de AND-bewerking moet niet de te wissen bit gezet zijn, maar alle andere. Dit wordt gedaan met de EOR-bewerking:

```
te wissen punt              % 00010000
alle bits omdraaien         % 11111111
                            ----------
resultaat met EOR           % 11101111
```

Nu zijn alle bits met uitzondering van de te wissen punt gezet en de AND-bewerking met de oorspronkelijke byte-inhoud geeft het gewenste resultaat:

```
oorspronkelijke byte-inhoud % 01001000
resultaat EOR-bewerking     % 11101111
                            ----------
resultaat met AND           % 01001000
```

De mogelijkheid om punten te wissen moet nog in het programma worden ingebouwd, zodat beslist kan worden of een bepaalde punt gezet, dan wel gewist moet worden. De carryflag laten we hierover beslissen. Is de carryflag gewist, dan wordt ook de punt gewist, is de carryflag gezet dan wordt ook de punt gezet. De toestand van de carryflag moet aan het begin vastgesteld worden. De instructie PHP wordt gebruikt om de inhoud van het statusregister tussentijds op te slaan in de stack. De instructie wordt in regel 140 tussengevoegd. In regel 735 wordt met PLP het statusregister weer van de stack gehaald. De rest van het programma ziet er dan zo uit:

```
760 BCC WIS
770 ORA (SUML),Y
780 BCS OK2
790 WIS EOR #$FF ; OMDRAAIEN
800 AND (SUM1),Y
810 OK2 STA (SUML)
820 LDX #$37
830 STX *1
840 CLI
850 RTS
860 .EN
```

Als de carryflag gewist is wordt naar regel 790 gesprongen. Daar worden de bits met EOR #$FF omgedraaid, vervolgens wordt de AND-bewerking uitgevoerd en het resultaat weggezet. Was de carryflag echter gezet dan zou net als eerst de bit gezet zijn en er zou gesprongen zijn naar de plaats waar de nieuwe waarde weer wordt weggezet. Daarvoor gebruiken we de BCS-instructie omdat de

carryflag door de ORA-bewerking niet wordt beïnvloed.

```
00FA                100 XL      =    $FA
00FB                110 XH      =    $FB
00FC                120 SUML    =    $FC
00FD                130 SUMH    =    $FD
C460                140         *=    $C460
C460  08            145         PHP
C461  85 FA         150         STA   *XL
C463  86 FB         160         STX   *XH ; X-COORDINAAT MERKEN
C465  98            170         TYA           ; Y-COORDINAAT
C466  29 F8         180         AND   #$F8
C468  85 FE         190         STA   *$FE
C46A  85 FC         200         STA   *SUML
C46C  A9 00         210         LDA   #0
C46E  85 FD         220         STA   *SUMH
C470  06 FC         230         ASL   *SUML
C472  26 FD         240         ROL   *SUMH
C474  06 FC         250         ASL   *SUML
C476  26 FD         260         ROL   *SUMH
C478  18            270         CLC           ; OVERSCHOT WISSEN
C479  A5 FC         280         LDA   *SUML
C47B  65 FE         290         ADC   *$FE
C47D  85 FC         300         STA   *SUML
C47F  A5 FD         310         LDA   *SUMH
C481  69 00         320         ADC   #0
C483  85 FD         330         STA   *SUMH
C485  06 FC         340         ASL   *SUML
C487  26 FD         350         ROL   *SUMH
C489  06 FC         360         ASL   *SUML
C48B  26 FD         370         ROL   *SUMH
C48D  06 FC         380         ASL   *SUML
C48F  26 FD         390         ROL   *SUMH
C491  98            400         TYA           ; Y-COORDINAAT
C492  29 07         410         AND   #7
C494  18            420         CLC
C495  65 FC         430         ADC   *SUML
C497  85 FC         440         STA   *SUML
C499  A5 FD         450         LDA   *SUMH
C49B  69 00         460         ADC   #0
C49D  85 FD         470         STA   *SUMH
C49F  18            480         CLC
C4A0  A5 FA         490         LDA   *XL
C4A2  29 F8         500         AND   #$F8
C4A4  65 FC         510         ADC   *SUML
C4A6  85 FC         520         STA   *SUML
C4A8  A5 FB         530         LDA   *XH
C4AA  65 FD         540         ADC   *SUMH
C4AC  85 FD         550         STA   *SUMH
```

```
C4AE  18        560         CLC
C4AF  A9 00     570         LDA   #<$E000
C4B1  65 FC     580         ADC   *SUML
C4B3  85 FC     590         STA   *SUML
C4B5  A9 E0     600         LDA   #>$E000
C4B7  65 FD     610         ADC   *SUMH
C4B9  85 FD     620         STA   *SUMH
C4BB  A5 FA     630         LDA   *XL
C4BD  29 07     640         AND   #7
C4BF  49 07     650         EOR   #7
C4C1  AA        660         TAX
C4C2  A9 01     670         LDA   #1
C4C4  CA        680 SHIFT   DEX
C4C5  30 03     690         BMI   OK
C4C7  0A        700         ASL   A
C4C8  D0 FA     710         BNE   SHIFT
C4CA  A0 00     720 OK      LDY   #0
C4CC  A2 34     730         LDX   #$34
C4CE  28        735         PLP
C4CF  78        740         SEI
C4D0  86 01     750         STX   *1
C4D2  90 04     760         BCC   WIS
C4D4  11 FC     770         ORA   (SUML),Y
C4D6  B0 04     780         BCS   OK2
C4D8  49 FF     790 WIS     EOR   #$FF
C4DA  31 FC     800         AND   (SUML),Y
C4DC  91 FC     810 OK2     STA   (SUML),Y
C4DE  A2 37     820         LDX   #$37
C4E0  86 01     830         STX   *1
C4E2  58        840         CLI
C4E3  60        850         RTS
                860         .EN
```

Het voorbeeld wordt enigszins verbeterd.

```
100 SYS 50176 : REM GRAFIEK INSCHAKELEN
110 SYS 50208 : REM GRAFIEK WISSEN
120 POKE 780,16
130 SYS 50240 : REM KLEUR INSTELLEN
140 I = 1
150 FOR X = 0 TO 319
160 POKE 780, X AND 255 : REM X-LO
170 POKE 781, X / 256   : REM X-HI
180 POKE 782, X * 0.625 : REM Y
190 POKE 783, I : REM INSTELLEN / WISSEN
200 SYS 50272 : NEXT
210 GET A$ : IF A$ = "" THEN I = I-1 : GOTO 150
220 SYS 50192 : REM GRAFIEK UITSCHAKELEN
```

Met dit programma wordt een diagonale lijn over het scherm getrokken, die vervolgens weer gewist wordt. Met de carryflag kan aangegeven worden of een punt gezet dan wel gewist wordt. De carryflag wordt samen met de andere flags weggezet in adres 783. De carryflag staat in bit 0, die nul of een is. Het programma kan door een druk op een toets worden onderbroken. De oorspronkelijke beeldscherminhoud blijft daarbij, evenals het grafiekscherm, behouden. Wilt u later op de grafische mode omschakelijk dan kunt u de wisroutine weglaten. We gaan nu nog wat experimenteren met de kleurenpresentatie.

```
100 SYS 50176 : REM GRAFIEK INSCHAKELEN
110 SYS 50208 : REM GRAFIEK WISSEN
120 POKE 780,16
130 SYS 50240 : REM KLEUR INSTELLEN
140 REM
150 FOR X = 70 TO 150 : FOR Y =X TO 199
160 POKE 780, X  : REM X-LO
170 POKE 781, 0  : REM X-HI
180 POKE 782, Y  : REM Y
190 POKE 783, I : REM INSTELLEN
200 SYS 50272 : NEXT : NEXT
210 FOR C = 0 TO 255
220 FOR I = 1 TO 500 : NEXT
230 POKE 780, C
240 SYS 50240 : REM KLEUR
250 NEXT
260 SYS 50192 : REM GRAFIEK UITSCHAKELEN
```

Eerst wordt een figuur getekend die daarna in alle 256 mogelijke kleurencombinaties op het scherm komt.

In het volgende hoofdstuk zullen we ons met een andere belangrijke programmeertechniek bezighouden, het gebruik van subroutines. Met deze kennis en nog wat oefening moet u zeker in staat zijn ook een hardcopy-routine voor de hires-grafiek met een printer te schrijven. Eerst moet u nagaan hoe uw printer grafische gegevens verwerkt. Meestal wordt een kolom van acht boven elkaar liggende punten door een byte overgedragen, maar dit verschilt van printer tot printer. Werkt uw printer volgens bovenstaand principe dan kunt u de grafiekdata niet simpel byte voor byte uit het grafiek-RAM naar de printer sturen, maar moet u telkens uit 8 op elkaar volgende bytes de betrokken bits door middel van logische bewerkingen isoleren om zo de byte-inhoud te kunnen bepalen die naar de printer gestuurd moet worden. Als dat ingewikkeld wordt, kunt u een stroomdiagram als hulpmiddel gebruiken. Met wat geduld zal het u zeker lukken. De werking van uw programma kunt u ook nagaan met de stap-voor-stap simulator uit dit boek.

8 BASIC-UITBREIDINGEN EN ROUTINES VAN HET BEDRIJFSSYSTEEM

In het vorige hoofdstuk zijn bij de grafiekroutines de parameters door middel van POKE-instructies aan de verschillende registers overgedragen. Er is een elegantere methode, die bovendien zeer eenvoudig te programmeren is. De parameters worden op dezelfde wijze overgedragen als de BASIC-interpreter dat doet. Als voorbeeld nemen we de POKE-instructie:

```
POKE A, B
```

Zoals u weet kunnen voor A en B willekeurig ingewikkelde uitdrukkingen, constanten of geïndexeerde variabelen worden gebruikt, bijvoorbeeld:

```
POKE A(1000)/750*INT(X%/9), EXP(ABS(SIN(3*A)))+2
```

Voor de behandeling van een dergelijke uitdrukking bezit de BASIC-interpreter een universele routine die dit overneemt. Deze routine kunnen we voor ons doel gebruiken en het werk aan de BASIC-interpreter overlaten. Bovendien wordt steeds getest of geen gebiedsoverschrijding plaatsvindt. Bij de POKE-routine bijvoorbeeld wordt bij het inlezen van de parameter A getest of de waarde ligt tussen 0 en 65535 (16-bits waarde). Als dit niet het geval is, verschijnt de foutmelding: ILLEGAL QUANTITY ERROR op het scherm. De tweede parameter B wordt getest op het gebied 0 t/m 255 en bij overschrijding wordt ook hier de foutmelding gegeven. Hoe kunnen deze routines voor ons doel gebruikt worden? Eerst houden we ons bezig met een tot dusverre niet gebruikte programmeertechniek, de subroutinetechniek. In BASIC hebt u daar zeker al gebruik van gemaakt. De instructies zijn:

```
GOSUB en RETURN
```

De GOSUB-routine springt naar een aangegeven regel. In tegenstelling tot de GOTO-instructie wordt onthouden vanaf welke regel gesprongen werd. Als de subroutine is afgewerkt wordt door de RETURN-instructie dit adres opgehaald en daarna teruggesprongen. De 6510-processor heeft twee instructies beschikbaar voor dit doel:

```
JSR en RTS
```

De oproep om naar de subroutine te springen gaat met JSR (jump to subroutine) en de terugkeer naar het hoofdprogramma met RTS (return from subroutine). Deze instructies worden net als in

BASIC gebruikt om programma-onderdelen die vaak worden gebruikt gemakkelijk te kunnen aanroepen. Bovendien kost dat minder geheugenruimte. In machinetaal moet u erop letten dat de subroutines zo veel mogelijk algemeen gebruikt kunnen worden. De constanten moeten voor alle subroutines dezelfde zijn. Zijn de constanten verschillend dan moet u handig gebruik maken van de adresseermogelijkheden om toch een universele subroutine te gebruiken. Als de processor de instructie JSR bereikt wordt het adres waarnaar gesprongen moet worden eerst opgehaald uit de stack. Voor de stack zijn de geheugenplaatsen $0100 t/m $01FF voor de opslag van adressen gereserveerd. Bij de oproep naar een subroutine worden de twee bytes van het actuele adres in de stack gezet en bij de sprong terug wordt dit adres opgehaald. Opdat de processor weet op welke plaats in de stack het betreffende adres staat, wordt gebruik gemaakt van een extra register, de stackpointer (SP). Dit register dient als wijzer naar het actuele adres. We bekijken wat er precies gebeurt:

```
C000  20 00 C1  JSR $C100
C003  ....


C100  60        RTS
```

Als het programma gestart wordt op adres $C000, wordt gesprongen naar adres $C100. In het voorbeeld staat daar meteen RTS en de processor springt terug naar de instructieplaats die volgt op JSR, in ons geval $C003. We gaan na wat er op de stack en met de stackpointer gebeurt:

adres	instructie	stackpointer	stack	
$C000	JSR $C100	$F9	$01F9	XX

De stackpointer heeft bijvoorbeeld de waarde $F9. Op het stackadres $01F9 staan gegevens van vorige bewerkingen. Als de processor bij de JSR-instructie komt, wordt de inhoud van de instructieteller (PC) met 2 verhoogd en gesplitst in een lo-byte en een hi-byte. De hi-byte wordt weggezet op het adres waarnaar de stackpointer wijst, de stackpointer wordt dan met 1 verminderd.

adres	instructie	stackpointer	stack	
$C000	JSR $C100	$F8	$01F9	C0
			$01F8	XX

Vervolgens wordt de lo-byte van het adres op de stack gezet en de stackpointer met 1 verminderd. De instructieteller wordt op het

adres van de subroutine gezet.

adres	instructie	stackpointer	stack	
$C100	RTS	$F7	$01F9	C0
			$01F8	02
			$01F7	XX

De instructieteller is op de stack gezet en de stackpointer is met 2 verminderd. De stackpointer wijst altijd naar de eerstvolgende vrije positie in de stack. We gaan nu na wat er bij de RTS-instructie gebeurt. Het boven beschreven proces verloopt dan in omgekeerde volgorde. Eerst wordt de stackpointer verhoogd en dan de waarde van de stack gehaald.

adres	instructie	stackpointer	stack	
$C100	RTS	$F8	$01F9	C0
			$01F8	02
			$01F7	XX

Deze waarde wordt als de lo-byte van de instructieteller beschouwd, de stackpointer wordt weer verhoogd en de volgende waarde wordt van de stack gehaald.

adres	instructie	stackpointer	stack	
$C100	RTS	$F9	$01F9	C0
			$01F8	02
			$01F7	XX

Zo worden na elkaar $02 en $C0 van de stack gehaald. Dit wordt als instructieteller $C002 geïnterpreteerd. De instructieteller wordt dan met 1 verhoogd en de volgende instructie kan opgehaald worden.

adres	instructie	stackppointer	stack	
$C003	...	$F9	$01F9	C0
			$01F8	02
			$01F7	XX

Let erop dat de stackpointer na de terugkeer uit de subroutine weer zijn oorspronkelijke waarde heeft teruggekregen. Met deze techniek is het ook mogelijk om subriutines binnen subroutines te maken. Wordt door een subroutine een tweede subroutine opgeroepen dan wordt ook hier het adres waarnaar teruggesprongen moet worden, onthouden. Bij de RTS-instructie wordt dan weer naar dit adres teruggesprongen. Zo kunnen meerdere subroutines genesteld worden.

De 6510-processor voert het proces met de subroutines automatisch uit en u hoeft er zich verder niet om te bekommeren. De processor biedt ook de mogelijkheid om tussentijds de inhoud van de accumulator of het statusregister op te slaan en daar ook weer vanaf te halen. Daarvoor worden de instructies PHA en PLA voor de accumulator en PHP en PLP voor het statusregister gebruikt. Ook bij deze instructie wordt de stackpointer automatisch mee veranderd: bij het schrijven wordt hij verlaagd en bij het lezen verhoogd.

```
PHA  ;  inhoud accumulator op de stack
TYA  ;  zet Y in A
PHA  ;  Y-register op de stack
TXA  ;  zet X in A
PHA  ;  X-register op de stack
...
PLA
TAX  ;  ophalen X-register
PLA
TAY  ;  ophalen Y-register
PLA  ;  ophalen accumulator
```

Het X- en Y-register kunnen niet direkt op de stack worden gezet, eerst moet de inhoud aan de accumulator worden doorgegeven en van daaruit op de stack worden gezet. Het ophalen van de registerinhouden gebeurt in omgekeerde volgorde volgens het principe last in-first out. Het opslaan van de registerinhouden vindt plaats als door de subroutine de registerinhoud wordt veranderd. Later kan de oorspronkelijke inhoud weer uit de stack worden gehaald. Met de stap-voor-stap simulator kan de werking van de stack goed gedemonstreerd worden, omdat bij elke stap de registerinhouden zichtbaar zijn.

We keren terug naar de routines van de BASIC-interpreter die bij het inlezen van parameters kunnen worden gebruikt. De routine GETBYT leest een uitdrukking uit de BASIC-tekst, test of de waarde tussen 0 en 255 ligt en zet deze waarde in het X-register. De routine start op het adres $B79E. Een andere routine maakt het mogelijk om 16-bits waarden (0 t/m 65535) in te lezen. Daarvoor zijn twee routines beschikbaar. De routine met de naam FRMNUM (formule numeriek) haalt een willekeurige numerieke uitdrukking. Met GETADR kan deze waarde getest worden op het bereik 0 t/m 65535. Ligt de waarde hierbinnen, dan wordt in adres $0014 de lo-byte en in adres $0015 de hi-byte gezet. De adressen van deze routine vindt u hieronder. Als meerdere parameters moeten worden doorgegeven, moeten deze van elkaar worden gescheiden. Daar wordt, net als in BASIC, de komma voor gebruikt. Er is een BASIC-routine beschikbaar die test of een komma al dan niet voorkomt. Deze routine heeft de naam CKHCOM. Vaak moet direkt een

karakter uit de BASIC-tekst worden gelezen. Ook daarvoor zijn geschikte programma-onderdelen beschikbaar. De routine CHRGOT zet dat karakter in de accumulator waarnaar de instructieteller wijst, terwijl met CHRGET de wijzer voor het lezen met 1 wordt verhoogd, zodat het volgende karakter kan worden gehaald. Deze routines zetten ook bepaalde flags, die nadere informatie over de gelezen karakters geven. Een gezette zeroflag betekent dat een nulbyte (in BASIC-programma's einde regel) of een dubbele punt gelezen is, als aanduiding dat het statement is bereikt. Als een cijfer wordt gelezen, wordt de carryflag gewist. De adressen:

```
GETBYT  $B79E
FRMNUM  $AD8A
GETADR  $B7F7
CHKCOM  $AEFD
CHRGOT  $0079
CHRGET  $0073
```

Als u na elkaar een 16-bits en een 8-bits waarde wilt lezen, zoals bij de POKE-instructie nodig is, kunt u de routine GETPAR gebruiken die ook test op de scheidende komma. In GETPAR worden na elkaar FRMNUM, GETADR, CHKCOM en GETBYT opgeroepen.

```
GETPAR  $B7EB
```

Deze routine kan bij de grafiekroutines toegepast worden. Daar kan de waarde van de X-coördinaat een 16-bits waarde zijn (0 t/m 319) en van de Y-coördinaat(0 t/m 199) is een 8-bits waarde voldoende. Overschrijden de waarden 65535 resppectievelijk 255 dan geeft de BASIC-interpreter ILLEGAL QUANTITY ERROR. Natuurlijk kan ook achteraf op deze grenzen worden getest en eventueel een foutmelding worden gegeven. Een sprong naar adres $B248 is daarvoor voldoende. Als deze routines voor de overdracht van parameters worden gebruikt kan de oproep er zo uitzien:

```
SYS 50240,X,Y
```

Met deze methode zijn de POKE's overbodig en wordt het geheel overzichtelijker. Dit passen we toe op de puntzetroutine:

```
100 JSR CHKCOM ; VOLGT EEN KOMMA ?
110 JSR GETPAR ; COORDINATEN HALEN
120 STA YCOOR  ; Y-COORDINAAT MERKEN
130 LDA #14
140 STA XL     ; X-COORDINAAT LO
150 LDA #15
160 STA XH
```

Eerst wordt getest op een komma die het SYS-adres scheidt van de X-coördinaten, dan gebruiken we de routine die de twee parameters haalt. De 8-bits waarde van de Y-coördinaat staat in het X-register en wordt op adres YCOOR weggezet. De 16-bits waarde met de X-coördinaat wordt uit de adressen $14 en $15 gehaald en weggezet op XL en XH. Vervolgens moet getest worden of de coördinaten ook binnen het toegestane gebied liggen. De Y-coördinaat wordt vergeleken met 200. Is de waarde kleiner dan is het in orde, anders moet gesprongen worden naar $B248 voor de foutmelding. Bij de X-coördinaat moet een 16-bits vergelijking worden gemaakt.

```
170 CPX #200 ; Y >= 200 ?
180 BCC OK
190 FOUT JMP ILLIGAL
200 OK LDA XH
210 CMP #>320
220 BCC OK1
230 LDA XL
240 CMP #<320
250 BCS FOUT
260 OK1 ...
```

Eerst wordt de inhoud van het X-register waarin de Y-coördinaat staat met 200 vergeleken. Als de Y-coördinaat kleiner is, wordt de carryflag gewist en wordt naar label OK gesprongen. Is dat niet het geval dan wordt de carryflag gezet en wordt er niet gesprongen. We bereiken dan de aanduiding FOUT en springen naar de uitvoerroutine op adres $B248. Nadat de Y-coördinaat in orde is bevonden wordt de X-coördinaat getest. De accumulator wordt met de hi-byte van de waarde geladen en vergeleken met de hi-byte van 320. De vergelijking vindt weer plaats en er wordt gesprongen naar OK1 (in orde) of naar FOUT. Als de hi-byte gelijk is moet nog getest worden op de lo-byte. Daarvoor wordt de lo-byte in de accumulator gezet en vergeleken met 320. De sprongen verlopen weer op dezelfde wijze.

indirect-indexed	$D1
indexed-indirect	$C1

8.1 In- en uitvoerroutines

Tot dusverre is nog niet besproken hoe de randapparaten moeten worden aangesproken of hoe karakters van het toetsenbord op het scherm komen. De instructies in BASIC zijn onder meer:

OPEN
CMD
PRINT#
INPUT#
CLOSE

Bij het programmeren in machinetaal gaan we op dezelfde manier te werk. Het operating system beschikt voor deze zaken over routines die hetzelfde doen als bovenstaande BASIC-instructies. Deze routines hebt u nodig voor de uitwisseling van gegevens met de randapparaten.

OPEN

Deze routine heeft drie parameters nodig: logisch filenummer, nummer randapparaat en secundair adres en eventueel een filenaam. Deze parameters worden door de routine SETFLS en SETNAM gezet. De OPEN-routine heeft zelf geen parameters nodig, maar roept eerst bovenstaande routines op.

SETFLS

De accumulator wordt geladen met het logische filenummer, het X-register met het nummer van het randapparaat en het Y-register met het secundaire adres. Daarna kan deze routine worden opgeroepen.

SETNAM

Hiermee kunt u een filenaam vastleggen. De lengte van de naam zet u in de accumulator (0 als geen filenaam wordt gebruikt) en het startadres van de naam komt in het X-register (lo-byte) en in het Y-register (hi-byte). Op het startadres begint de naaam die u in het geheugen hebt gezet.

PRINT

Deze routine is eenvoudig te gebruiken. U moet het karakter dat u wilt uitvoeren in de accumulator zetten en vervolgens de routine oproepen. Normaal vindt de uitvoer op het beeldscherm plaats.

Wilt u de printer als uitvoerapparaat gebruiken dan moet met de OPEN-routine eerst een file worden geopend en voor de uitvoer de volgende routine worden opgeroepen:

CHKOUT

Deze routine doet hetzelfde als de CMD-instructie in BASIC. Wilt u een karakter in een geopende file uitvoeren dan moet het X-register met het logische filenummer geladen worden en vervolgens de routine CHKOUT worden opgeroepen. Met PRINT worden alle karakters op het randapparaat uitgevoerd totdat ze met de volgende routine CLRCH terugkeren naar de beeldschermuitvoer.

CLRCH

De routine heft de CMD-mode, die u met CHKOUT hebt ingesteld, weer op. Deze routine heeft geen parameter nodig.

INPUT

De routine van het operating system haalt een karakter van het toetsenbord en zet dat in de accumulator. Wilt u data uit een bestand halen, dan moet dat eerst worden geopend. Met de volgende routine CHKIN moet de invoer van deze data worden geactiveerd.

CHKIN

In het X-register moet het logische filenummer gezet worden om de invoer van data van een file in te leiden. Na de oproep wordt de invoer niet meer van het toetsenbord gehaald, maar van de logische file. Deze mode kan weer worden opgeheven met de CLRCH-routine.

CLOSE

Een file kan worden afgesloten met deze routine. Het logische filenummer moet in de accumulator staan. De volgende tabel bevat de oproepadressen van deze routines:

routine	adres
OPEN	$FFC0
SETFLS	$FFBA
SETNAM	$FFBD
PRINT	$FFD2
CHKOUT	$FFC9
CLRCH	$FFCC
INPUT	$FFCF
CHKIN	$FFC6
CLOSE	$FFC3

Voorbeeld: de BASIC-instructies

```
OPEN 1,8,15
PRINT# 1,I
CLOSE 1
```

worden in machinetaal:

```
100 LDA #1 ; LOGISCH FILENUMMER
110 LDX #8 ; RANDAPPARAATNUMMER
120 LDY #15 ; SECUNDAIR ADRES
130 JSR SETFLS
140 LDA #0
150 JSR SETNAM ; GEEN NAAM
160 JSR OPEN ; FILE OPENEN
170 LDX #1 ; LOGISCH FILENUMMER
180 JSR CHKOUT ; UITVOER NAAR FILE
190 LDA #73 ; "I"
200 JSR PRINT
210 JSR CLRCH
220 LDA #1 ; LOGISCH FILENUMMER
230 JSR CLOSE
240 RTS
```

In regel 100 t/m 120 worden de parameters geladen en in regel 130 op de goede plaats weggezet. In regel 140 wordt de lengte (=0) van de filenaam vastgesteld, er wordt geen filenaam gebruikt. In regel 150 wordt deze waarde door middel van de betrokken routine weggezet. Dan volgt in regel 160 de OPEN-routine. In regel 170 wordt de uitvoer op de logische file met nummer 1 gezet. De ASCII-waarde van I is 73 en deze wordt door middel van de PRINT-routine aan de diskdrive doorgegeven. Met de volgende CLRCH vindt de uitvoer weer op het scherm plaats. In regel 220 en 230 wordt de file weer gesloten en wordt tenslotte met RTS weer naar BASIC teruggekeerd. In het volgende voorbeeld wordt het foutkanaal van de diskdrive gelezen en op het scherm gezet. De oplossing is in BASIC:

```
100 OPEN 1,8,15
110 INPUT#1,A,B$,C,D
120 PRINT A; B$; C; D
130 CLOSE 1
```

Omdat de foutmelding direkt moet worden uitgevoerd, zijn in machinetaal geen variabelen nodig. We lezen zolang tot de statusvariabele gelijk is aan 64, daarmee wordt het einde van de foutmelding aangegeven. In BASIC is dit dan:

```
100 OPEN 1,8,15
110 GET#1, A$ : PRINT A$;
120 IF ST <> 64 THEN 110
130 CLOSE 1
```

De statusvariabele van het operating system wordt opgeslagen op adres $0090 (144). De machinetaalversie is:

```
10 OPEN = $FFC0
20 SETFLS = $FFBA
30 SETNAM = $FFBD
40 PRINT = $FFD2
50 CLRCH = $FFCC
60 INPUT = $FFCF
70 CHKIN = $FFC6
80 CLOSE = $FFC3
90 STATUS = $90 ;STATUSVARIABELE
100 LDA #1 ; LOGISCH  FILENUMMER
110 LDX #8 ; NUMMER RANDAPPARAAT
120 LDY #15 ; SECUNDAIR ADRES
130 JSR SETFLS
140 LDA #0
150 JSR SETNAM ; GEEN NAAM
160 JSR OPEN ; FILE OPENEN
170 LDX #1 ; LOGISCH FILENUMMER
180 JSR CHKIN ; INVOER VAN FOUTKANAAL
190 L1 JSR INPUT ; KARAKTER HALEN
200 JSR PRINT ; EN UITVOEREN
210 BIT STATUS ; STATUS TESTEN
220 BVC L1
230 JSR CLRCH
240 LDA #1
250 JSR CLOSE
260 RTS
270 .END
```

De procedure voor het openen van een file kunnen we uit de vorige routine overnemen. De invoer van het foutkanaal vindt plaats door de instructies in regel 170 en 180. Het X-register wordt met het logische filenummer 1 geladen en vervolgens wordt de routine CHKIN opgeroepen. Vanaf dit moment werken alle invoerinstructies op de diskdrive. In regel 190 wordt een karakter van de floppy gehaald en via JSR PRINT in regel 200 op het beeldscherm gezet. Deze uitvoer vindt op het scherm plaats omdat vooraf geen CHKOUT gebruikt is. De volgende stap is het testen van de status. Dit wordt met de BIT-instructie gedaan die tot dusverre nog niet is toegepast. Als het einde van een file wordt bereikt, wordt de status op 64 gezet, dit is 2^6 of %01000000. Bit 6 van deze geheugenplaats wordt gezet. Door middel van de BIT-instructie

wordt bit 6 van de geadresseerde geheugenplaats in de V-flag en bit 7 in de N-flag gekopieerd. Na de BIT-instructie hoeft alleen maar getest te worden of de V-flag al dan niet gezet is. Als de V-flag niet gezet is (ST <> 64) wordt met de BVC-instructie teruggesprongen naar de invoer van de floppy, omdat het einde nog niet bereikt is. Als de V-flag gezet is wordt met JSR CLRCH de invoer weer op de normale mode gezet en vervolgens het kanaal gesloten. Assembleert u het programma en probeert u het vervolgens uit. U moet daarbij bedenken dat de assembler slechts labelnamen van maximaal 5 karakters toestaat.

```
6:    C000          .OPT P1,00  ;ASSEMBL.PROGR.OP PRINTER
10:   FFC0          OPEN    =   $FFC0
20:   FFBA          SETFLS  =   $FFBA
30:   FFBD          SETNAM  =   $FFBD
40:   FFD2          PRINT   =   $FFD2
50:   FFCC          CLRCH   =   $FFCC
60:   FFCF          INPUT   =   $FFCF
70:   FFC6          CHKIN   =   $FFC6
80:   FFC3          CLOSE   =   $FFC3
90:   0090          STATUS  =   $90     ;STATUSVARIABELE
100:  C000 A9 01            LDA  #1     ; LOGISCH FILENR.
110:  C002 A2 08            LDX  #8     ; NUMMER RANDAPP.
120:  C004 A0 0F            LDY  #15    ; SECUNDAIR ADRES
130:  C006 20 BA FF         JSR  SETFLS
140:  C009 A9 00            LDA  #0
150:  C00B 20 BD FF         JSR  SETNAM ; GEEN NAAM
160:  C00E 20 C0 FF         JSR  OPEN   ; FILE OPENEN
170:  C011 A2 01            LDX  #1     ; LOGISCH FILENR.
180:  C013 20 C6 FF         JSR  CHKIN; INVOER FOUTKANAAL
190:  C016 20 CF FF L1      JSR  INPUT  ; KARAKTER HALEN
200:  C019 20 D2 FF         JSR  PRINT  ; EN UITVOEREN
210:  C01C 24 90            BIT  STATUS ; STATUS TESTEN
220:  C01E 50 F6            BVC  L1
230:  C020 20 CC FF         JSR  CLRCH
240:  C023 A9 01            LDA  #1
250:  C025 20 C3 FF         JSR  CLOSE
260:  C028 60               RTS
```

Zoals u uit de listing kunt afleiden is deze keer het programma met een andere assembler voor de Commodore 64, de PROFI-ASS 64, geassembleerd. Deze assembler wordt verderop beschreven. De aanduiding in regel 6 betekent dat de listing in de vooraf geopende logische file 1 moet gaan, terwijl de objectcodes (het geproduceerde machinetaalprogramma) direkt in het geheugen worden gezet. Het testen van deze routine gebeurt met:

```
SYS 49152
```

De foutmelding van de floppy verschijnt op het beeldscherm, bijvoorbeeld:

```
00, OK,00,00
```

9 BASIC-LAADPROGRAMMA

Als een programma in machinetaal is geschreven en met een listing moet worden doorgegeven, is meestal de vraag in welke vorm dat moet gebeuren. Heeft de gebruiker een assembler, dan kan in principe het assembler-sourceprogramma worden doorgegeven. Vaak beschikt men echter niet over een assembler. In dit geval is de volgende methode uitstekend uitvoerbaar.

De Op-codes en de operanden van het machinetaalprogramma worden als een rij DATA ingevoerd. De DATA worden met een BASIC-programma ingelezen en door middel van de POKE-instructie in het geheugen gezet. Voor dit doel is een programma beschikbaar dat dit automatisch doet. Eerst moet het machinetaalprogramma worden geladen en dan het hieronder afgedrukte BASIC-programma. Vervolgens RUNt u het. Het start- en eindadres van het machinetaalprogramma worden opgevraagd en de rest doet het programma voor u. Op de printer wordt een volledig laadprogramma gemaakt, met een automatisch bepaalde checksom. Alle waarden van de DATA worden gesommeerd. Bij het laden van het programma worden weer alle DATA gesommeerd en de daarbij bepaalde som wordt vergeleken met de door het programma vastgestelde. Als beide sommen niet gelijk zijn wordt er op gewezen dat fouten bij de invoer van het programma gemaakt zijn. Veel invoerfouten worden zo vastgesteld. In de praktijk is gebleken dat bij de invoer van lange rijen getallen, die de intyper niets zeggen, vaak fouten optreden. In dit boek heeft u ook een BASIC-programma aangetroffen waar voor de controle een checksom is toegepast.

```
100 OPEN 1,4 : Z=100
110 INPUT"STARTADRES";A
120 INPUT"EINDADRES";E
130 CMD 1:PRINT Z"FOR I ="A "TO" E
140 I=A: Z=Z+10: PRINT"READ X: POKE I,X: S=S+X: NEXT
150 Z=Z+10: N=0: PRINT Z "DATA ";
160 X=PEK(I): S=S+X: PRINT RIGHT$(" "+STR$(X),3);:N=N+1
170 IF I=E THEN PRINT: GOTO 200
180 I=I+1: IF N=12 THEN PRINT: GOTO 150
190 PRINT",";:GOTO160
200 PRINTPRINT Z+10 "IF S <>" S "THEN PRINT"CHR$(34) "FOUT
    IN DATA !!" CHR$(34): END
210 PRINT Z+20 "PRINT " CHR$(34) "OK" CHR$(34)
220 PRINT#1:CLOSE1
```

10 6510 DISASSEMBLER VOOR DE COMMODORE

In dit hoofdstuk presenteren we een disassembler. Machinetaalprogramma's die in het geheugen van de computer staan worden door een disassemblerprogramma omgezet in de symbolische betekenis, die ook bij de invoer van programma's gebruikt wordt. Uit de byte-inhouden $A9, $80 vormt de disassembler LDA #$80. De disassembler wordt met RUN gestart en het begin- en eindadres van het te vertalen programma wordt gevraagd. De uitvoer komt op het beeldscherm. Door het invoegen van de OPEN-instructie, gevolgd door CMD kan de uitvoer ook op de printer worden gezet.

In het kort nog iets over de werkwijze van de disassembler: het programma haalt de inhoud van het startadres op uit het geheugen en interpreteert dit als een Op-code. Deze instructiecode dient als index in een tabel, waaruit het instructiewoord, bijvoorbeeld LDA, en de adresseerwijze gehaald worden. De disassembler weet uit hoeveel bytes de desbetreffende instructie bestaat en in welke vorm de operand moet worden uitgevoerd. Dit gebeurt door middel van subroutines die aan het einde van het programma met de belangrijkste variabelen beschreven zijn.

De disassembler kan voor de omzetting van eigen machinetaalprogramma's worden gebruikt, maar ook voor de disassemblering van gedeelten uit het operating system en de BASIC-interpreter. Met deze programma's kunt u inspiratie opdoen voor uw eigen programma's. Nog beter lenen zich daarvoor van commentaar voorziene listings van het operating system zoals ze bijvoorbeeld in 64 INTERN van DATA BECKER NEDERLANDS* zijn te vinden.

```
100 REM 6510 - DISASSEMBLER
110 DIMMN$(255),AD(255),H$(15)
120 FF=255:HI=256:UL=2↑16:SC=2↑15-1
130 PRINT"■■■■■■■■■■■■■■6510 - DISASSEMBLE
R"
140 FORI=0TO15:READH$(I):NEXT
150 FORI=0TO255:READMN$(I),AD(I):NEXT
160 PRINT"■STARTADRES   :-    $■■■■■■■■■■■"
;:INPUTA$
170 GOSUB540:S=A
180 PRINT"■EINDADRES     :-    $■■■■■■■■■■■
";:INPUTA$:PRINT
190 GOSUB540:E=A
200 FORP=STOE
210 A=P:GOSUB450:REM ADRES
220 PRINT" ";:A=PEEK(P):GOSUB480:PRINT"
";:I=PEEK(P):OP=AD(I)
230 ONOPGOSUB510,520,520,510,530,520,520
,530,530,520,520,520,530
240 PRINT" ";MN$(I)" ";
250 ONOPGOSUB270,280,290,300,310,320,330
,340,350,360,370,380,400
260 NEXTP:GOTO160
270 PRINT:RETURN
280 PRINT"#";:GOSUB490:P=P+1:PRINT:RETUR
N
290 GOSUB490:P=P+1:PRINT:RETURN
300 PRINT" A":RETURN
310 GOSUB420:P=P+2:PRINT:RETURN
320 GOSUB490:P=P+1:PRINT",X":RETURN
330 GOSUB490:P=P+1:PRINT",Y":RETURN
340 GOSUB420:P=P+2:PRINT",X":RETURN
350 GOSUB420:P=P+2:PRINT",Y":RETURN
360 PRINT"(";:GOSUB490:P=P+1:PRINT"),Y":
RETURN
370 PRINT"(";:GOSUB490:P=P+1:PRINT",X)":
RETURN
380 T=PEEK(P+1):Q=T+HI*(T>127)+2+P
390 A=INT(Q/HI)*HI+((Q+(Q>SC)*UL)ANDFF):
PRINT"$";:GOSUB450:P=P+1:PRINT:RETURN
400 PRINT"(";:GOSUB420
410 PRINT")":P=P+2:RETURN
420 PRINT"$";
430 A=PEEK(P+1)+HI*PEEK(P+2)
440 REM HEXADECIMAAL ADRES
450 HB=INT(A/HI):A=A-HI*HB
460 PRINTH$(HB/16)H$(HBAND15);
470 REM HEXADEC.BYTE A
480 PRINTH$(A/16)H$(AAND15);:RETURN
```

```
490 PRINT"$";
500 A=PEEK(P+1):GOTO480
510 PRINT"      ";:RETURN
520 GOSUB500:PRINT"    ";:RETURN
530 GOSUB500:PRINT" ";:A=PEEK(P+2):GOTO4
80
540 IFASC(A$)=42THENEND
550 A=0:FORI=1TO4:V=ASC(RIGHT$(A$,I))-48
:V=V+(V>9)*7:A=A+V*(16↑(I-1)):NEXT:RETURN
1000 DATA0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
1010 DATA"BRK",1,"ORA",11,"???",1
1020 DATA"???",1,"???",1,"ORA",3
1030 DATA"ASL",3,"???",1,"PHP",1
1040 DATA"ORA",2,"ASL",4,"???",1
1050 DATA"???",1,"ORA",5,"ASL",5
1060 DATA"???",1,"BPL",12,"ORA",10
1070 DATA"???",1,"???",1,"???",1
1080 DATA"ORA",6,"ASL",6,"???",1
1090 DATA"CLC",1,"ORA",9,"???",1
1100 DATA"???",1,"???",1,"ORA",8
1110 DATA"ASL",8,"???",1,"JSR",5
1120 DATA"AND",11,"???",1,"???",1
1130 DATA"BIT",3,"AND",3,"ROL",3
1140 DATA"???",1,"PLP",1,"AND",2
1150 DATA"ROL",4,"???",1,"BIT",5
1160 DATA"AND",5,"ROL",5,"???",1
1170 DATA"BMI",12,"AND",10,"???",1
1180 DATA"???",1,"???",1,"AND",6
1190 DATA"ROL",6,"???",1,"SEC",1
1200 DATA"AND",9,"???",1,"???",1
1210 DATA"???",1,"AND",8,"ROL",8
1220 DATA"???",1,"RTI",1,"EOR",11
1230 DATA"???",1,"???",1,"???",1
1240 DATA"EOR",3,"LSR",3,"???",1
1250 DATA"PHA",1,"EOR",2,"LSR",4
1260 DATA"???",1,"JMP",5,"EOR",5
1270 DATA"LSR",5,"???",1,"BVC",12
1280 DATA"EOR",10,"???",1,"???",1
1290 DATA"???",1,"EOR",6,"LSR",6
1300 DATA"???",1,"CLI",1,"EOR",9
1310 DATA"???",1,"???",1,"???",1
1320 DATA"EOR",8,"LSR",8,"???",1
1330 DATA"RTS",1,"ADC",11,"???",1
1340 DATA"???",1,"???",1,"ADC",3
1350 DATA"ROR",3,"???",1,"PLA",1
1360 DATA"ADC",2,"ROR",4,"???",1
1370 DATA"JMP",13,"ADC",5,"ROR",5
1380 DATA"???",1,"BVS",12,"ADC",10
1390 DATA"???",1,"???",1,"???",1
```

```
1400 DATA"ADC",6,"ROR",6,"???",1
1410 DATA"SEI",1,"ADC",9,"???",1
1420 DATA"???",1,"???",1,"ADC",8
1430 DATA"ROR",8,"???",1,"???",1
1440 DATA"STA",11,"???",1,"???",1
1450 DATA"STY",3,"STA",3,"STX",3
1460 DATA"???",1,"DEY",1,"???",1
1470 DATA"TXA",1,"???",1,"STY",5
1480 DATA"STA",5,"STX",5,"???",1
1490 DATA"BCC",12,"STA",10,"???",1
1500 DATA"???",1,"STY",6,"STA",6
1510 DATA"STX",7,"???",1,"TYA",1
1520 DATA"STA",9,"TXS",1,"???",1
1530 DATA"???",1,"STA",8,"???",1
1540 DATA"???",1,"LDY",2,"LDA",11
1550 DATA"LDX",2,"???",1,"LDY",3
1560 DATA"LDA",3,"LDX",3,"???",1
1570 DATA"TAY",1,"LDA",2,"TAX",1
1580 DATA"???",1,"LDY",5,"LDA",5
1590 DATA"LDX",5,"???",1,"BCS",12
1600 DATA"LDA",10,"???",1,"???",1
1610 DATA"LDY",6,"LDA",6,"LDX",7
1620 DATA"???",1,"CLV",1,"LDA",9
1630 DATA"TSX",1,"???",1,"LDY",8
1640 DATA"LDA",8,"LDX",9,"???",1
1650 DATA"CPY",2,"CMP",11,"???",1
1660 DATA"???",1,"CPY",3,"CMP",3
1670 DATA"DEC",3,"???",1,"INY",1
1680 DATA"CMP",2,"DEX",1,"???",1
1690 DATA"CPY",5,"CMP",5,"DEC",5
1700 DATA"???",1,"BNE",12,"CMP",10
1710 DATA"???",1,"???",1,"???",1
1720 DATA"CMP",6,"DEC",6,"???",1
1730 DATA"CLD",1,"CMP",9,"???",1
1740 DATA"???",1,"???",1,"CMP",8
1750 DATA"DEC",8,"???",1,"CPX",2
1760 DATA"SBC",11,"???",1,"???",1
1770 DATA"CPX",3,"SBC",3,"INC",3
1780 DATA"???",1,"INX",1,"SBC",2
1790 DATA"NOP",1,"???",1,"CPX",5
1800 DATA"SBC",5,"INC",5,"???",1
1810 DATA"BEQ",12,"SBC",10,"???",1
1820 DATA"???",1,"???",1,"SBC",6
1830 DATA"INC",6,"???",1,"SED",1
1840 DATA"SBC",9,"???",1,"???",1
1850 DATA"???",1,"SBC",8,"INC",8
1860 DATA"???",1
```

10.1 Beschrijving van het programma

100–150

Vrijmaken van ruimte voor de array's. Inlezen van de data om de tabellen van de instructiewoorden en de adresseerwijzen te vormen.

160–190

Invoer van het start- en eindadres voor de disassemblering. Omzetting in decimale vorm.

200–260

FOR-NEXT lus voor de disassemblering van het programma vanaf het startadres tot het eindadres. Instructies met operanden verhogen de instructieteller automatisch. In regel 220 wordt de instructiecode opgehaald en met het betrokken adres uitgevoerd. In regel 230 worden afhankelijk van de adresseerwijze de operandbytes uitgevoerd. Regel 240 voert het instructiewoord uit en in regel 250 wordt de operand overeenkomstig de adresseerwijze uitgevoerd. Regel 260 tenslotte beëindigt de lus en springt naar de invoer.

280–410

De operand wordt overeenkomstig de adresseerwijze uitgevoerd.

420–530

Met deze subroutines worden bytes en adressen hexadecimaal uitgevoerd.

540–550

Omzetting van een hexadecimaal getal in de decimale vorm.

1000–1860

Tabellen met de instructiewoorden en de adresseerwijzen.

10.2 Variabelen

MN$(255)
Tabel van de instructiewoorden

AD(255)
Tabel met de bijbehorende adresseerwijzen

H$(15)
De hexadecimale getallen 0 t/m F

FF
Constante 255

HI
Constante 256

UL
Constante 65536

SC
Constante 32767

A$
Stringvariabele voor hexadecimaal getal

S
Startadres

E
Eindadres

P
Instructieteller

OP
Adresseerwijze

11 PROFESSIONELE HULPMIDDELEN VOOR HET MAKEN VAN MACHINETAALPROGRAMMA'S

In dit boek hebt u kennis gemaakt met een assembleerprogramma. Dit programma is gebruikt om de sourceprogramma's om te zetten in de betreffende codes. Deze assembler is zelf in BASIC geschreven en neemt 11 KByte geheugenruimte in beslag. Deze 2-pass-assembler staat het gebruik van symbolen voor variabelen en adressen toe. Als slechts relatief kleine programma's worden geassembleerd kan goed met een dergelijk programma worden gewerkt. Als de programma's, door de toenemende ervaring, groter en complexer worden is spoedig de grens van een dergelijke assembler bereikt. Het omzetten van lange sourceprogramma's is bijzonder tijdrovend en ons geduld wordt erg op de proef gesteld. Buitengewoon irritant is het als de assembler een fout ontdekt die verbeterd moet worden. Het sourceprogramma moet dan opnieuw geladen en gecorrigeerd weer in het geheugen worden gezet. Vervolgens moet de assembler opnieuw worden geladen en kan de procedure weer van voren af aan beginnen om dan tot de ontdekking te komen dat een fout in de regel na de verbetering voorkomt. Dat is er de reden van dat we u een professioneel assembleerprogramma presenteren voor de Commodore 64 dat zelf in machinetaal is geschreven en 4K geheugenruimte in beslag neemt. De invoer van sourceprogramma's verloopt analoog aan die van BASIC-programma's maar het is niet nodig om het sourceprogramma apart op diskette op te slaan. De assembler wordt met de SYS-instructie opgeroepen en assembleert in seconden, waarvoor onze BASIC-assembler minuten nodig had. De belangrijkste eigenschappen van de assembler zijn voor u samengevat.

PROFI-ASS 64 is een comfortabele 2-pass macro assembler voor de 65XX-microprocessorfamilie zoals de 6502 en de daarvan afstammende 6510 voor de Commodore 64. Het programma is geheel in machinetaal geschreven en neemt de bovenste 8K RAM (adres $8000-$9FFF) van het BASIC-RAM in beslag. Het meestal voor machinetaalprogramma's te gebruiken gebied $C000-$CFFF blijft vrij. Het programma wordt van diskette geladen met

```
LOAD "PROFI-ASS 64",8,1
```

Met het programma biedt dit de mogelijkheid voor: volledige assemblerlistings, laadbare labels, verschillende mogelijkheden om de geproduceerde Op-codes en operanden weg te zetten, opnieuw te definiëren labels, en het verbinden van kleine programma's tot grote, het gebruik van pseudo-Opcodes (assembler-instructies). De syntax is nauw verwant aan het MOS-standaardformaat.

Assembler sourceprogramma's worden, net als in BASIC, met regelnummers ingevoerd. Evenals in BASIC kunt u ook regelnummers veranderen, tussenvoegen of verwijderen. Daardoor is geen eigen editor vereist en staan u meer geheugenplaatsen voor uw assembler source ter beschikking, samen ruim 30 KByte. Tevens is het mogelijk om meerdere instructies, ook weer net als in BASIC, gescheiden door een dubbele punt, op een regel te zetten.

Assembler sourceregels kunnen bevatten: labels, instructiewoorden (mnemonics), de operanden en commentaar. Bovendien kunnen verschillende assemblerinstructies, pseudo-Opcodes geplaatst worden die PROFI-ASS instrueren om bepaalde dingen te doen.

Iedere programmaregel met een mnemonic of een pseudo-Opcode kan een label bevatten. De labelnaam wordt voor de instructie gezet en gevolgd door een of meer spaties. Een label begint met een letter, gevolgd door een of meer letters en/of cijfers. Verschillende labels onderscheiden zich van elkaar in de eerste acht karakters. Niet-alfanumerieke karakters zijn niet toegestaan. Een spatie scheidt de labelnaam van de erop volgende instructie.

Instructiemnemonics kunnen na een label geplaatst worden of aan het begin van de regel als er geen label geplaatst is. Alle mnemonics bestaan uit drie letters. Mnemonics zijn gereserveerde woorden en mogen niet als label gebruikt worden. Als de instructie met een punt (".") begint, betreft het een pseudo-Opcode (assemblerinstructie). Er zijn drie pseudo-Opcodes die niet met een punt, maar met een speciaal teken beginnen. Alle pseudo-Opcodes moeten van hun operand gescheiden zijn door een spatie met uitzondering van: "=" en "*=". Pseudo-Opcodes die met een punt beginnen worden door de eerste drie karakters van elkaar onderscheiden. In assemblerlistings worden ze wel volledig afgedrukt.

Een assemblerregel kan op iedere plek door een puntkomma ";" afgesloten worden. Alles wat volgt is commentaar. Commentaar wordt met de assemblerlisting afgedrukt maar de computer doet er verder niets mee. (vergelijk REM in BASIC). Een dubbele punt ":" beëindigt het commentaar en is het begin van een nieuwe instructie, indien de dubbele punt niet tussen aanhalingstekens staat. De assembler kent ook regels met alleen commentaar. De regel begint dan natuurlijk met een puntkomma.

De adresseerwijze wordt aangegeven door op een bepaalde manier de operand op te schrijven. De adresseerwijzen met de uitdrukkingen hebben de volgende syntaxis:

#uitdrukking	immediate

uitdrukking		absolute of relative
uitdrukking,x	absolute,x	indexed x
uitdrukking,y	absolute,y	indexed y
(uitdrukking,x)		indexed indirect
(uitdrukking),y		indirect, indexed
(uitdrukking)		indirect

Zoals gewoonlijk mag de regel met een ";" worden afgesloten en kan commentaar volgen.

PROFI-ASS 64 zet de absolute adressering automatisch om in de zeropage-adresseerwijze als de uitdrukking een waarde heeft die kleiner is dan 256. Het is dus niet nodig, zoals bij onze BASIC-assembler, de uitdrukking door een ster vooraf te laten gaan. Als u absolute adressering wilt, kan voor de uitdrukking een uitroepteken geplaatst worden. LDA !5,X geeft de code BD 05 00, de absolute vorm van LDA terwijl LDA 5,X de zeropage-adressering B5 05 oplevert.

11.1 Uitdrukkingen

PROFI-ASS 64 dankt zijn veelzijdigheid aan de mogelijkheid om willekeurig ingewikkelde uitdrukkingen te kunnen berekenen. De assembler heeft een recursieve routine voor de berekening van willekeurig in elkaar verstrengelde uitdrukkingen, wat u veel meer mogelijkheden biedt dan de MOS-standaard of andere assemblers. De BASIC-assembler liet immers alleen constanten en labels toe. Een "PROFF-ASS 64 uitdrukking" kan overal staan waar in bovenstaande lijst het woord uitdrukking staat. Ook bij de pseudo-Opcodes zijn dergelijke uitdrukkingen toegestaan, indien een numerieke uitdrukking verwacht wordt. Dit gebruik van uitdrukkingen maakt PROFI-ASS 64 bijzonder krachtig, uw assemblerprogramma's kunnen volledig voorzien worden van symbolen om zo het programma gemakkelijk te kunnen veranderen en te verplaatsen. De programma's worden zo erg overzichtelijk. De syntaxis van de uitdrukkingen is erg eenvoudig en komen overeen met de MOS-standaard. Uitdrukkingen worden op dezelfde manier ingevoerd als bij een rekenmachientje. Ook daar wordt met haakjes gewerkt. De operatoren worden van links naar rechts afgewerkt, maar ronde en rechte haakjes geven de prioriteit aan.

De volgende operatoren zijn beschikbaar:

+ telt waarden op
- trekt waarden af
* vermenigvuldigt waarden
! logische OR tussen twee waarden
& logische AND tussen twee waarden
^ logische XOR (exclusive OR) tussen twee waarden
> verschuift het linker argument het aantal bits naar rechts, als het rechter argument aangeeft
< verschuift het linker argument het aantal bits naar links, als het rechter argument aangeeft.

Alle bewerkingen worden tussen 16-bits waarden uitgevoerd. Als de bewerking een overschot oplevert, verschijnt 'ILLEGAL QUANTITY ERROR'. Deze fout treedt op als met waarden boven de 32767 wordt vermenigvuldigd of over meer dan 15 bits naar links wordt verschoven. Bij de optelling en de aftrekking wordt een waarde boven 65535 geïnterpreteerd als een negatief getal in het "two's complement".

De operanden zelf kunnen op verschillende manieren voorkomen.

11.2 Operandtypes

type	voorbeeld
hexadecimaal	$1C3
decimaal	127
binair	%110011
PC	*
ASCII-teken	"A"
label	SYMB
uitdrukking	("Z"+6)

Ieder type kan met de boven beschreven bewerkingen worden verbonden. De ronde en de vierkante haken (accolades) kunnen gebruikt worden om de volgorde van de bewerkingen aan te geven. Voor iedere operand (ook een uitdrukking tussen haakjes) kan een minteken geplaatst worden. Dit betekent dat het two's complement moet worden bepaald. Een volledige uitdrukking kan nog door een voorafgaand teken worden bewerkt. Het uitroepteken gaf de absolute adresseerwijze aan. Het groter dan teken ">" voor een uitdrukking betekent dat de hi-byte wordt genomen, het kleiner dan

teken "<" zorgt voor de lo-byte van de uitdrukking. Bij de direkte adressering of bij de .BYTE pseudo-Opcode is dit nodig. De hi-byte operator (>) doet hetzelfde als:

uitdrukking : 8

De lo-byte operator is identiek aan:

uitdrukking & $FF

Deze operatoren kent de assembler ook bij de immediate adressering.

11.3 Pseudo-Opcodes

. BYTE

De .BYTE pseudo-Opcode wordt gebruikt om een bytewaarde aan de positie van de instructieteller toe te voegen. De instructie komt overeen met de .BY van de BASIC-assembler. Als operanden kunnen geldige PROFI-ASS uitdrukkingen gebruikt worden, die door een komma van elkaar gescheiden zijn. Het aantal wordt begrensd door de regellengte en de capacteit van de PROFI-ASS-64 buffer. Er kunnen willekeurige uitdrukkingen met willekeurige operatoren gebruikt worden. De verkregen waarde van de uitdrukking mag de inhoud van een byte niet te boven gaan, anders wordt een "ILLIGAL QUANTITY ERROR" gegeven. 2-bytes waarden kunnen verdeeld worden met de operatoren ">" en "<" in een hi-byte en een lo-byte waarde. Een 1-byte waarde heeft de waarde 0 t/m 255 of $FF80 t/m $FFFF. De hoge waarden zijn toegestaan, omdat ze normaal negatieve getallen van -1 t/m -128 betekenen. Daarom is de regel ".BYTE -1" toegestaan. .BYTE dient voor het definiëren van tabellen zoals sprongadressen of wijzers. U kunt er ook instructies mee "binnensmokkelen", zoals de BIT-instructie. Een bekende programmeertruc:

```
       .BYTE $2C  ;  absolute BIT-instructie
 LABEL LDA #-1    ;  verborgen LDA-instructie
```

.WORD

De .WORD pseudo-Opcode wordt gebruikt om 2-byte adressen in het

standaardformaat low-high in de objectcode in te voegen. De instructie komt overeen met ".BYTE <ADRES, >ADRES".

. FILE

Als meer assembler sourceprogramma's met elkaar verbonden moeten worden, wordt de .FILE-instructie gebruikt. De syntaxis is als volgt:

```
.FILE nummer randapparaat, "naam file"
```

het nummer van de diskdrive is 8 en de naam van de file is de naam van het assemblerprogramma dat nageladen moet worden. Als u een zeer lang programma schrijft, is het dus mogelijk dit in stukken te verdelen en pas achteraf bij het assembleren met elkaar te verbinden.

.IF

De .IF-instructie wordt voor voorwaardelijke assemblering gebruikt. Hij heeft een uitdrukking als argument en wordt in pass-1 en in pass-2 berekend. Als de uitdrukking ongelijk is aan nul wordt de code in dezelfde regel na .IF geassembleerd. Meestal zal daar een .GOTO voor de sprong staan. De toegevoegde code in de regel moet door een dubbele punt ":" gescheiden zijn. Met .IF, .GOTO en het opnieuw definiëren van labels is het mogelijk assemblerlussen te maken. Hoewel .IF alleen maar test op nul is het met een simpele truc mogelijk andere vergelijkingen te maken. Het schuiven van 15 bits naar rechts geeft een resultaat 1 als de uitdrukking negatief was en 0 als dit positief was. Twee getallen kunnen dus met elkaar worden vergeleken als ze van elkaar worden afgetrokken en het resultaat op een positieve dan wel negatieve waarde test.

.GOTO

De .GOTO-instructie maakt een voorwaardelijke sprong mogelijk naar het regelnummer dat als argument staat aangegeven. Dit regelnummer kan ook een uitdrukking zijn. Samen met .IF en het verminderen van een bepaalde variabele kan een lus worden samengesteld.

.GTB

Met .GTB "GO TO BASIC" wordt de controle overgedragen aan BASIC. Er is geen argument. De in het programma volgende BASIC-instructies worden uitgevoerd. Naar assembler kan teruggekeerd worden via een speciaal inspringpunt. Het adres is 5

bytes voor het laatste adres van de assembler, dit is $9FFA (40954)

.ASC "TEKST"

Hiermee kan tekst in het assemblerprograma worden gezet. De tekst kan 55 karakters bevatten en moet tussen aanhalingstekens worden gezet. Ook cursorbesturingstekens kunnen toegepast worden.

.SYS

Machinetaalprogramma's worden hiermee, zowel in pass-1 als in pass-2, aangeroepen. Als inspringadres kan een willekeurige uitdrukking staan. De instructie komt overeen met die in BASIC. De instructie kan gebruikt worden om bijvoorbeeld zelf geconstrueerde pseudo-Opcodes in machinetaal tijdens het assembleren op te roepen.

.SST nummer randapparaat, secundair adres, "naam file"

Hiermee kan de tabel van gebruikte symbolen weggezet worden op het aangegeven randapparaat (de diskdrive heeft nummer 8). Het secundaire adres ligt in tussen 2 en 14. De naam van de file wordt net als bij de OPEN-instructie aangegeven en vereist een ",S,W" aan het einde om aan te geven dat het een sequentieel bestand betreft en dat de file "geschreven" moet worden. Deze instructie wordt ook gebruikt als een gesorteerde lijst met symbolen en labels moet worden uitgevoerd. Het bestand kan ook als invoer van de symbolen voor een programma gebruikt worden.

.LST nummer randapparaat, secundairadres, "naam file"

Hiermee kan een symbooltabel geladen worden die door een ander programma is gemaakt, bijvoorbeeld een lijst van symbolen die gebruikt worden voor routines van het operating system.

.END

Het sourceprogramma wordt hiermee afgesloten. Noodzakelijk is dit echter niet. Met .END wordt aan het einde van pass-2 naar BASIC teruggekeerd. BASIC-instructies worden daarna uitgevoerd. Ook kunt u met SYS uw pas geassembleerde machinetaalprogramma's oproepen.

.OPT

De OPT-pseudo-Opcode betekent optie. Met de opties kunt u beslissingen nemen omtrent de asemblerlisting en de objectcodes. De syntaxis is de volgende:

.OPT optie, optie, optie.....

De volgende opties zijn beschikbaar:

P - Print.

Hiermee bepaalt u of een listing gemaakt moet worden en waarheen deze gestuurd moet worden. Als alleen een P wordt ingevoerd (dus .OPT P) wordt de listing op het scherm gezet. Wilt u de listing op de printer hebben, dan moet vooraf met bijvoorbeeld OPEN 1,4 de logische file met nummer 1 naar de printer (nummer 4) gestuurd worden. Met .OPT P1 gaat de listing naar de printer.

O - Objectcode-uitvoer.

Hiermee bepaalt u wat er met de geproduceerde codes gebeuren moet. Met .OPT O gaan de objectcodes in een speciale buffer, direkt boven het assemblerprogramma, waar normaal de arrayvariabelen neergezet worden; door middel van dezelfde wijzer worden ze gebruikt. Met .OPT O0 worden de objectcodes direkt naar de plaatsen in het geheugen gestuurd waarvoor zij bestemd zijn. Met .OPT O1 worden ze op de logische file met nummer 1 gezet. Als bijvoorbeeld vooraf een file geopend was met OPEN 1,8,1,"0:PROGRAMMA" dan wordt uw geproduceerde machinetaalprogramma direkt op schijf gezet en kan dan later met LOAD geladen worden.

Met het volgende voorbeeldprogramma wordt de inhoud van de zeropage vanaf regel "LINE" op het scherm gezet. Met dit voorbeeld kunt u zien hoe met de assembler moet worden omgegaan.

```
10 SYS 9*4096 : REM PROFI-ASS 64 OPROEPEN OP 32768 ($8000)
20 .OPT P,O0 ; LISTING OP SCHERM EN CODES IN GEHEUGEN
30 *= $C000 ; STARTADRES MACHINETAALPROGRAMMA
40 LINE = 10 ; BEELDSCHERMREGEL
50 VIDEO = $400 ; BEELDSCHERM ADRES
60 COLOR = $D800 ; ADRES VAN HET KLEURENRAM
70 KLEUR = 1 ; WIT SCHRIFT
80 LDX #0
90 LUS LDA 0,X ; BYTE UIT DE ZEROPAGE HALEN
100 STA VIDEO+(40*LINE),X ; KARAKTER UITVOEREN
110 LDA #KLEUR
120 STA COLOR+(40*LINE),X ; KLEUR ZETTEN
130 INX
140 BNE LUS
150 RTS
160 .END
```

In het volgende voorbeeld worden de objectcodes direkt op diskette gezet en de listing wordt naar de printer gestuurd. Het sourceprogramma bestaat uit meer afzonderlijke programma's.

```
10 OPEN 1,8,1, "):OBJECTCODE"
20 OPEN 2,4 : REM PRINTER
30 SYS 9*4096
40 .OPT O1,P2
50 ; AASEMBLERINSTRUCTIES
...
1000 .FILE 8, "PROGRAMMA 2"
```

Programma 2 bevat

```
10 ; VERDERE INSTRUCTIES
...
1000 .FILE 8, "PROGRAMMA 3"
```

Programma 3 bevat

```
10 ; VERDERE INSTRUCTIES
...
1000 .END 8, "PROGRAMMA 1"
```

Programma 1 is het programma dat SYS 9*4096 bevat.

PROFI-ASS 64 heeft een serie duidelijke foutmeldingen die voor de foutieve regel uitgevoerd worden.

We hebben nu een professionele assembler voor de Commodore 64 leren kennen. PROFI-ASS 64 is een onderdeel van het softwarepakket PROFIMAT en bevat nog een comfortabel monitorprogramma PROFI-MON 64.

De functie van een monitorprogramma is bij de invoer van machinetaalprogramma's al kort toegelicht. In de eerste plaats moeten register- en geheugeninhouden aangegeven en veranderd kunnen worden. Op het beeldscherm moet dit zichtbaar gemaakt worden. Ten tweede moet met het monitorprogramma het wegzetten en laden van machinetaalprogramma's mogelijk zijn. Bovendien moeten de machinetaalprogramma's uitgevoerd kunnen worden. Als de programma's met een BRK-instructie worden afgesloten, wordt vervolgens naar het monitorprogramma gesprongen en de inhoud van de registers wordt getoond. Naast deze basisfuncties van de monitor biedt PROFI-MON 64 nog een serie comfortabele instructies die het werken in machinetaal en het uittesten van programma's vergemakkelijken.

Het programma wordt met LOAD "PROFI-MON64",8,1 geladen van de diskette. De monitor neemt 3K geheugen in beslag en staat op de adressen $C000 t/m $CBFF, dus buiten het BASIC-gebied.

Met SYS 49152 wordt het programma gestart. De monitor meldt zich met:

```
C*
     PC IRQ SR AC XR YR SP  NV-BDIZC
>; 0003 EA31 32 34 02 A2 F8 00110010
```

De letter "C" betekent "Call" en geeft de normale oproep van de monitor aan. De registerinhouden worden vervolgens getoond op de manier die wij al kennen van de stap-voor-stap simulator. Tevens laat dit programma nog de interruptvector IRQ zien, naar dit adres wordt gesprongen als een lopend programma door een impuls op de interruptleiding wordt onderbroken. Als geen interruptroutines gebruikt worden hoeven we ons om deze waarde niet te bekommeren.

De registerinhouden kunnen worden gewijzigd door met de cursor naar het betreffende register te gaan en de inhoud te veranderen. Door op de RETURN-toets te drukken wordt de verandering uitgevoerd. De flags worden veranderd via het statusregister en als deze inhoud wordt veranderd, veranderen de flags automatisch mee.

>R register display

Op ieder moment kunt u de registers op het scherm krijgen met bovenstaande instructie en indien gewenst wijzigen.

>M memory display

Om geheugeninhouden te zien en te wijzigen wordt de >M-instructie gebruikt. Het start- en eindadres wordt hexadecimaal aangegeven.

```
>M A0A0 A0AF
>: A0A0 C4 46 4F D2 4E 45 58 D4 DFORNEXT
>: A0A8 44 41 54 C1 49 4E 50 55 DATAINPU
```

Het teken '>' wordt door PROFI-MON 64 zelf neergezet en geeft aan dat de monitor klaar staat voor nieuwe instructies. De uitvoer van deze instructies kan op de volgende manier geïnterpreteerd worden.

Na de dubbele punt wordt een adres uitgevoerd. Dit adres is het eerste van de acht bytes die erop volgen. In het voorbeeld heeft $A0A0 de inhoud $C4. Het adres $A0A1 heeft als waarde $46 enz. In totaal worden acht bytes getoond. Achter de acht bytes staan de

ASCII-waarden. Als de waarde niet als een afdrukbaar karakter kan worden weergegeven volgt een punt. Dit is nuttig als u tekst in het geheugen moet opzoeken. Voor het veranderen van een byte gaat u met de cursor naar de te veranderen plek en u verandert de waarde. Na het indrukken van de RETURN-toets wordt de verandering uitgevoerd. De ASCII-presentatie wordt automatisch mee veranderd.

>G uitvoering van machinetaalprogramma's

U kunt een machinetaalprogramma starten met >G CF20. Het startadres is dan $CF20. Eerst krijgen de registers de waarden die via >R getoond worden. Natuurlijk kunt u desgewenst de inhoud van een of meer registers wijzigen voordat het programma gestart wordt. Als laatste instructie kunt u BRK gebruiken. Na deze instructie wordt weer naar de monitor gesprongen en worden de registerinhouden getoond, bijvoorbeeld:

```
B*
     PC IRQ SR AC XR YR SP  NV-BDIZC
>: CF39 EA31 B3 8F 73 B0 F6 10110011
```

De B betekent dat het programma bij een BRK-instructie is gekomen en naar de monitor is gesprongen. De instructieteller wijst naar de byte waarachter de BRK-instructie staat. Als u meerdere BRK-instructies opneemt in een programma, weet u precies waar uw programma's worden onderbroken. Om programma's te testen kunnen op strategisch belangrijke plaatsen BRK-instructies worden opgenomen zodat het programma hier wordt onderbroken. Ook kunnen nu de registerinhouden evenals enkele gegevens in het geheugen worden gecontroleerd. Als het programma tot die plaats goed functioneert, kan de BRK-instructie worden vervangen door de oorspronkelijke instructie en kan op de volgende kritieke plek een nieuwe BRK-instructie worden geplaatst. Hiermee kunt u het programma stap voor stap testen totdat het naar tevredenheid werkt.

>L en >S laden en wegzetten van machinetaalprogramma's

Om een machinetaalprogramma te laden wordt de instructie >L gevolgd door de naam van het programma tussen aanhalingstekens en daarna, door een komma gescheiden, het nummer van het randapparaat (hexadecimaal, twee plaatsen). Als het programma "GRAFIEK" geladen moet worden van diskette kunt u aangegeven:

```
>L "GRAFIEK",08
```

Het laden van cassetteband gaat met: >L "GRAFIEK,01. PROFI-MON 64 biedt nog de mogelijkheid om programma's weg te zetten op bepaalde

adressen. Bij de vorige instructie werd het programma geladen in de adressen waar ze bij het wegzetten vandaan kwamen. Wordt nu een startadres mee ingevoerd dan wordt het programma vanaf dit adres in het geheugen gezet. Als u bijvoorbeeld een programma wilt laden vanaf adres $1000, is de instructie:

```
>L "GRAFIEK",08,1000
```

De S-instructie verloopt analoog. Om de computer te laten weten welke geheugeninhouden moeten worden bewaard moet een start- en eindadres worden opgegeven. Bijvoorbeeld:

```
>S "PROGRAMMA",08,7000,8000
```

Hiermee worden de inhouden van de adressen $7000 t/m $7FFF op diskette weggezet (01 cassetteband).

>D disassembler

De ingebouwde disassembler leest de geheugeninhouden en zet ze om in de bijbehorende Op-codes en operanden. De uitvoer vindt op dezelfde wijze plaats als bij de BASIC-disassembler. Bij niet bestaande Op-codes worden drie vraagtekens uitgevoerd. Als een programma op de adressen $ B824 t/m $B82C gedisassembleerd wordt, ontstaat bijvoorbeeld het volgende:

```
>D B824 B82C
>, B824   20 EB B7   JSR $B7EB
>, B827   8A         TXA
>, B828   A0 00      LDY #$00
>, B82A   91 14      STA ($14),Y
>, B82C   60         RTS
```

Dit is de POKE-instructie van de BASIC-interpreter.

>C compare - vergelijking van geheugengebieden

Met >C 8000 8100 9000 worden de adresinhouden van $8000 t/m$8100 vergeleken met die van $9000 t/m $9100. De adresinhouden die niet overeenstemmen worden uitgevoerd. Als bijvoorbeeld 8056 verschijnt betekent dat, dat de inhoud van $8056 niet overeenstemt met $9056.

>T transfer - verschuiven van geheugengebieden

Met >T 6000 6FFF 3000 wordt het programma met startadres $6000 en eindadres $6FFF gekopieerd in de adressen $3000 t/m$3FFF. De inhouden van de oorspronkelijke adressen blijven evenals bij de transfer-instructies van de processor behouden.

>H hunt - zoeken naar bytecombinaties

Om een bepaald geheugengebied te onderzoeken op een zekere bytecombinatie wordt de H-instructie gebruikt. Met de instructie:

```
>H E000 EFFF 20 D2 FF
```

wordt het geheugengebied vanaf $E000 t/m $EFFF onderzocht op de bytevolgorde $20 $D2 en $FF (oproep voor de uitvoerroutine JSR $FFD2). Als resultaat worden alle adressen getoond waarin deze combinatie voorkomt. Ook kan een geheugengebied onderzocht worden op tekst (ASCII-waarden).

```
>H A000 AFFF "READY"
```

Het geheugengebied $A000 t/m $AFFF wordt doorzocht op het woord "READY". Wanneer als resultaat A378 wordt uitgevoerd, betekent dat, dat vanaf adres $A378 het woord "READY" in het geheugen staat.

>F fill - vullen met een constante waarde

Met >F 8000 8FFF 00 wordt het gehele geheugengebied van $8000 t/m $8FFF gevuld met de constante waarde $00.

>BR >BA >BC schakelaar voor geheugenconfiguratie

Met >BA wordt het gehele geheugen op RAM gezet, met BC is alleen het karakter-ROM ingeschakeld. Naar de normale ROM-configuratie wordt geschakeld met >BR

adresgebied	>BR	>BA	>BC
$E000-$FFFF	ROM	RAM	RAM
$D000-$DFFF	I/O	RAM	CHARACTERROM
$C000-$CFFF	RAM	RAM	RAM
$A000-$BFFF	ROM	RAM	RAM
$0000-$9FFF	RAM	RAM	RAM

>W walk - stap-voor-stap

Met >W BC16 wordt vanaf adres $BC16 het programma stap-voor-stap uitgevoerd. De registerinhouden verschijnen op het scherm.

>X exit-terugkeer naar BASIC

Met deze instructie wordt weer naar de BASIC-interpreter teruggekeerd. Het BASIC bevindt zich in dezelfde toestand als voor het aanroepen van de monitor. U kunt met SYS 49152 (12*4096)

weer naar de monitor teruggaan.

Adresseerwijze en instructiecodes

*	A	#	ZP	AB	ABX	ABY	ZPX	ZPY	,X)	),Y
ADC	-	69	65	6D	7D	79	75	-	61	71
AND	-	29	25	2D	3D	39	35	-	21	31
ASL	0A	-	06	0E	1E	-	16	-	-	-
BIT	-	-	24	2C	-	-	-	-	-	-
CMP	-	C9	C5	CD	DD	D9	D5	-	C1	D1
CPX	-	E0	E4	EC	-	-	-	-	-	-
CPY	-	C0	C4	CC	-	-	-	-	-	-
DEC	-	-	C6	CE	DD	-	D6	-	-	-
EOR	-	49	45	4D	5D	59	55	-	41	51
INC	-	-	E6	EE	FD	-	F6	-	-	-
LDA	-	A9	A5	AD	BD	B9	B5	-	A1	B1
LDX	-	A2	A6	AE	-	BE	-	B6	-	-
LDY	-	A0	A4	AC	BC	-	B4	-	-	-
LSR	4A	-	46	4E	5E	-	56	-	-	-
ORA	-	09	05	0D	1D	19	15	-	01	11
ROL	2A	-	26	2E	3E	-	36	-	-	-
ROR	6A	-	66	6E	7E	-	76	-	-	-
SBC	-	E9	E5	ED	FD	F9	F5	-	E1	F1
STA	-	-	85	8D	9D	99	95	-	81	91
STX	-	-	86	8E	-	-	-	96	-	-
STY	-	-	84	8C	-	-	94	-	-	-

spronginstructies	BPL	BMI	BVC	BVS	BCC	BCS	BNE	BEQ
	10	30	50	70	90	B0	D0	F0
transferinstructies	TXA	TAX	TYA	TAY	TSX	TXS		
	8A	AA	98	A8	BA	9A		
stackinstructies	PHP	PLP	PHA	PLA				
	08	28	48	68				
onderbrekingsinstructies	BRK	JSR	RTI	RTS	JMP	JMP	NOP	
	00	20	40	60	4C	6C	EA	
flaginstructies	CLC	SEC	CLI	SEI	CLV	CLD	SED	
	18	38	58	78	B8	D8	F8	
getalinstructies	DEY	INY	DEX	INX				
	88	C8	CA	E8				

omrekeningstabel decimaal - hexadecimaal - binair

decimaal	hexadecimaal	binair	decimaal	hexadecimaal	binair
0	00	00000000	26	1A	00011010
1	01	00000001	27	1B	00011011
2	02	00000010	28	1C	00011100
3	03	00000011	29	1D	00011101
4	04	00000100	30	1E	00011110
5	05	00000101	31	1F	00011111
6	06	00000110	32	20	00100000
7	07	00000111	33	21	00100001
8	08	00001000	34	22	00100010
9	09	00001001	35	23	00100011
10	0A	00001010	36	24	00100100
11	0B	00001011	37	25	00100101
12	0C	00001100	38	26	00100110
13	0D	00001101	39	27	00100111
14	0E	00001110	40	28	00101000
15	0F	00001111	41	29	00101001
16	10	00010000	42	2A	00101010
17	11	00010001	43	2B	00101011
18	12	00010010	44	2C	00101100
19	13	00010011	45	2D	00101101
20	14	00010100	46	2E	00101110
21	15	00010101	47	2F	00101111
22	16	00010110	48	30	00110000
23	17	00010111	49	31	00110001
24	18	00011000	50	32	00110010
25	19	00011001	51	33	00110011

decimaal	hexadecimaal	binair	decimaal	hexadecimaal	binair
52	34	00110100	78	4E	01001110
53	35	00110101	79	4F	01001111
54	36	00110110	80	50	01010000
55	37	00110111	81	51	01010001
56	38	00111000	82	52	01010010
57	39	00111001	83	53	01010011
58	3A	00111010	84	54	01010100
59	3B	00111011	85	55	01010101
60	3C	00111100	86	56	01010110
61	3D	00111101	87	57	01010111
62	3E	00111110	88	58	01011000
63	3F	00111111	89	59	01011001
64	40	01000000	90	5A	01011010
65	41	01000001	91	5B	01011011
66	42	01000010	92	5C	01011100
67	43	01000011	93	5D	01011101
68	44	01000100	94	5E	01011110
69	45	01000101	95	5F	01011111
70	46	01000110	96	60	01100000
71	47	01000111	97	61	01100001
72	48	01001000	98	62	01100010
73	49	01001001	99	63	01100011
74	4A	01001010	100	64	01100100
75	4B	01001011	101	65	01100101
76	4C	01001100	102	66	01100110
77	4D	01001101	103	67	01100111

decimaal	hexadecimaal	binair	decimaal	hexadecimaal	binair
104	68	01101000	130	82	10000010
105	69	01101001	131	83	10000011
106	6A	01101010	132	84	10000100
107	6B	01101011	133	85	10000101
108	6C	01101100	134	86	10000110
109	6D	01101101	135	87	10000111
110	6E	01101110	136	88	10001000
111	6F	01101111	137	89	10001001
112	70	01110000	138	8A	10001010
113	71	01110001	139	8B	10001011
114	72	01110010	140	8C	10001100
115	73	01110011	141	8D	10001101
116	74	01110100	142	8E	10001110
117	75	01110101	143	8F	10001111
118	76	01110110	144	90	10010000
119	77	01110111	145	91	10010001
120	78	01111000	146	92	10010010
121	79	01111001	147	93	10010011
122	7A	01111010	148	94	10010100
123	7B	01111011	149	95	10010101
124	7C	01111100	150	96	10010110
125	7D	01111101	151	97	10010111
126	7E	01111110	152	98	10011000
127	7F	01111111	153	99	10011001
128	80	10000000	154	9A	10011010
129	81	10000001	155	9B	10011011

decimaal	hexadecimaal	binair	decimaal	hexadecimaal	binair
156	9C	10011100	182	B6	10110110
157	9D	10011101	183	B7	10110111
158	9E	10011110	184	B8	10111000
159	9F	10011111	185	B9	10111001
160	A0	10100000	186	BA	10111010
161	A1	10100001	187	BB	10111011
162	A2	10100010	188	BC	10111100
163	A3	10100011	189	BD	10111101
164	A4	10100100	190	BE	10111110
165	A5	10100101	191	BF	10111111
166	A6	10100110	192	C0	11000000
167	A7	10100111	193	C1	11000001
168	A8	10101000	194	C2	11000010
169	A9	10101001	195	C3	11000011
170	AA	10101010	196	C4	11000100
171	AB	10101011	197	C5	11000101
172	AC	10101100	198	C6	11000110
173	AD	10101101	199	C7	11000111
174	AE	10101110	200	C8	11001000
175	AF	10101111	201	C9	11001001
176	B0	10110000	202	CA	11001010
177	B1	10110001	203	CB	11001011
178	B2	10110010	204	CC	11001100
179	B3	10110011	205	CD	11001101
180	B4	10110100	206	CE	11001110
181	B5	10110101	207	CF	11001111

decimaal	hexadecimaal	binair
208	D0	11010000
209	D1	11010001
210	D2	11010010
211	D3	11010011
212	D4	11010100
213	D5	11010101
214	D6	11010110
215	D7	11010111
216	D8	11011000
217	D9	11011001
218	DA	11011010
219	DB	11011011
220	DC	11011100
221	DD	11011101
222	DE	11011110
223	DF	11011111
224	E0	11100000
225	E1	11100001
226	E2	11100010
227	E3	11100011
228	E4	11100100
229	E5	11100101
230	E6	11100110
231	E7	11100111
232	E8	11101000
233	E9	11101001

decimaal	hexadecimaal	binair
234	EA	11101010
235	EB	11101011
236	EC	11101100
237	ED	11101101
238	EE	11101110
239	EF	11101111
240	F0	11110000
241	F1	11110001
242	F2	11110010
243	F3	11110011
244	F4	11110100
245	F5	11110101
246	F6	11110110
247	F7	11110111
248	F8	11111000
249	F9	11111001
250	FA	11111010
251	FB	11111011
252	FC	11111100
253	FD	11111101
254	FE	11111110
255	FF	11111111

Tabel van de 6510 instructiecodes

	0	1	2	4	5	6	8	9	A	C	D	E
0	BRK	ORA ,X)			ORA ZP	ASL ZP	PHP	ORA #	ASL A		ORA AB	ASL AB
1	BPL	ORA),Y			ORA ZPX	ASL ZPX	CLC	ORA ABY			ORA ABX	ASL ABX
2	JSR	AND ,X)		BIT ZP	AND ZP	ROL ZP	PLP	AND #	ROL A	BIT AB	AND AB	ROL AB
3	BMI	AND),Y			AND ZPX	ROL ZPX	SEC	AND ABY			AND ABX	ROL ABX
4	RTI	EOR ,X)			EOR ZP	LSR ZP	PHA	EOR #	LSR A	JMP AB	EOR AB	LSR AB
5	BVC	EOR),Y			EOR ZPX	LSR ZPX	CLI	EOR ABY			EOR ABX	LSR ABX
6	RTS	ADC ,X)			ADC ZP	ROR ZP	PLA	ADC #	ROR A	JMP IND	ADC AB	ROR AB
7	BVS	ADC),Y			ADC ZPX	ROR ZPX	SEI	ADC ABY			ADC ABX	ROR ABX
8		STA ,X)		STY ZP	STA ZP	STX ZP	DEY		TXA	STY AB	STA AB	STX AB
9	BCC	STA),Y		STY ZPX	STA ZPX	STX ZPY	TYA	STA ABY	TXS		STA ABX	
A	LDY #	LDA ,X)	LDX #	LDY ZP	LDA ZP	LDX ZP	TAY	LDA #	TAX	LDY AB	LDA AB	LDX AB
B	BCS	LDA),Y		LDY ZPX	LDA ZPX	LDX ZPY	CLV	LDA ABY	TSX	LDY ABX	LDA ABX	LDX ABY
C	CPY #	CMP ,X)		CPY ZP	CMP ZP	DEC ZP	INY	CMP #	DEX	CPY AB	CMP AB	DEC AB
D	BNE	CMP),Y			CMP ZPX	DEC ZPX	CLD	CMP ABY			CMP ABX	DEC ABX
E	CPX #	SBC ,X)		CPX ZP	SBC ZP	INC ZP	INX	SBC #	NOP	CPX AB	SBC AB	DEC AB
F	BEQ	SBC),Y			SBC ZPX	INC ZPX	SED	SBC ABY			SBC ABX	INC ABX

De instructiecodes kunnen op de volgende manier worden verkregen. Uit de regel wordt de hi-nibble verkregen. Uit de kolom volgt de lo-nibble. De code EOR# heeft als code $49. De afkortingen voor de adressseerwijzen zijn:

A	accumulator
AB	absolute
ABX	absolute, X indexed
ABY	absolute, Y indexed
IND	indirect
ZP	zeropage
ZPX	zeropage, X indexed
ZPY	zeropage, Y indexed
#	immediate
,X)	X-indexed, indirect
),Y	indirect, Y indexed

Instructies en byte-inhoud en de beïnvloeding van de flags van alle 6510-instructies en de verwijzing naar de bladzijde waar zij op voorkomen.

*	Bitpatroon	N	V	B	D	I	Z	C	bladzijde
ADC	011XXX01	X	X				X	X	28
AND	001XXX01	X					X		33
ASL	000XXX10	X					X	X	50
BCC	10010000								41
BCS	10110000								41
BEQ	11110000								41
BIT	0010X100	M	M				X		37
BMI	00110000								41
BNE	10010000								41
BPL	00010000								41
BRK	00000000			1		1			57
BVC	01010000								41
BVS	01110000								41
CLC	00011000							0	49
CLD	11011000				0				49
CLI	01011000					0			49
CLV	10111000		0						49
CMP	110XXX01	X					X	X	38
CPX	111XXX00	X					X	X	40
CPY	1100XX00	X					X	X	41
DEC	110XX110	X					X		48
DEX	11001010	X					X		47
DEY	10001000	X					X		48
EOR	010XXX01	X					X		36
INC	000XX110	X					X		47
INX	11101000	X					X		46
INY	11001000	X					X		47
JMP	01X01100								45
JSR	00100000								54
LDA	101XXX01	X					X		18
LDX	101XXX10	X					X		19
LDY	101XXX00	X					X		19
LSR	010XXX10	0					X	X	51
NOP	11101010								58

*	Bitpatroon	N	V	B	D	I	Z	C	bladzijde
ORA	000XXX01	X					X		35
PHA	01001000								56
PHP	00001000								56
PLA	01101000	X					X		56
PLP	00101000	X	X	X	X	X	X	X	56
ROL	001XXX10	X					X	X	52
ROR	011XXX10	X					X	X	52
RTI	01000000	X	X	X	X	X	X	X	57
RTS	01100000								55
SBC	111XXX01	X	X				X	X	31
SEC	00111000							1	49
SED	11111000				1				49
SEI	01111000					1			49
STA	100XXX01								25
STX	100XX110								25
STY	100XX100								25
TAX	10101010	X					X		26
TAY	10101000	X					X		27
TSX	10111010	X					X		27
TXA	10001010	X					X		27
TXS	10011010								27
TYA	10011000	X					X		27

Staat bij de byte-inhoud een "X" dan betekent dat, dat de waarde van X afhankelijk is van de adresseerwijze. Bij de flags betekent een X dat deze flag door de betreffende instructie wordt beïnvloed. 0 of 1 betekent dat de flag gewist of gezet wordt. Als onder de flag niets staat, wordt de flag door de instructie niet beïnvloed.

Een boek voor iedereen die naast het werken met de ingebouwde BASIC v2.0 van de Commodore 64 wil gaan programmeren in machinetaal.
Alles over de opbouw en werking van de 6510 microprocessor, machinetaalmonitor en assembler.

Een greep uit de inhoud:
- commando's en adressering van de 6510
- het ingeven en uitvoeren van machinetaalprogramma's
- de assembler
- one-step simulator (het stap voor stap testen van een ML-programma)
- machinetaal op de Commodore 64
- het gebruik van KERNAL-routines (ingebouwde routines)
- machinetaal laden via BASIC
- professionele hulpmiddelen om in machinetaal te programmeren
- omrekentabel en commando-overzicht

ISBN 90 229 3333 4

Commodore Bibliotheek 9

DATA BECKER
NEDERLANDS