Machine Code met de ZX SPECTRUM

I. Stewart & R. Jones

MACHINE CODE MET DE ZX SPECTRUM

MAARTEN KLUWER'S
INTERNATIONALE UITGEVERSONDERNEMING
Antwerpen-Apeldoorn

Oorspronkelijke titel: SPECTRUM Machine Code

Machine Code met de ZX SPECTRUM

Antwerpen-Apeldoorn
Vertaling: L.A.P. van den Wijngaert
ISBN 90 6215 089 6
D 1984/1997/2
UGI A 650
Boekhandelsrubriek 19
Boekbladrubriek 15

Inhoudsopgave

Voorwoord

Dit boek is een introductie tot de Z8Ø machinecode van de ZX Spectrum microcomputer. We stellen in deze introductie voorop dat u reeds een uitgebreide kennis van BASIC hebt verworven maar we gaan ervan uit dat u nog helemaal niets afweet van machinecode. We leren u aan de hand van eenvoudige voorbeelden en een aantal ontwerpen hoe u zelf routines in machinecode schrijft, hoe u ze in een BASIC-programma laat draaien, hoe u ze op tape moet SAVEn en hoe u ze later kunt LOADen.

Machinecode biedt het grote voordeel dat u er een aantal opdrachten snel mee kunt uitvoeren die, weliswaar met BASIC ook *mogelijk* zijn maar die in deze programmeertaal te traag lopen om binnen aanvaardbare normen te blijven. Het grote nadeel van machinecode zijn de hogere eisen die aan de programmeur worden gesteld; deze laatste moet immers op elk ogenblik precies weten waar de informatie in de machine opgeslagen zit, welke vorm deze gegevens hebben en hoe de Spectrum ze zal interpreteren.

Maar zodra u de machinecode onder de knie hebt, zult u tevens een heleboel over uw machine geleerd hebben!

We beginnen met een stukje "theorie": hoe worden getallen in de computer opgeslagen, wat met negatieve getallen, hoe werken binaire en hexadecimale codes?

Daarna bestuderen we de architektuur van een *vereenvoudigde* versie van de Z8Ø chip. Dit doen we opdat u vertrouwd raakt met de basisprincipes ("registers", "adresseermodes", de "indexering" en de "indirectie", de "programmateller" en de "stackpointer") en niet teveel aandacht hoeft te besteden aan minder ter zake doende details. Immers, mocht u onmiddellijk met de Z8Ø van start gaan, dan zou u al vlug ondervinden dat bij elk gemaakt statement rekening moet worden gehouden met attributieven als: indien, maar en misschien; de Z8Ø is inderdaad een complex zaakje en het zal veel eenvoudiger zijn de werking ervan te begrijpen door hem eerst tot iets eenvoudigers te reduceren.

Vervolgens wordt de "echte" Z8Ø beschreven en behandelen we in detail een belangrijke groep van commando's, de LOAD-instructies. Deze commandogroep wordt gebruikt om de verschillende "adresseermodes" in machinecode aan te leren.

We maken u ook duidelijk hoe u in machinecode moet opslaan (store), runnen (run), bewaren (save) en laden (load). Bovendien tonen we u hoe een BASIC-programma, dat al deze opdrachten zo eenvoudig mogelijk maakt, wordt uitgewerkt; alle andere programma's in dit boek gebruiken trouwens dit BASIC-programma.

Onder de behandelde routines treft u ook een machinecode vermenigvuldiger aan (die als voorbeeld dient bij tal van belangrijke commando's).

We leggen van a tot z uit hoe het beeldscherm van de Spectrum in elkaar zit en hoe het kan worden gemanipuleerd; in aparte hoofdstukken onderzoeken

we nuttige machinecode-routines voor de 'Attributes File'' en voor de ''Display File'' (hierbij besteden we eveneens de nodige aandacht aan zaken als scrollen, kleurveranderingen, patroongeneratie en aan het in- en uitschakelen van de FLASH).

Door middel van een routine die de regels opnieuw nummert worden de belangrijkste ''flags'' uitvoerig beschreven.

Twee krachtige commandogroepen worden geïntroduceerd: ''block search'' en ''block transfer''.

Deze laatste commandogroep wordt in verschillende scrollroutines toegepast.

In het laatste hoofdstuk worden een aantal extra's weggegeven die écht de moeite waard zijn.

De appendices bevatten verschillende tabellen met waardevolle informatie voor het in machinecode programmeren: hexadecimaal naar decimaal conversie, geheugenreservering, systeemvariabelen in hexadecimale code, de Z8Ø commando's en hun effect op ''Carry-'' en ''Zero flags'', hexadecimale codes (in een praktische alfabetische listing) en een waardevolle partiële assembler (HELPA) die in BASIC werd geschreven en die zonder veel moeite kan worden aangepast zodat u op een gemakkelijke manier zelf machinecode-routines kunt schrijven, editten en runnen. Opvallend aan deze partiële assembler is dat hij automatisch relatieve sprongen berekent zodat u hierdoor een boel rekenwerk uit handen wordt genomen.

Dit boek werd zó geschreven dat u de machinecode-routines kunt gebruiken *zonder* dat u snapt hoe ze werken, toch durven we de wens uit te spreken dat u een hoger en meer bevrediging schenkend doel zult nastreven, namelijk dat u zelf *uw* machinecode-routines zult leren schrijven...

1 Een voorproefje

U zou dit boek zeker niet hebben gekocht, of u zou het nu niet in de boekhandel staan door te bladen indien u niet op één of andere wijze aan de weet was gekomen dat de Spectrum op een uiterst snelle manier merkwaardige zaken kan doen in iets wat men machinecode noemt, en dit is inderdaad juist. Maar wat machinecode zo lastig maakt is dat u – in tegenstelling tot BASIC – het denkwerk voor uw rekening moet nemen. U moet veel meer aandacht besteden aan pietluttige details en u moet in het oog houden waar precies uw stukje code zich in de machine bevindt. Machinecode is dus beslist *niet* "gebruikervriendelijk"; bij het begin lijkt het meer op Egyptische hiëroglyfen en begrijpt u het niet beter, of is het even verstaanbaar als een telefoongids van één of ander Mohammedaans telefoondistrict.

In werkelijkheid is het allemaal niet zó erg en zult u er zich vrij vlug mee kunnen behelpen, hoewel het vrijwel zeker is dat het veel inspanning zal kosten voordat u de machinecode volledig onder de knie hebt. Om u ervan te overtuigen dat het de moeite waard zal zijn, beginnen we met enkele machinecode-routines die razendsnel bewegende displays van nogal abstracte aard voortbrengen.

Probeer niet te begrijpen hoe deze routines werken, dat komt later wel, u hoeft ze slechts te copiëren en te runnen. U zult een aantal toetsen nodig hebben die u waarschijnlijk tot nog toe maar weinig hebt gebruikt, namelijk:

USR (toets "L" in extended mode)
CLEAR (toets "X" in keyword mode)

en

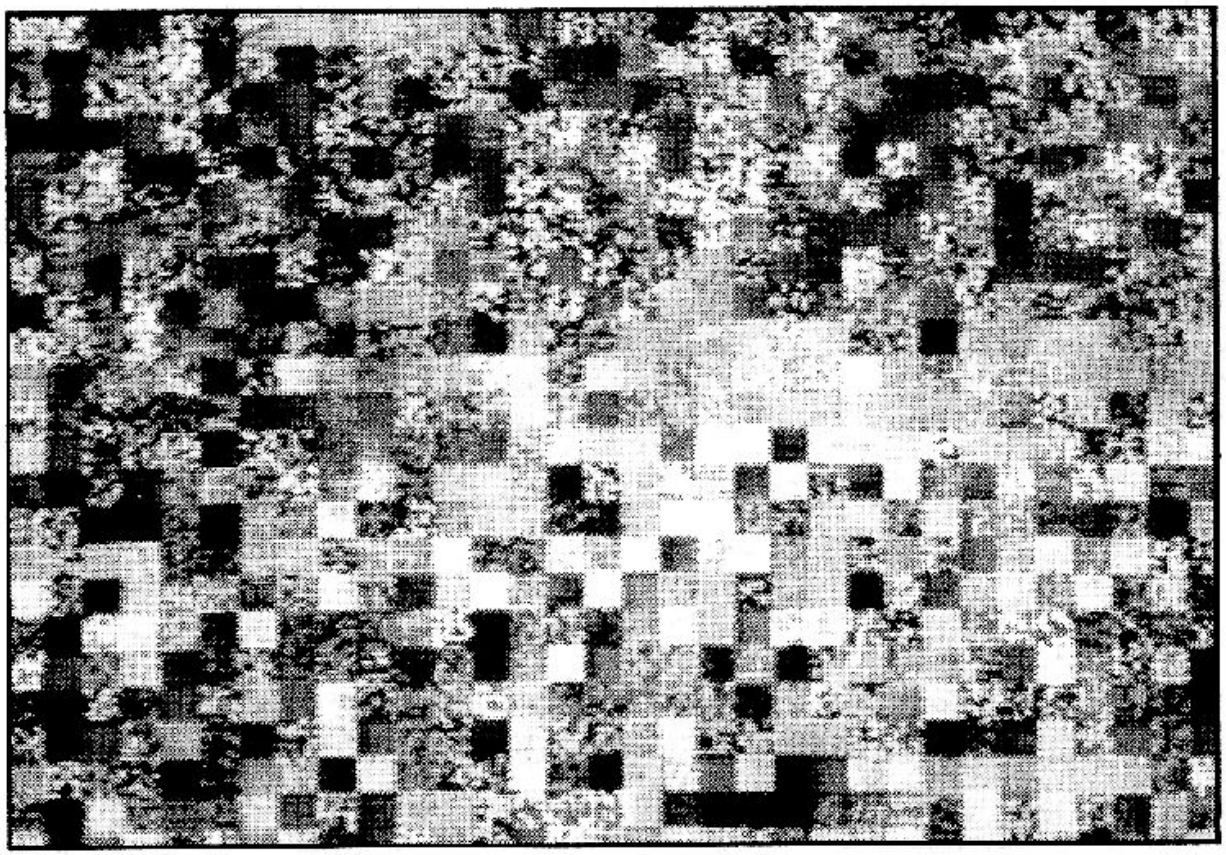
POKE (toets "O" in keyword mode)

Hier volgt het eerste programmaatje.

```
10  CLEAR 31999
20  BORDER 0
30  DATA 1, 0, 3, 17, 0, 88, 33, 0, 0, 237, 176, 201
40  FOR i = 0 TO 11
50  READ x
60  POKE 32000 + i, x
70  NEXT i
80  PAUSE 0
90  LET y = USR 32000
```

Zorg ervoor dat u alles juist overneemt, in het bijzonder de DATA-list en run het. Het scherm blijft leeg totdat u een toets indrukt (regel 80 is hiervoor verantwoordelijk.)

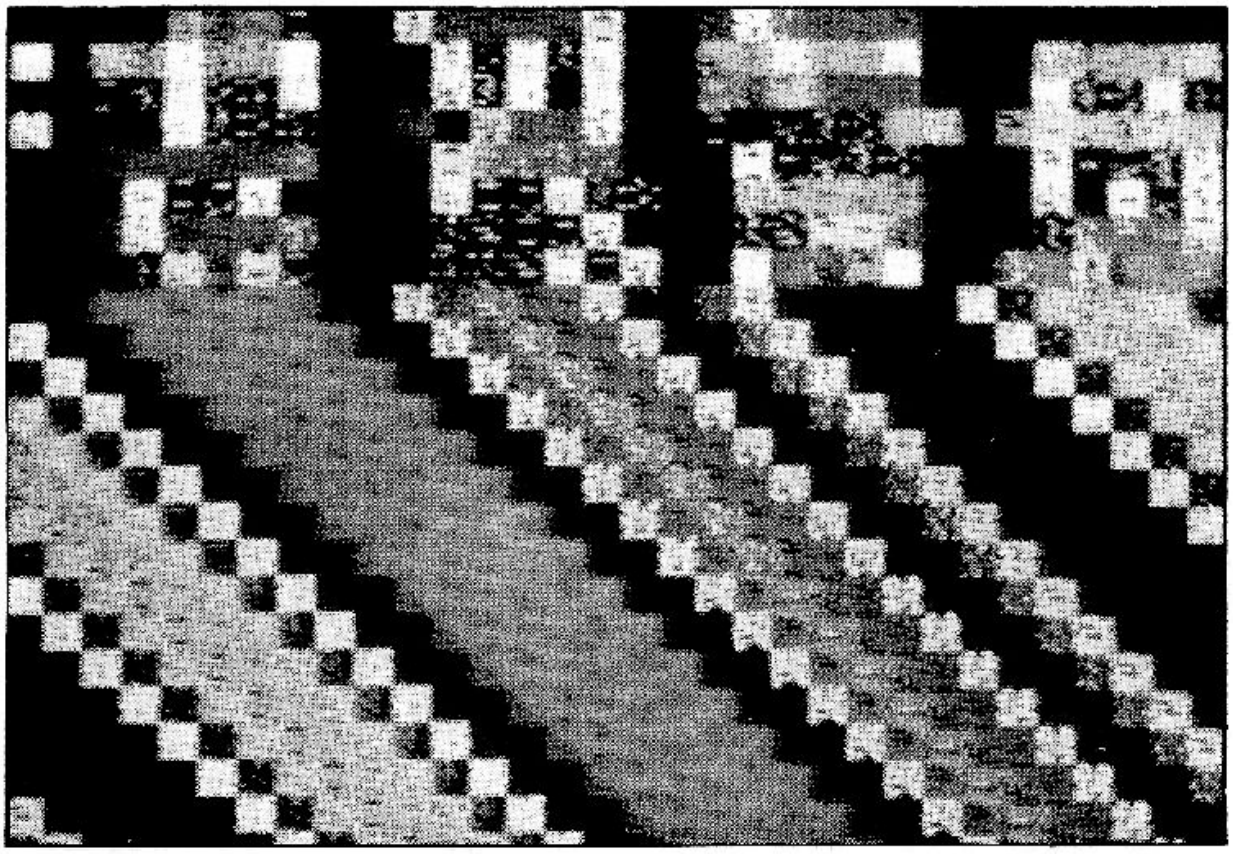
Druk en u krijgt een ogenblikkelijke respons: het scherm wordt gevuld met gekleurde vierkantjes waarvan er een aantal aan- en uitflitsen. (Indien dit niet het geval is moet u de listing opnieuw nakijken. Het is mogelijk dat u eerst de stekker uit het stopcontact moet trekken om het zaakje in orde te brengen: dit is één van de kneepjes van machinecode.)



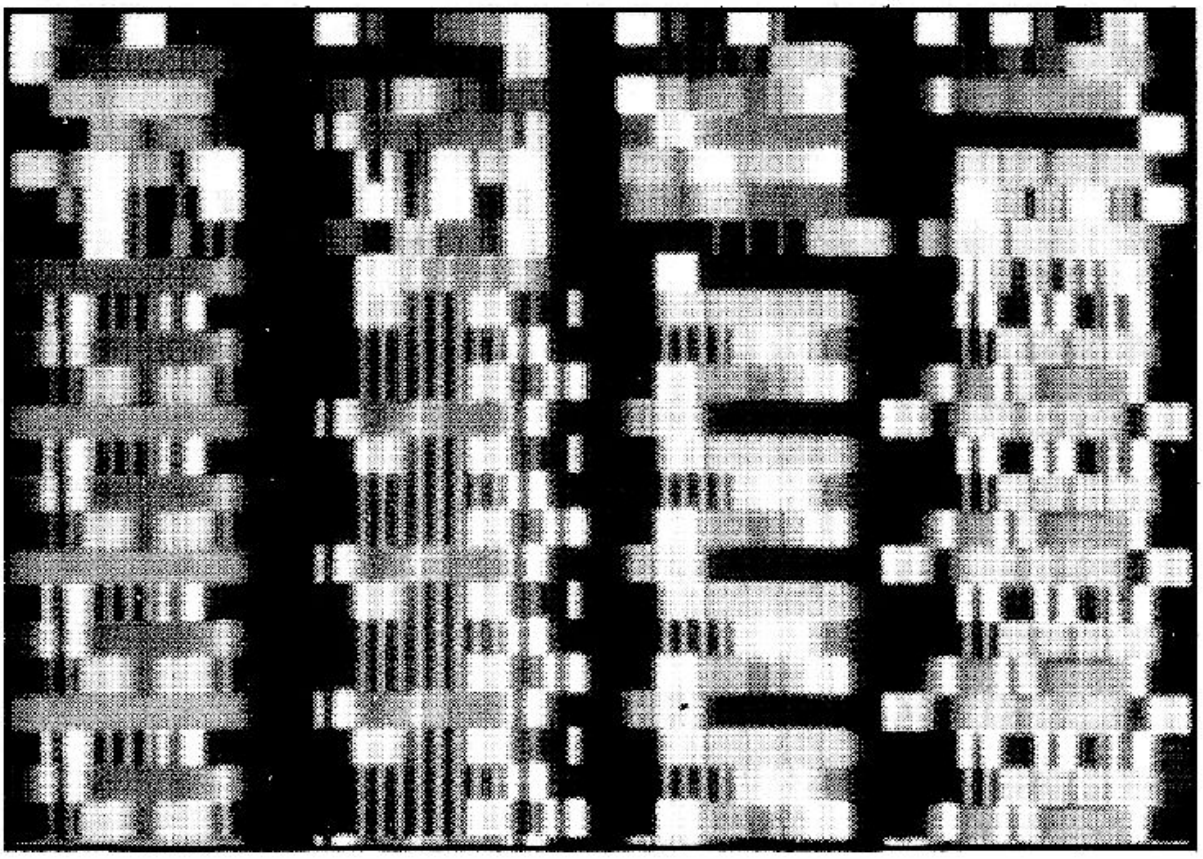
Figuur 1.1 Zijdelings scrollende vierkantjes van willekeurige kleur, sommige gespikkeld, andere helder, weer andere aan- en uitflitsend...

Dit gaat snel maar is toch nog niet spektakulair. In een volgende stap brengen we enkele wijzigingen aan in ons programma. Verwijder regel 90 en voeg het volgende toe:

```
100  LET t = 0: LET s = 0
110  IF t >= 256 THEN LET t = t - 256 * INT (t/256):
     LET s = s + 1
120  POKE 32007, t: POKE 32008, s
130  LET y = USR 32000
140  LET t = t + 1
150  GO TO 110
```



Figuur 1.2 ...nu komt er wat meer structuur in voor: snel roterende strepenpatronen...



Figuur 1.3 ...en we gaan alsmaar sneller...

Druk op RUN en een willekeurige toets en daar gaan we. U ziet een gelijksoortig display maar nu beweegt alles zijdelings met een redelijk hoge scrollsnelheid. Verander regel 40 in

```
140  LET t = t + 32
```

en het hele zootje scrollt opwaarts. Maak van regel 140

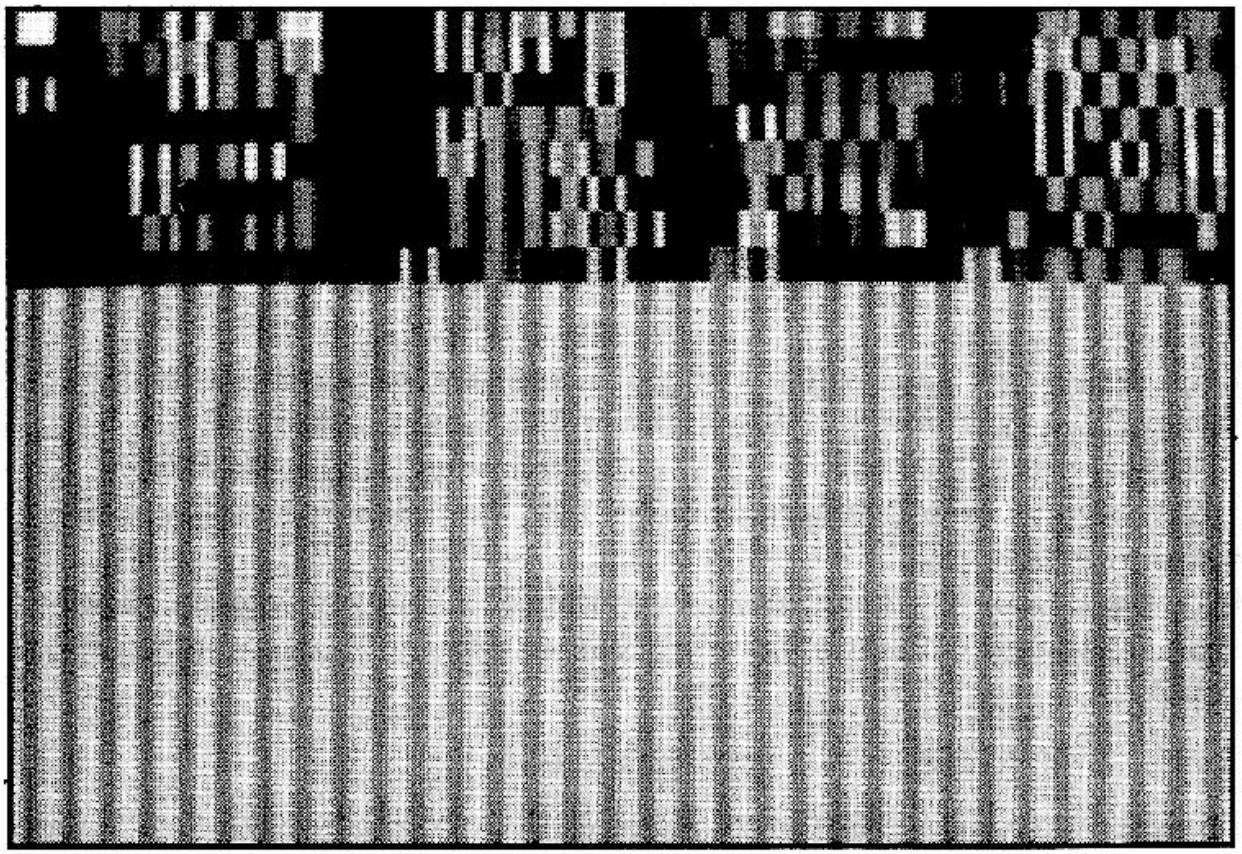
```
140  LET t = t + 31
```

en nu scrollt alles diagonaal. Probeer zelf nog enkele andere getallen die bij t worden opgeteld.

Vervolgens verandert u regel 140 opnieuw in LET t = t + 1. Wijzig het DATA-statement in

```
30  DATA 1, 0, 24, 17, 0, 64, 33, 0, 0, 237, 176, 201
```

en doe de procedure over. We verklappen niet wat u kunt verwachten; u ziet zelf maar! Wijzig regel 140 zoals hiervoor!



Figuur 1.4 ...een pauze nu met aardige groene banden...

LIGHT-SHOW

Tot nog toe was alles misschien wel ongebruikelijk maar zeker niet erg spektakulair. Nu combineren we beide routines en zorgen we voor een kleine fijnregeling:

```
10  CLEAR 31999
20  BORDER 0
30  DATA 1, 0, 24, 17, 0, 64, 33, 0, 0, 237, 176, 201
35  DATA 1, 0, 3, 17, 0, 88, 33, 0, 0, 237, 176, 201
40  FOR i = 0 TO 23
50  READ x
60  POKE 32000 + i, x
70  NEXT i
100 LET t = 0: LET s = 60
110 IF t >= 256 THEN LET t = t - 256 * INT (t/256):
    LET s = s + 1
120 POKE 32007, t: POKE 32008, s
125 POKE 32019, t: POKE 32020, s - 1
130 LET y = USR 32000
140 LET y = USR 32012
150 LET t = t + 1
160 GO TO 110
```

RUN dit en laat het programma gedurende een tweetal minuten draaien: aanvankelijk lijkt alles vreedzaam, maar dan breekt de hel los . . .

Verander regel 150 in t + 32, t + 31, enzovoort zoals voordien.

Verander de initialiseringen uit regel 100. Wat gebeurt er indien u begint met s = 0? s = 40?

In de buurt van s = 63 gaat alles zo snel dat het moeilijk te volgen is. Indien u regel 145 toevoegt merkt u dat er niets gebeurt indien u geen toets ingedrukt houdt.

```
145 IF INKEY$ = " " THEN PAUSE 0
```

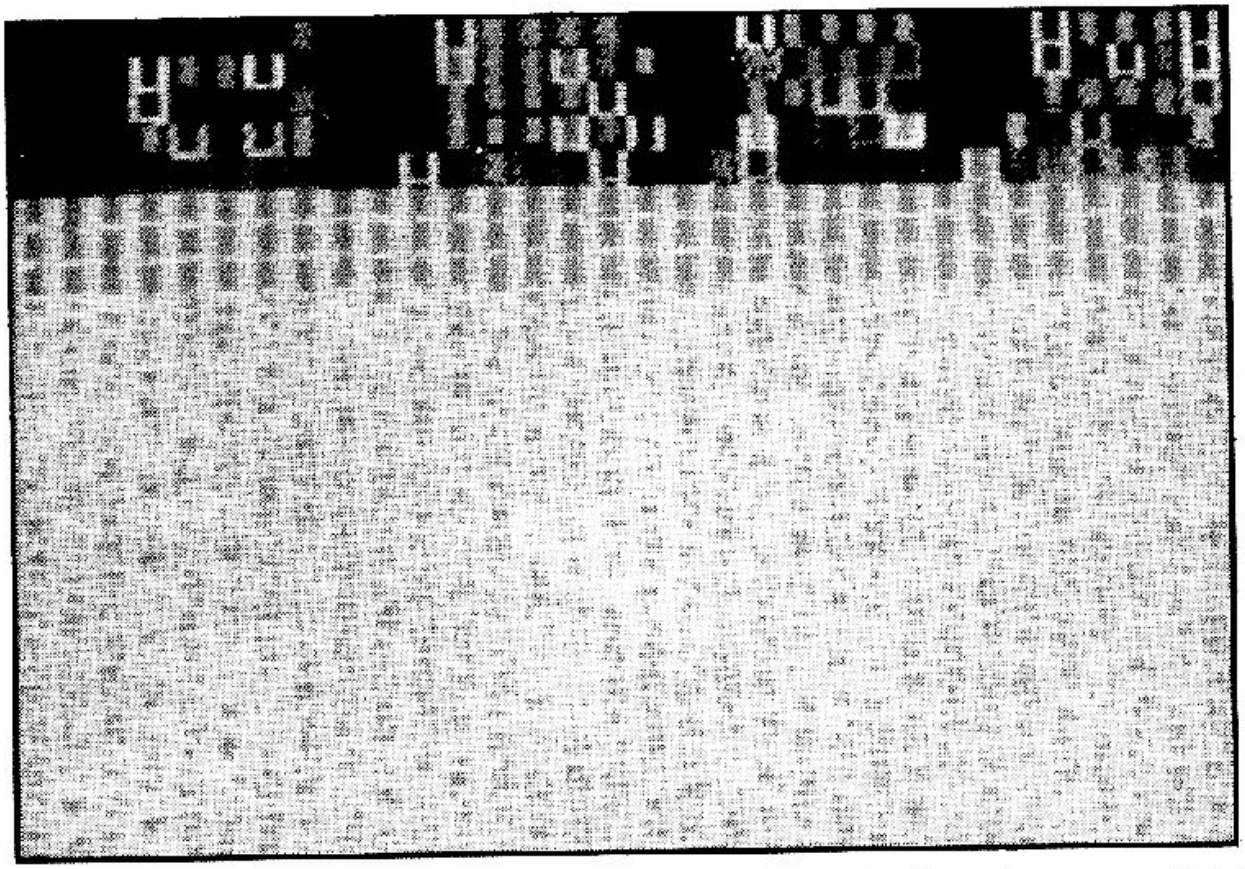
Op deze manier kunt u het programma op elk punt met "single-steps" verder laten lopen door snel een toets los te laten en weer in te drukken etc. . . .

Nu zult u een boel patronen zien die daarjuist veel te snel voorbijvlogen.

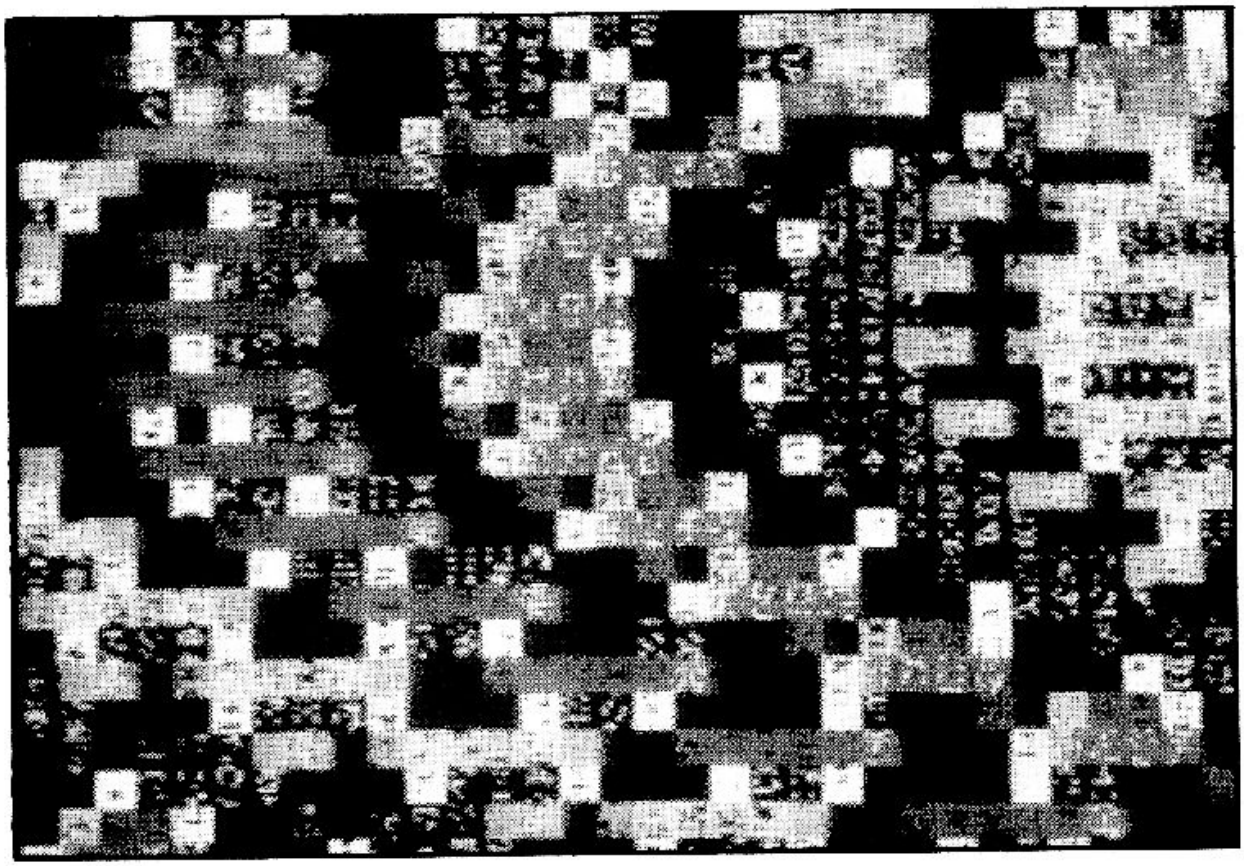
U kunt eindeloze variaties op dit programma maken, maar van de regels 30, 35, 130 of 140 moet u afblijven want daarin zit de machinecode vervat.

We hopen dat u er nu van overtuigd raakt dat machinecode iets extra te bieden heeft. Nochtans kan men van dit voorbeeld moeilijk zeggen dat het behalve het razendsnel voortbrengen van aardige patronen ook nog iets bruikbaars doet. Dit voorbeeld illustreert hoe een klein stukje machinecode in een totaal gedisproportioneerd effect resulteert. Om machinecodes nuttig te kunnen gebruiken moeten we ze leren schrijven in een geschikte gestructu-

reerde vorm en moeten we een specifiek doel nastreven (net als in een goed BASIC programma). Al wat volgt in dit boek is met dit doel geschreven.



Figuur 1.5 ...hier komen ze opzetten als hordes boze insekten die alles op hun weg verslinden!



Figuur 1.6 ...en voorts tot massieve kleurige krullen. Allemaal met twee dozijn bytes machinecode (en 16K ROM). En dit zijn slechts eenvoudige staaltjes!

2 Getallen in machinecode

Meestal beschouwen we getallen als een som van veelvouden van machten van tien. Indien we een getal, bijvoorbeeld 3814, opschrijven dan begrijpt iedereen dat dit betekent:

$$3 \times 1000 + 8 \times 100 + 1 \times 10 + 4 \times 1$$

en we stellen vast dat in een getal de "plaatswaarde" van een cijfer meer naar links wordt verkregen door met tien te vermenigvuldigen; daarom zeggen we dat een dergelijk getal wordt gegeven in *basis* tien.

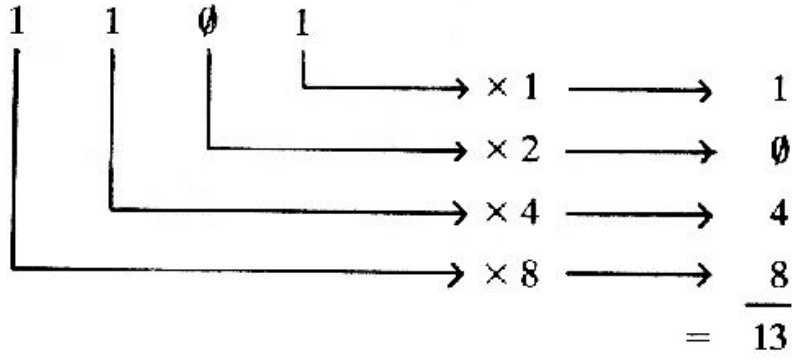
Omdat we, voor zover we ons herinneren, steeds op deze manier hebben geredeneerd, kunnen we ons moeilijk voorstellen dat er andere, minstens even goede, manieren bestaan om getallen te schrijven.

Kunnen we ook een talstelsel uitwerken dat slechts nullen en enen gebruikt?

Jawel. In een getal met basis tien is het grootst mogelijke cijfer de 9. Tel 1 op bij 9 en u krijgt 10 – hier heeft een *carry* (of overdracht) plaatsgehad. We kunnen alle getallen schrijven door gebruik te maken van een vrij te kiezen basis, en het grootst mogelijke cijfer dat in deze gevallen voorkomt zal steeds één kleiner zijn dan de basis. Indien de basis 2 is dan is het grootste cijfer 1, dus bevat een getal met basis 2 (d.i. een binair getal) slechts nullen en enen als cijfers (of "bits", van binary digits).

Hoe zit dat nu met de plaatswaarden? In basis tien verkrijgen we ze te beginnen met 1 (helemaal rechts) en door telkens met 10 te vermenigvuldigen wanneer we een plaats naar links opschuiven. Voor een binair getal beginnen we eveneens met de 1 rechts, maar telkens als we een plaats naar links opschuiven vermenigvuldigen we met 2.

We kunnen bijvoorbeeld het binair getal 1101 als volgt naar basis tien converteren (omzetten):



De conversie in de andere richting verloopt even eenvoudig. We nemen als voorbeeld het tiendelig getal 25 en we schrijven de binaire plaatswaarden op:

32 16 8 4 2 1

indien we van links naar rechts werken is het duidelijk dat we een 16 nodig hebben waardoor een 9 overblijft om de som 25 te vormen; deze 9 wordt dan samengesteld door een 8 en een 1, dus is 25 in ons binair talstelsel:

Ø 1 1 Ø Ø 1

HEXADECIMALE CODE

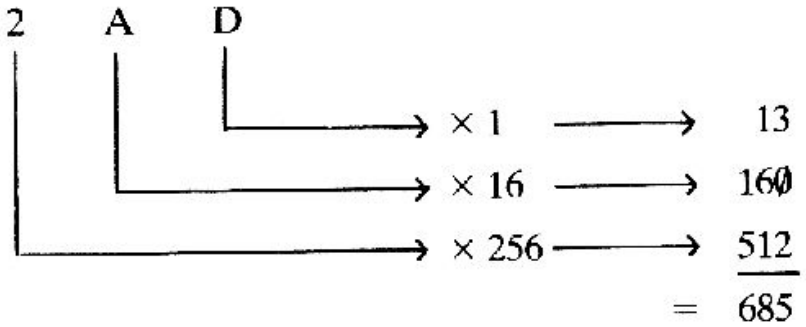
Dit is prachtig voor relatief kleine waarden maar nogal rommelig voor grote. Er bestaan enkele vlugge conversietechnieken en in het boekje *Basic met de ZX Spectrum* werden binair-naar-decimaal en decimaal-naar-binair conversie programmalistings opgenomen; maar nu onderzoeken we een werkwijze die steunt op de *hexadecimale* code omdat we deze in een verder stadium nodig zullen hebben.

Een getal in hex (of anders gezegd in hexadecimale) notering is een getal met basis 16, men verkrijgt dus de plaatswaarden door opeenvolgende vermenigvuldigingen met 16. De eerste vijf waarden zijn:

65536 4Ø96 256 16 1

"Probeer dat maar eens te volgen!" Dit zijn getallen waar u geen touw aan kunt vastknopen, en daarbij komt nog dat in basis 16 het grootste cijfer de waarde 15 heeft. "Hex maakt het nu wel erg moeilijk!" horen we de meesten al klagen.

Even geduld; we lossen het probleem van cijfers die groter zijn dan 9 op door aan de waardes 1Ø tot en met 15 de letters A tot en met F toe te kennen. Op deze manier wordt het hex getal 2AD als volgt in het overeenkomstig decimaal getal omgezet:



[≙ betekent: komt overeen met]

Om alles nog mooier te maken: omdat 16 één van de binaire plaatswaarden is (namelijk de vijfde: 1ØØØØ) heeft dit tot gevolg dat elk hex cijfer in een getal kan worden vervangen door de vier binaire cijfers die het hex cijfer samenstellen. (Men gebruikt dikwijls de term "bit" in de plaats van "binair getal" zoals men ook meer spreekt over "hex" in de plaats van "hexadecimaal".)

De volgende tabel geeft een overzicht van de conversies.

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

Een meer uitgebreide tabel geven we in appendix 1. *Opmerking:* omdat dit boek voor de Spectrum werd geschreven, kunnen de letters ofwel hoofdletters ofwel kleine letters zijn; voor de inputs moeten – om reden van overeenkomst *kleine* letters worden gebruikt (dus zonder CAPS LOCK).

Stel nu dat we het getal 9041 in hex willen omzetten. Eerst trekken we er tweemaal 4096 van af, daarna een aantal keer 256 enzovoort:

$$
\begin{array}{rrl}
 & 9041 & \\
2 \times 4096 = & \underline{8192} & - \\
 & 849 & \\
3 \times \ 256 = & \underline{\ 768} & - \\
 & 81 & \\
5 \times \ \ 16 = & \underline{\ \ 80} & - \\
 & 1 &
\end{array}
$$

$$1 \times \quad 1 = \frac{1 -}{\emptyset}$$

Derhalve wordt de hex voorstelling 2351.

Nu hoeven we slechts de cijfercodes uit de tabel te copiëren:

2	3	5	1
ØØ1Ø	ØØ11	Ø1Ø1	ØØØ1

en op deze manier hebben we het binaire equivalent van het decimale getal 9Ø41 gevonden, namelijk ØØ1ØØØ11Ø1Ø1ØØØ1, wat u verkrijgt door de vier opeenvolgende blokken aan elkaar te lassen.

De hex-naar-binair conversie is zo eenvoudig dat we – meer wél dan niet – de getallen in hex notering laten staan ook al hebben we de binaire notering nodig; dit doen we dan omdat bij het copiëren van een lange reeks nullen en enen gemakkelijk een foutje wordt gemaakt.

CONVERSIE DOOR DE COMPUTER

Hier volgt een programma dat decimale getallen in hex getallen converteert; dit programma deelt het getal door 16 en kijkt telkens naar de rest, dus vindt het de cijfers in de omgekeerde volgorde als hiervoor werd getoond!

```
2Ø  LET p = 4
3Ø  LET h$ = "ØØØØ"
4Ø  INPUT "dec. no. (max 65535)"; dn
5Ø  LET n = INT (dn/16)
6Ø  LET r = dn - 16 * n
7Ø  LET h$ (p) = CHR$ (r + 48 + 39 * (r > 9) )
8Ø  LET dn = n
9Ø  LET p = p - 1
1ØØ  IF dn > Ø THEN GO TO 5Ø
11Ø  PRINT "Hex value is:"; h$
```

Regel 7Ø zorgt ervoor dat de cijfers en de letters a tot en met f (we wensen hiervoor kleine letters te gebruiken) worden geselecteerd. Het geheel komt misschien nogal verwarrend over omdat de ASCII-code, die de Spectrum gebruikt om intern karakters voor te stellen, nogal gecompliceerd in zijn werk gaat. We willen als volgt tellen: . . . 7, 8, 9, a, b, . . . en het zou mooi zijn indien de ASCII-code voor "a" één groter zou zijn dan die voor "9". Ongelukkig genoeg is de ASCII-code voor "a" 4Ø groter, dus is ze 39 te groot.

Dus moeten we, voor karakters groter dan 9, 39 bijtellen. De logische uitdrukking "r > 9" heeft 1 als waarde indien ze waar is en Ø indien ze vals is; op

deze manier wordt de 39 slechts bijgeteld indien het een karakter uit het bereik a tot en met f betreft. (De 48 is nodig omdat de ASCII-code voor Ø gelijk is aan 48.)

Het resultaat wordt steeds voorgesteld als een getal met vier cijfers voorzien van de vereiste nullen. De letters worden als kleine letters weergegeven omdat we steeds hex cijfers zullen *invoeren* als kleine letters, op deze manier worden we er steeds aan herinnerd dat het hier om hex getallen gaat (en niet om variabelen of andere zaken). Het programma loopt vast indien het resultaat een hex getal met meer dan vier hex cijfers moet kunnen bevatten, maar voor ons is dat geen probleem omdat door de beperkte geheugencapaciteit van de Spectrum toch slechts getallen met vier hex cijfers toegelaten zijn.

Hier volgt de code voor de conversie in de andere richting (hex-naar-decimaal).

```
140  INPUT "Enter 4-digit number in hex"; h$
150  LET dn = Ø
160  FOR p = 1 TO 4
170  LET dn = dn * 16 + CODE h$ (p) - 48 - 39 * (h$ (p) > "9")
180  NEXT p
190  PRINT "Decimal value is:"; dn
```

Ook hier werd een spitsvondigheid gebruikt opdat regel 17Ø de juiste waarden zou opleveren; regel 17Ø is dan ook precies het omgekeerde van regel 7Ø.

Door middel van een klein "menu" kunnen we beide programma's met elkaar verbinden:

```
2   PRINT "Dec/Hex converter"
3   PRINT "1) DEC - > HEX
           2) HEX - > DEC
           3) END"
5   INPUT "Enter 1, 2, of 3"; sel
8   IF sel = 1 THEN GO SUB 2Ø
9   IF sel = 2 THEN GO SUB 14Ø
10  IF sel = 3 THEN STOP
12  PAUSE Ø: GO TO 2
```

Vanzelfsprekend zullen we RETURN's nodig hebben in regels 12Ø en 2ØØ.

3 Positief en negatief

Nu we reeds iets afweten van binaire getallen en hoe ze kunnen worden gemanipuleerd, doen we een stap terug om te zien hoe ze in de machine worden verwerkt. Meestal wordt een getal in een vast aantal binaire cijfers (bits) – dijkwijls 16 of 24 of 32, afhankelijk van de betreffende machine – opgeslagen. Dit aantal bits noemt men de *word size* (woordgrootte) van de machine.

We gaan even na hoeveel cijfers een 4-bit woord kan bevatten:

4-bit patroon	Decimale waarde
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Het is duidelijk waarom men in de praktijk een woordgrootte van meer dan vier bits verkiest; een machine die slechts de getallen 0 tot en met 15 kan voor-

stellen is verre van doeltreffend. Maar er zijn nog twee andere problemen; deze notatie kan geen gebroken waarden (bijvoorbeeld 7.14) en evenmin negatieve getallen weergeven.

Het probleem van de gebroken waarden zullen we achterwege laten omdat de meeste machinecode-routines slechts gehele getallen gebruiken, het probleem van de negatieve getallen is echter dringender.

De techniek is eenvoudig: indien u beschikt over de binaire voorstelling van een positief getal waarvan u het negatief equivalent wenst, dan moet u twee zaken doen:

1. Alle nullen verandert u in enen en van alle enen maakt u nullen (dit noemt men "flipping the bits").
2. Bij het resultaat telt u 1 op.

Een voorbeeld. Stel dat u -3 wenst voor te stellen.

In een 4-bit woord is 3 $\hat{=}$	ØØ11
na flipping krijgt u:	11ØØ
één bijtellen geeft:	+1
	11Ø1

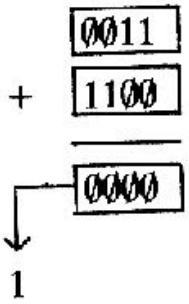
Omdat 11Ø1 het getal -3 voorstelt zegt men dat dit het twee *complement* is van ØØ11.

We treden niet in detail om te verklaren hoe dit in zijn werk gaat maar u kunt zelf nagaan of het inderdaad in dergelijke gevallen opgaat deze techniek toe te passen. Indien we immers bij -3 drie optellen (of $-5 + 5$ of iets gelijksoortigs) dan moet het resultaat van de bewerking nul opleveren. Even kijken of dat zo is:

	ØØ11	($\hat{=}$ 3)
+	11Ø1	($\hat{=}$ -3)
=	1ØØØØ	
	111	(Denk eraan dat bij het rekenen met binaire getallen 1 + 1 = Ø met overdracht 1!)

We krijgen dus *niet* helemaal ØØØØ, maar indien we met 4-bit woorden werken zal de laatst bijgekomen bit (de senior bit) afvallen zodat we de 4 junior bits overhouden die alle Ø zijn. (Vergelijk het met een kilometerteller van een wagen met vier wieltjes; indien de teller 9999 aanwijst en u rijdt nóg een kilometer dan wijst hij ØØØØ aan en is er een "1" aan de linkerkant "afgevallen".)

Eigenlijk moeten we het dus als volgt beschouwen:



Deze techniek werkt steeds indien we ervan uitgaan dat het aantal bits steeds onveranderd blijft. Vergeet niet voldoende nullen toe te voegen zodat het aantal bits van de standaardlengte wordt bereikt, alvorens u het twee complement neemt. (In dit geval dus 0011 in plaats van 11 om getal 3 voor te stellen.)

We schrijven nu de 4-bit waardetabel mét negatieve getallen.

Decimaal	Binair	2 complement	Decimaal
0	0000	0000	0
1	0001	1111	−1
2	0010	1110	−2
3	0011	1101	−3
4	0100	1100	−4
5	0101	1011	−5
6	0110	1010	−6
7	0111	1001	−7
8	1000	1000	−8
9	1001	0111	−9
10	1010	0110	−10
11	1011	0101	−11
12	1100	0100	−12
13	1101	0011	−13
14	1110	0010	−14
15	1111	0001	−15

We stellen vast dat er een probleem is: elk bitpatroon verschijnt tweemaal zodat, bijvoorbeeld, 1001 zowel 9 als −7 kan betekenen. We zullen derhalve het waardebereik nog verder moeten beperken. In de tabel werd een streep-

jeslijn getrokken rond het gebied dat we willen voorstellen. Kijken we naar de senior bit (de uiterst linkse) in elk bitpatroon dan stellen we vast dat deze "Ø" is indien het getal positief is en "1" indien het negatief is. Het ligt voor de hand dat we van dit onderscheid gebruik zullen maken.

Dus is het bereik van getallen dat we d.m.v. een 4-bit woord kunnen voorstellen begrepen tussen -8 en $+7$. Voor 5-bit woorden ligt het tussen -16 en $+15$ en voor 6-bits tussen -32 en $+31$ enzovoort.

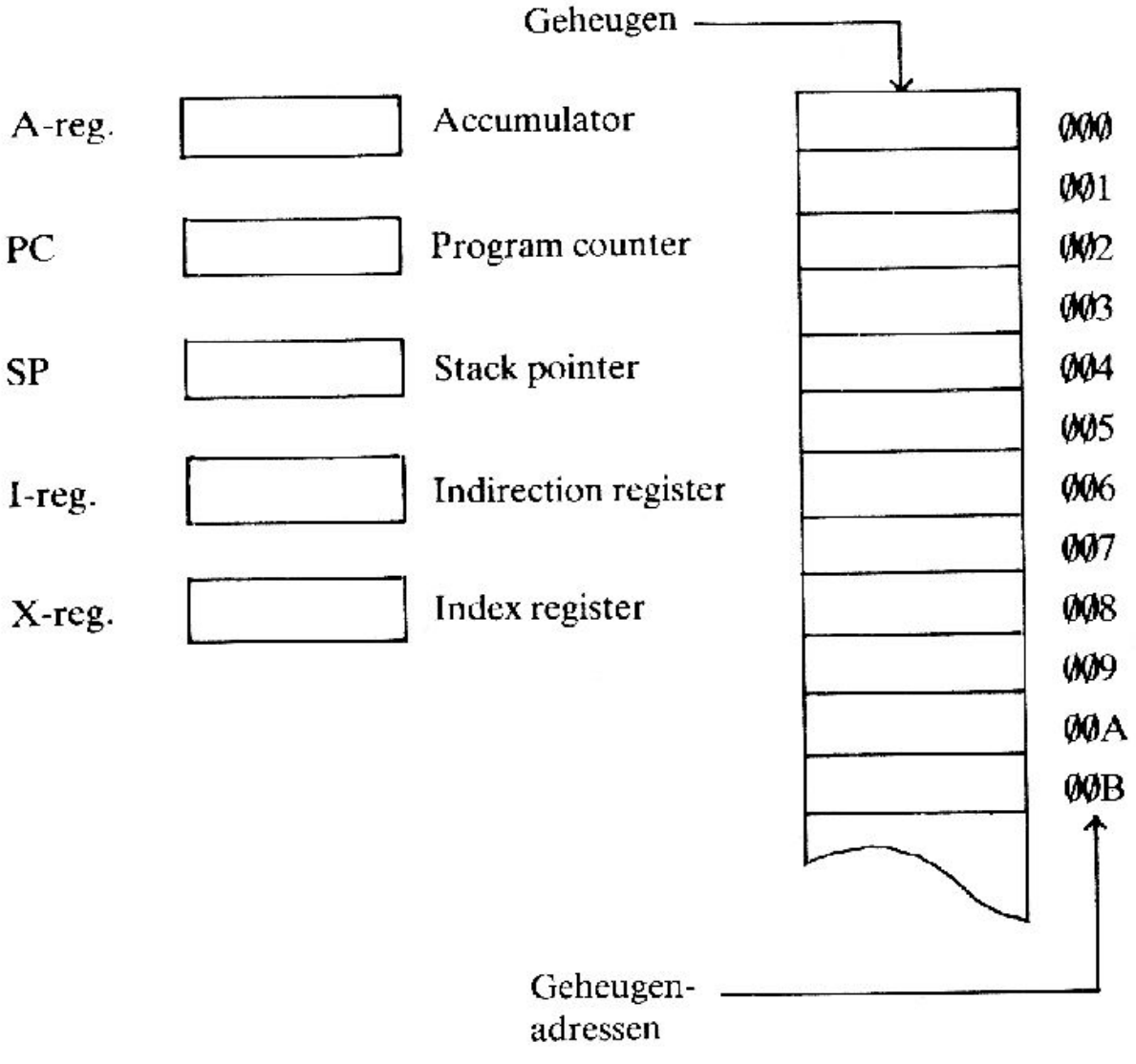
Een 16-bit woord (belangrijk voor zover het over een Z8Ø gaat!) omvat het bereik tussen -32768 en $+32767$. In appendix I wordt een tabel gegeven met de twee complementen van 8-bit woorden.

4 Machine-architectuur

We weten nu voldoende over getallen zodat we ons nu even gaan bezighouden met de manier waarop de machine ze verwerkt. Hiervoor moeten we eerst wat meer afweten van de interne structuur van de processor – zijn *architectuur*.

De Z8Ø processor is het produkt van een vijf en twintig jarige evolutie op computergebied en is daardoor een vrij gesofisticeerd werktuig. Voor de beginner is dit natuurlijk een minder gunstig uitgangspunt, vandaar dat we eerst een eenvoudige processor zullen beschrijven. We doen dit om de relevante concepten in te leiden, die van belang zijn bij alle op de markt zijnde apparaten, zonder dat we ons zorgen hoeven te maken over de finesses; die volgen later wel (in hoofdstuk 7 en verder beginnen we daarmee).

We gaan ervan uit dat onze denkbeeldige machine een geheugen heeft met 16-bit woorden en een aantal 16-bit registers voor speciale doeleinden bevat; dat wordt als volgt voorgesteld:



We bekijken eerst het geheugen van nabij. In BASIC konden we deze geheugenlocaties een naam geven die we mooi vonden klinken, maar dat laat de naakte machine niet toe, ze staat erop dat elke locatie wordt genummerd, te beginnen bij nul, zoals we reeds hebben gedaan. Deze getallen noemt men de geheugenadressen en ze werden in hex genummerd, hoewel u steeds in het achterhoofd moet houden dat de uiteindelijke codering binair zal zijn.

Wat kan een geheugenwoord omvatten? Het antwoord is: een willekeurig patroon van 16 bits. Dat is nogal duidelijk, maar waar we naartoe willen is het volgende: we kunnen die 16 bits elke betekenis geven die we zelf kiezen. Indien we deze bits coderen als een twee complement van een geheel getal dan bevat een woord een getal uit het bereik −32768 tot en met +32767. Of we kunnen de bits een positief geheel getal zonder tekenbit laten voorstellen, dan hebben we een getal uit het bereik van Ø tot en met 65535. We kunnen, indien we dit willen, een woord splitsen in twee 8-bit getallen; elk getal stelt dan een letter uit het alfabet, een leesteken of een grafisch symbool voor.

Kijken we even naar de registers voor speciale doeleinden. Het A-register wordt aangesproken telkens als men aan rekenkunde doet. Het resultaat van een som wordt door de machine in het A-register gestopt. (Tussen haakjes, men noemt dit register wel eens de *accumulator*.) De meeste rekenkundige bewerkingen hebben betrekking op twee waarden; het is dus fout de machine 3 + te laten uitwerken, u moet meedelen waarbij de 3 moeten worden opgeteld. Eén van deze waarden moet zich in het A-register bevinden alvorens de optelbewerking wordt uitgevoerd. U kunt de volgende instructie schrijven:

```
ADD (1A3)
```

de machine vat dit als volgt op:

1. Tel de inhoud van geheugenlocatie 1A3 op bij de inhoud van het A-register. (De haakjes rond 1A3 worden gebruikt om aan te geven dat het gaat om de *inhoud* van geheugenlocatie 1A3 en niet om het *getal* Ø1A3.)
2. Plaats het resultaat terug in het A-register.

We hebben hiermee onze eerste instructie op machineniveau geschreven; deze instructie staat nog niet in machinecode maar het lijkt er toch al sterk op. Deze instructie bestaat uit een code voor de bewerking (een operatiecode), ADD, en uit een adres, 1A3. Vele instructies hebben deze gedaante. Vermelden we tenslotte nog dat het woord operatiecode gewoonlijk wordt afgekort tot *opcode*.

EEN OPTELPROGRAMMA

We bedenken een sequentie machine-instructies die het volgende BASIC-statement samenstellen:

```
LET R = B + C
```

Eerst moeten we actuele adressen toekennen aan R, B en C. Stel dat dit

respectievelijk de adressen 103, 104 en 105 zijn. We moeten ervoor zorgen dat de inhoud van adres 104 in het A-register terechtkomt. Om dit te realiseren bedenken we een LD-instructie (LD van load accumulator), dus:

LD (104)

Daarna moet de inhoud van adres 105 erbij worden opgeteld:

ADD (105)

en tenslotte moeten we op één of andere manier de inhoud van het A-register kunnen opslaan in adres 103. We bedenken daarvoor een opslag ("store")-instructie:

ST (103)

En voor we het goed en wel beseffen hebben we een eenvoudig programma op machineniveau geschreven dat bestaat uit de drie instructies:

LD (104)	[laad B in het A-register]
ADD (105)	[tel er C bij op]
ST (103)	[plaats het resultaat in R]

Hoe kunnen we nu een dergelijk programma op de machine laten lopen?

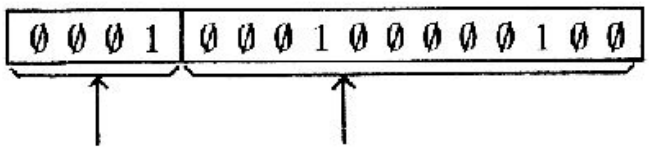
We vinden het heel normaal dat een programma in de machine wordt opgeslagen *vóór* het wordt uitgevoerd, en we willen dat het daar blijft zitten totdat we het nodig hebben. Op dezelfde manier moet ook een programma op machineniveau eerst worden opgeslagen. De meest geschikte plaats om een instructie op te slaan is in een geheugenwoord. (Denk eraan dat "woord" hier de betekenis heeft die u er zelf aan geeft!) Dit brengt met zich mee dat de opcodes LD, ADD, enzovoort als bitpatronen moeten worden gecodeerd. Niets is eenvoudiger; we moeten slechts, op willekeurige basis, een tabel met bitpatronen samenstellen en telkens als we een nieuwe opcode bedenken, plaatsen we die in de tabel. Dit kan bijvoorbeeld als volgt gebeuren:

Opcode	Binaire code
ADD	0000
LD	0001
ST	0010

In bovenstaande tabel hebben we aangenomen dat alle opcodes een 4-bit binaire code hebben. Hiermee kunnen we 16 verschillende patronen samenstellen en zijn er dus 16 verschillende instructies mogelijk. Volgens hedendaagse normen is dit een nogal beperkte instructieset, maar voor onze hypothetische computer is het voldoende. We hebben 16 bits in een woord, dus blijven er 12

over voor het adresgedeelte van de instructie.

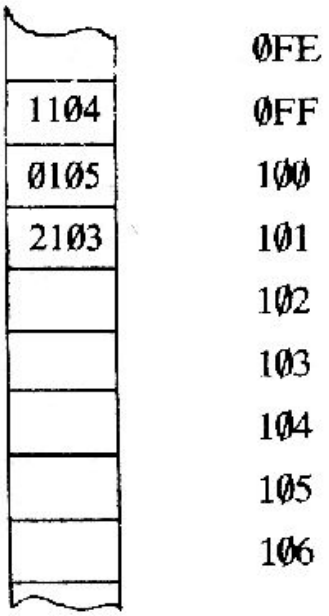
In de machine ziet de instructie LD (1Ø4) er zó uit:



Zo zien alle bitpatronen eruit. Van nu af schrijven we de hex versies van instructies, omdat die niet zo omslachtig zijn, maar onthoudt dat de machine ze onder bovenstaande vorm aangeboden krijgt.

DE PROGRAMMATELLER

Stel dat we ons programmaatje met drie instructies van geheugenlocagie ØFF af beginnen op te slaan.



Nu moeten we de computer op een of andere manier kunnen zeggen: "Voer de instructie uit die in ØFF staat, doe dan wat in 1ØØ staat en vervolgens wat in 1Ø1 voorkomt." Daartoe dient de *programmateller* (program counter) of PC-register. De PC is voor de computer een soort leeswijzer. Het programma wordt gerund door de PC op het adres van de eerste instructie te initialiseren. Tijdens de uitvoering van deze instructie wordt de PC automatisch met 1 geïncrementeerd zodat het systeem – wanneer het terugkeert om de PC te onderzoeken – de volgende instructie zal uitvoeren, enzovoort.

Nochtans schuilt hier gevaar in. Terwijl de laatste instructie (in 1Ø1) wordt behandeld, wordt de PC zoals altijd met 1 bijgewerkt, dus zal de machine wanneer ze PC opnieuw opzoekt 1Ø2 vinden en de instructie die daar voorkomt, uitvoeren. Welke instructie? We hebben immers geen instructie in 1Ø2

gezet! Vergeet niet dat er in 1Ø2 een bitpatroon zit dat daar door een vorig programma is achtergelaten of dat er een willekeurig bitpatroon is terechtgekomen door de machine in te schakelen. De computer zal dit patroon als een instructie interpreteren want dat hebben we hem opgedragen te doen. Dus loopt de machine door de locaties 1Ø3, 1Ø4 en 1Ø5 en dit zijn de locaties waar we gegevens hebben opgeslagen! Indien het getal in 1Ø4 bijvoorbeeld 2ØFF is dan interpreteert de computer dit als:

ST (ØFF)

en zal hij de inhoud van het A-register in ØFF copiëren waarbij de eerste instructie van ons programma wordt vernietigd! We hebben dus een "halt"-instructie nodig (als mnemonisch hulpmiddeltje gebruiken we hiervoor HLT) die het bijwerken van de PC stopzet. Nu wordt het programma:

LD (1Ø4)

ADD (1Ø5)

ST (1Ø3)

HLT

Hierbij moeten we een belangrijke opmerking maken. Juist *omdat* we woorden gebruiken die op verschillende ogenblikken verschillende betekenissen hebben, moeten we zeer goed in het oog houden welke gevolgen de machine zal trekken uit wat we haar opdragen te doen. Indien we haar vragen de inhoud van een locatie bij het A-register op te tellen, dan veronderstelt de machine dat die locatie een getal bevat. De machine voert geen tests uit, dat kan ze niet en daarenboven kan om het even welk bitpatroon een getal voorstellen. Op analoge wijze kan om het even welk bitpatroon een instructie voorstellen; in dit geval zal – indien de PC naar een locatie wijst – de inhoud van deze locatie als een instructie worden uitgevoerd.

Een vuistregel is: *houdt gegevens en programma's goed van elkaar gescheiden*: Indien u dit niet doet kunt u ervan op aan dat u regelmatig zult worden beetgenomen. Zoals we reeds hebben gezegd kan een heel programma spoorloos verdwijnen terwijl het loopt!

5 Sprongen en subroutines

Tot nu toe beschikten we over een nogal beperkte instructieset. We kennen reeds de LD- en ST-instructies die een aantal verplaatsingen in het geheugen bewerkstelligen, de ADD-instructie die erg primitieve rekenkunde mogelijk maakt en de HLT-instructie waarmee we alles kunnen stilleggen.

We voeren het rekenkundig vermogen van de microprocessor een beetje op door de SUB-instructie toe te voegen die de inhoud van een locatie aftrekt van de inhoud van het A-register. Maar hierbij blijft het wat de rekenkundige bewerkingen betreft; dus geen vermenigvuldiging, geen deling en zeker geen worteltrekking.

Wat we, voor alles, nodig hebben is een instructieset met vertakkingsinstructies die equivalent zijn met de IF ... THEN-instructies in BASIC.

SPRONGINSTRUCTIES

Het is vrij eenvoudig om naar een instructie over te gaan die niet voorkomt in de gebruikelijke opeenvolging van instructies. Om dit te realiseren moeten we de inhoud van de PC wijzigen d.m.v. een instructie als:

```
JP 416          [jump to (spring naar) 416]
```

Telkens als deze instructie wordt uitgevoerd, plaatst ze de waarde 416 in de PC. Het systeem wordt als het ware "om de tuin geleid" door aan te geven dat de volgende instructie in 416 zit, daarna gaat het naar 417, 418 enzovoort totdat een volgende "sprong"-instructie wordt tegengekomen. Natuurlijk kan elk adres volgen op de JP-opcode.

Deze instructie lijkt meer op de GO TO-instructie dan op een IF... THEN...-instructie. Om deze laatste instructie te "benaderen" hebben we een instructie nodig die de PC slechts beïnvloedt indien aan een bepaalde voorwaarde wordt voldaan. De eenvoudigste test die we kunnen uitvoeren is nagaan of het A-register de waarde nul bevat. Een instructie voor deze test is:

```
JPZ 2A7         [spring naar 2A7 indien het A-register Ø bevat]
```

Een andere testinstructie is:

```
JPN 14E         [spring naar 14E indien de inhoud van het A-regis-
                ter negatief is]
```

Met deze instructies komen we al een heel eind verder want nu kunnen we ook de test op een positief (van nul verschillend) getal formuleren door na te gaan wanneer in het programma geen sprong wordt gemaakt na een JPZ- *en* een JPN-instructie.

SUBROUTINES EN STACKS

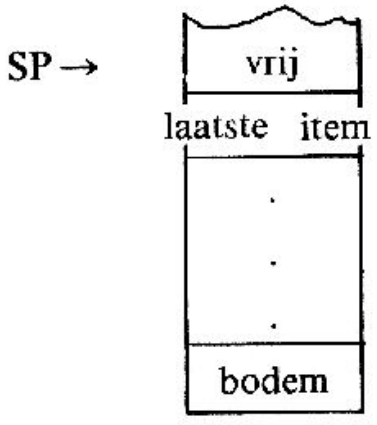
Nu we toch bezig zijn met de stuuroverdracht van de ene plaats in het programma naar een andere is het misschien wel interessant ons eens te buigen over zaken die overeenkomen met de BASIC-commando's GO SUB en RETURN.

Hiervoor hebben we de instructie:

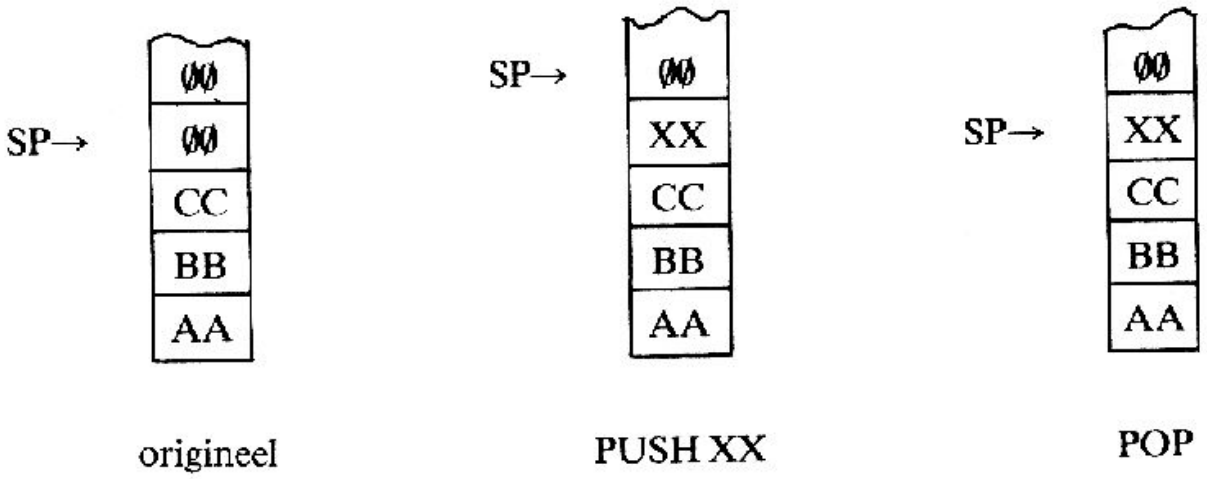
```
CALL 2Ø5          [roep de subroutine aan die begint op 2Ø5]
```

Wat doet deze instructie? Het is duidelijk dat ze de waarde 2Ø5 in de PC stopt, maar daarvoor konden we een JP-instructie gebruiken!

De CALL-instructie doet nog iets meer: ze slaat ook het adres op van de instructie die voorkomt na de CALL. Op deze manier kan de CALL-instructie het opgeslagen adres opnieuw in de PC laden wanneer een "return" (opcode RET) wordt tegengekomen. Het programma wordt dan, na de sprong, verder doorlopen vanaf de instructie die na de CALL voorkomt. Dit is slechts mogelijk door gebruik te maken van een *stack*. Een stack is een geheugensegment met een vaste "bodem" en een veranderlijke "top". Een stackpointer, SP, bevat het adres van de top; men noemt dit een "pointer" (of aanwijzer) omdat hij de top van de stack aanwijst en wel op volgende manier:



Door de SP één plaats omhoog te brengen en een nieuw item in het geheugen te brengen kan men extra items op de stack duwen (PUSH); men kan ze er afhalen (POP) door de SP één naar beneden te brengen. (Het is niet noodzakelijk dat men het opgehaalde item uit het geheugen wist omdat stackroutines alles negeren wat boven de SP voorkomt.) Bijvoorbeeld:



De stackpointer wijst dus de eerste *niet gebruikte* stacklocatie aan. (Terloops vermelden we dat de stack van de Z8Ø in de Spectrum naar beneden toe aanwijst en niet naar boven toe; maar daarover hoeft u zich geen zorgen te maken omdat dit voor de gebruiker toch van geen belang is daar hij nooit de actuele stackadressen nodig heeft.)

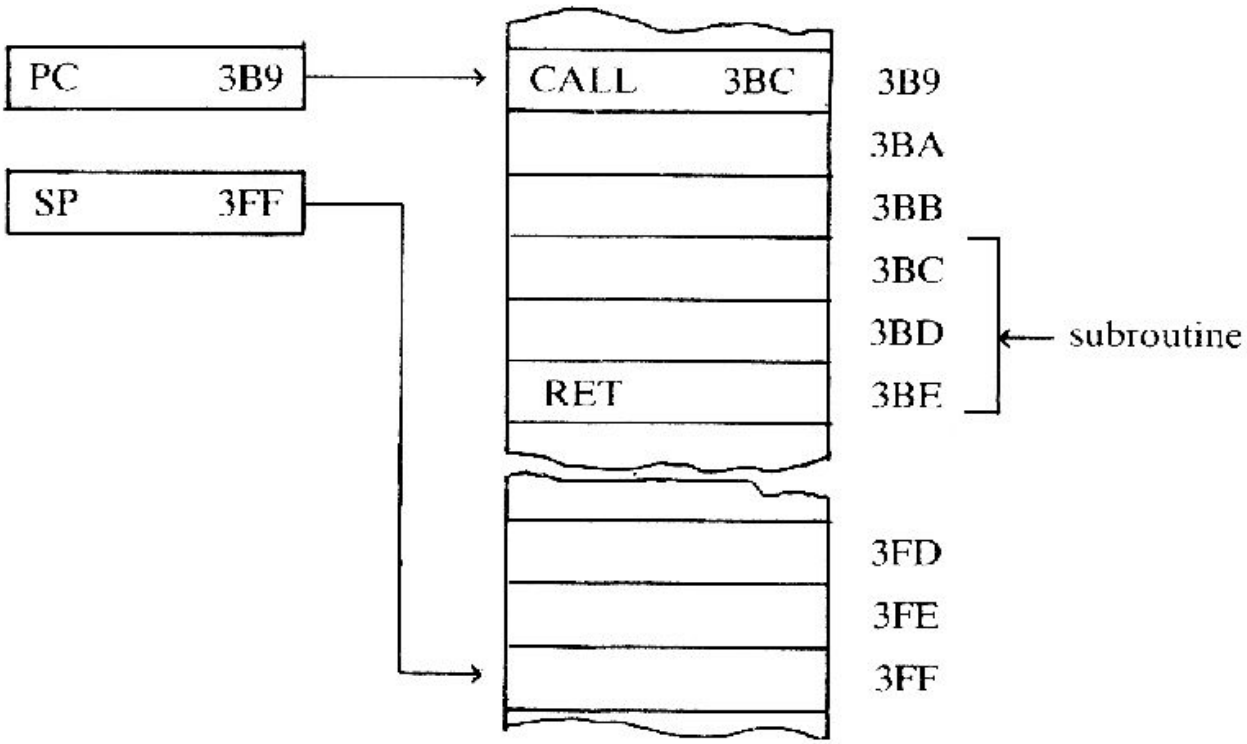
Stacks werken volgens het principe "laatst in, eerst uit". Vergelijk dit met een stapel boeken op een bureau: PUSH betekent "leg nog een boek op de stapel" en POP betekent "neem een boek van de stapel". Indien u dus achtereenvolgens drie items X, Y en Z PUSHt:

```
PUSH X
PUSH Y
PUSH Z
```

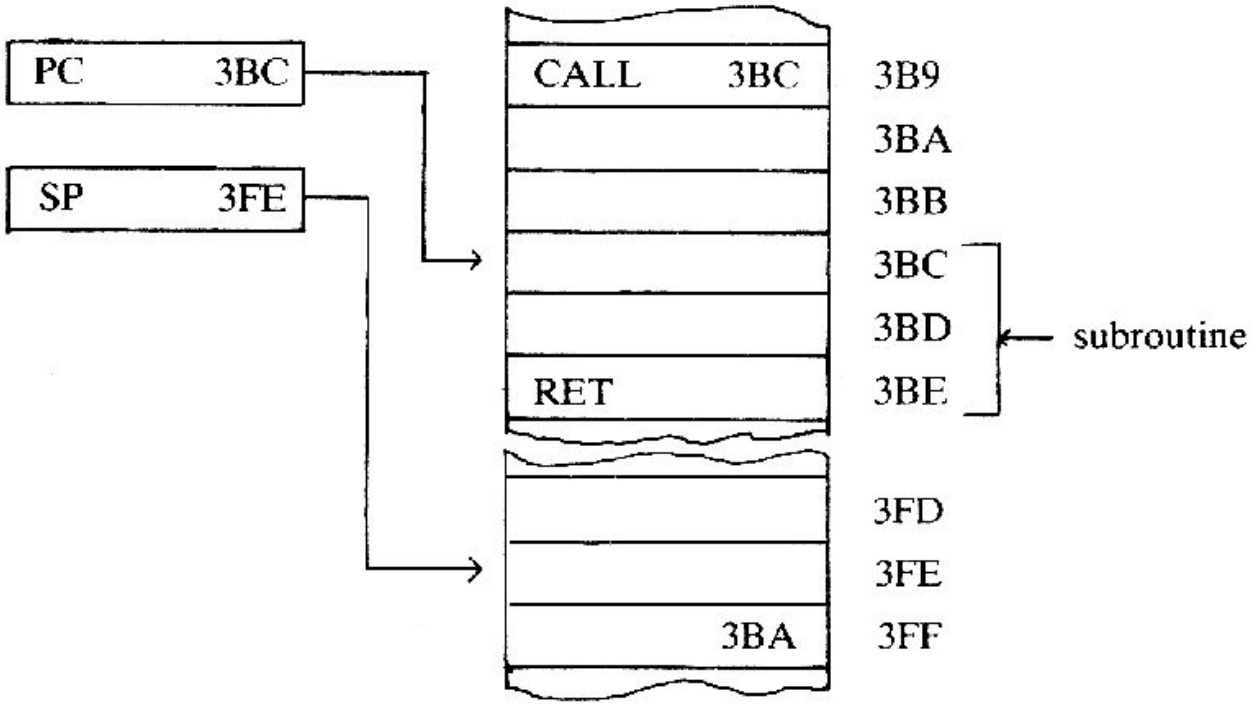
dan moet u om ze in de juiste volgorde van de stack te halen als volgt POPen!

```
POP Z
POP Y
POP X
```

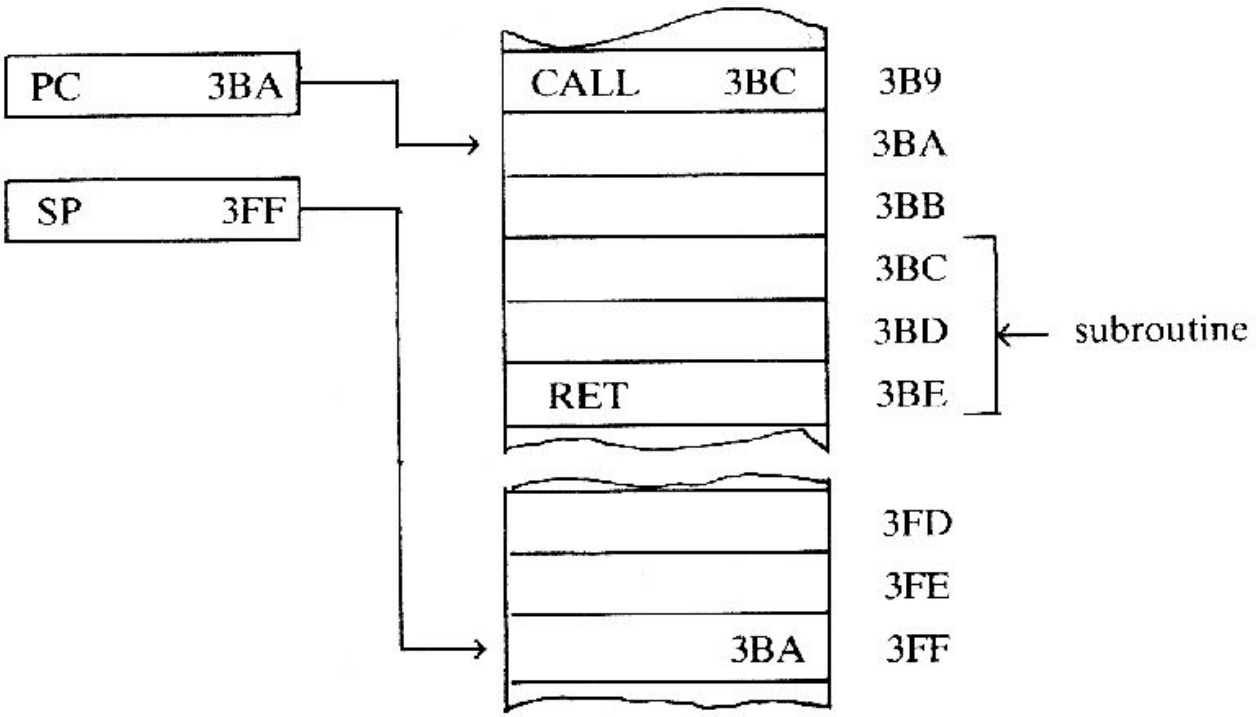
Een subroutine gebruikt de stack om zijn returnadres te onthouden. Wanneer een CALL wordt uitgevoerd, wordt het returnadres (d.i. het adres van de CALL + 1) op de stack geduwd. Wanneer de RET wordt bereikt, wordt de stack in de PC gePOPT. Een voorbeeld:



Nu gaat de CALL worden uitgevoerd...



Nu is de CALL volledig uitgevoerd en bevindt het returnadres zich in de stack. Het programma doorloopt de subroutine totdat het de RET tegenkomt waarna:



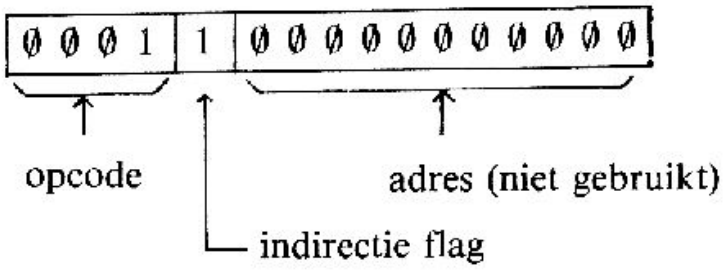
en nu bevindt de sturing zich terug in het hoofdprogramma.

6 Indirectie en indexering

Resten ons slechts twee registers te bespreken die soortgelijke functies hebben. Ze kunnen n.l. beide het adresgedeelte van een instructie wijzigen terwijl het programma loopt.

INDIRECTIE

We bekijken eerst even hoe het I-register dit doet. We bedenken een nieuwe opcode, LDI of "load indirect". Aan deze instructie wordt, net als met HLT, geen adres toegevoegd. Voor de machine werkt deze instructie als een LD-instructie behalve dat de hoge bit van het adresveld op "1" wordt geset. Deze bit noemt men de *indirectie flag* en hij is voor de machine slechts een aanduiding dat er indirectie in gebruik is. Derhalve is de binaire vorm van de LDI-instructie de volgende:



De hex code is 18ØØ. Wanneer de machine deze instructie tegenkomt, gebruikt ze het getal (wat dit ook moge zijn) uit het I-register als effectief adres. Indien het I-register 1E4 bevat en er wordt een LDI-instructie uitgevoerd dan heeft dit precies hetzelfde effect als de instructie LD 1E4. Met andere woorden het I-register speelt de rol van geheugenpointer en we zouden er ons naar hartelust in kunnen begeven indien we er rekenkunde mee aankonden. Dit betekent waarden naar het A-register verhuizen, want dat is de enige plaats waar we kunnen rekenen. We bedenken dus een opcode XAI of "exchange contents of A-register with contents of I-register" (verwissel de inhoud van het A-register met die van het I-register).

De indirectie flag kan natuurlijk worden geset voor elke instructie die een adresgedeelte heeft, dus kunnen we STI, JPI, ADDI enzovoort hebben, en in elk geval zullen de laatste drie cijfers van de hex code 8ØØ zijn.

EEN VOORBEELD

Kijken we weer naar een voorbeeld dat de volgende ideeën bundelt. Stel dat we een één dimensionale rij met lengte 2Ø wensen te initialiseren die de getallen 2, 4, 6, 8, . . . , 4Ø bevat. We willen dus, met andere woorden, een machinecode equivalent opstellen voor de BASIC-statements:

```
FOR c = 1 TO 2Ø
LET a(c) = c * 2
NEXT c
```

Er zullen om dit te doen lukken in elk geval een aantal waarden ergens in het geheugen een plaatsje moeten krijgen. Dit zijn 1 (omdat de lustelling met stappen van 1 omhoog gaat), 2 (omdat dit het increment is voor de array-inhoud) en 2Ø (die nodig is voor de eindtest van de lus). Voor het ogenblik wensen we ons niet bezig te houden met de plaats in het geheugen te zoeken waar deze waarden moeten worden opgeslagen, dus zullen we tijdelijk d.m.v. namen naar adressen verwijzen (in de aard van BASIC-namen). Deze namen zullen we achteraf, wanneer we uiteindelijk tot de machinecode komen, omzetten in getallen. Dit is een toepassing van de eerste computerwet van Jansen die zegt dat "je nooit moet uitstellen tot morgen wat je ook tot overmorgen kunt uitstellen." We gaan er dus van uit dat de gewenste getallen tot onze beschikking zijn in de locaties N1, N2 en N2Ø. Op dezelfde manier zullen we een locatie, BASE (basis), hebben die het adres bevat van het eerste element van de array en een locatie, COUNT, die dienst zal doen als lusteller.

Eerst en vooral zetten we het I-register zo dat het de basis van de array aanwijst:

```
LD    BASE
XAI
```

Nu stellen we de lustelling in op 1:

```
LD    N1
ST    COUNT
```

Vervolgens verdubbelen we dit (door het in het A-register op te tellen) en slaan we het op in de locatie die door het I-register wordt aangewezen. (Voorlopig zullen we spreken van "opslaan *via* het I-register.)

```
ADD   COUNT
STI
```

We "ontdubbelen" de waarde in het A-register, trekken 2Ø af en kijken na of het resultaat reeds nul is. Indien dit zo is, is onze opdracht volbracht:

```
SUB   COUNT
```

```
SUB   N2Ø
JPZ   OUT
```

OUT is nog zo een, voorlopig, niet-gespecificeerd adres. We weten nog niet waar het zich bevindt, omdat we nog niet weten waar het programma eindigt en daarom is het raadzaam het een voorlopige naam te geven.

Indien de sprong niet plaatsvindt dan tellen we 1 op bij COUNT:

```
LD    COUNT
ADD   N1
ST    COUNT
```

en we incrementeren het I-register met 1:

```
XAI
ADD   N1
XAI
```

Nu bevindt de actuele COUNT zich terug in het A-register en kunnen we dus een lus leggen naar de plaats waar het verdubbelen gebeurt:

```
JP    LOOP
```

op voorwaarde dat we de "ADD COUNT"-instructie het symbolisch adres "LOOP" (lus) geven; laten we dit eens doen door de instructie te laten voorafgaan door haar symbolisch adres, gevolgd door een dubbele punt:

```
LOOP: ADD   COUNT
```

Op analoge wijze kunnen we de beginwaarden vastleggen die we nodig hebben. Hiertoe definiëren we een nieuwe opcode, HEX, die een woord set op de vereiste waarde. Eigenlijk is dit helemaal geen opcode omdat het niet equivalent is met een machinecode, daarom noemen we het een pseudo-operatie. Het volledige programma ziet er als volgt uit (let voorlopig niet op de getallen in de linker en rechter kantlijn):

```
Ø20          LD     BASE     1  Ø33
Ø21          XAI             A  ØØØ
Ø22          LD     N1       1  Ø3Ø
Ø23          ST     COUNT    2  Ø32
Ø24 LOOP:    ADD    COUNT    Ø  Ø32
Ø25          STI             2  8ØØ
Ø26          SUB    COUNT    4  Ø32
Ø27          SUB    N2Ø      4  Ø31
```

Ø28	JPZ	OUT	6	Ø47
Ø29	LD	COUNT	1	Ø32
Ø2A	ADD	N1	Ø	Ø3Ø
Ø2B	ST	COUNT	2	Ø32
Ø2C	XAI		A	ØØØ
Ø2D	ADD	N1	Ø	Ø3Ø
Ø2E	XAI		A	ØØØ
Ø2F	JP	LOOP	5	Ø24
Ø3Ø N1:	HEX	ØØØ1	Ø	ØØ1
Ø31 N2Ø:	HEX	ØØ14	Ø	Ø14
Ø32 COUNT:	HEX	ØØØØ	Ø	ØØØ
Ø33 BASE	HEX	ØØØØ	Ø	ØØØ

Het enige symbolische adres dat nog niet voorkomt in de linker kolom en dat daarom nog steeds niet gespecificeerd werd, is OUT. Hier zullen we ons later om bekommeren.

Het programma zoals het er nu staat, is geschreven in wat men noemt *assembly code.*

In moderne geavanceerde computers is een *assembler programma* aanwezig dat als taak heeft de assembly code voor ons om te zetten in echte machinecode.

ZELF ASSEMBLEREN

Helaas beschikt noch onze hypothetische machine noch de Spectrum over een dergelijk programma (hoewel commerciële programma's in de handel verkrijgbaar zijn). We moeten het werk dus zelf doen. We hebben hiervoor een tabel nodig met opcodes en hun equivalente hex waarden.

Opcode	Hex
ADD	Ø
LD	1
ST	2
HLT	3
SUB	4
JP	5

JPZ	6
JPN	7
CALL	8
RET	9
XAI	A

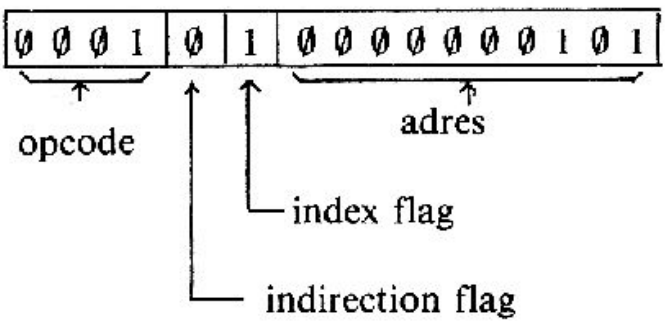
Bovendien moeten we weten waar het programma begint. Hierover beslist de programmeur eigenmachtig, dus nemen we aan dat het programma begint op Ø2Ø. Omdat elke instructie 1 woord in beslag neemt, kunnen we het adres van elke instructie opschrijven. U zult vaststellen dat we dit hebben gedaan in de linker kantlijn van het programma uit vorige paragraaf. Nu kunnen we de opcodes en adressen vervangen door hun hex equivalenten. Zo wordt bijvoorbeeld LD BASE vervangen door 1Ø33 omdat BASE nu wordt geïdentificeerd als Ø33. De volledige code vindt u in de rechter kantlijn van bovengenoemd programma.

De enige instructie die verder commentaar behoeft, is JPZ OUT die werd gecodeerd als 6Ø47. Waarom moet OUT voorkomen bij Ø47? Dit zou ook ergens anders kunnen maar Ø47 is de *eerste* locatie waar dit mogelijk is. De reden is de volgende: de array neemt de ruimte in beslag vanaf Ø33 tot en met Ø46 (twintig woorden), en we wensen niet dat er wordt geploegd in het datagebied van ons programma.

HET INDEXREGISTER

Wanneer men met het X-register werkt, dan wordt het echte instructie-adres gevormd door het adresveld van de instructie op te tellen bij de inhoud van het X-register. Indien, bijvoorbeeld, het X-register 4ØØ bevat dan heeft de instructie LDX Ø05 hetzelfde effect als de instructie LD 4Ø5.

We nemen nog maar eens een bit uit het adresveld om aan te duiden of het indexregister in werking is of niet. De LDX Ø05-instructie ziet er in machinecode als volgt uit:



In hex is dit 14Ø5. Eigenlijk is er niets dat u met het indexeren kunt doen wat u al niet kon met indirectie. Het enige verschil is dat u met indexering automatisch kunt rekenen met adressen en dat u het werk dus zelf niet meer hoeft te doen.

7 Eindelijk de Z8Ø!

Alvorens we de architectuur van de Z8Ø onder de loep nemen, buigen we ons even over enkele moeilijkheden die inherent zijn aan de hypothetische microprocessor.

Ten eerste is er de 4-bit operatiecode die steeds 16 verschillende instructies mogelijk maakt. (OK, we hebben een beetje vals gespeeld door de indirectie- en indexflags het adresveld te laten overlappen, maar anderzijds hebben we hierdoor de grootte van het adres en dus ook de maximum grootte van het geheugen beperkt!) De Z8Ø verstaat 694 instructies! Om elk van deze instructies met een apart bitpatronen te laten overeenstemmen hebben we een 8-bit veld (1 byte) nodig; en zelfs dan is hier en daar wat geknoei onvermijdelijk.

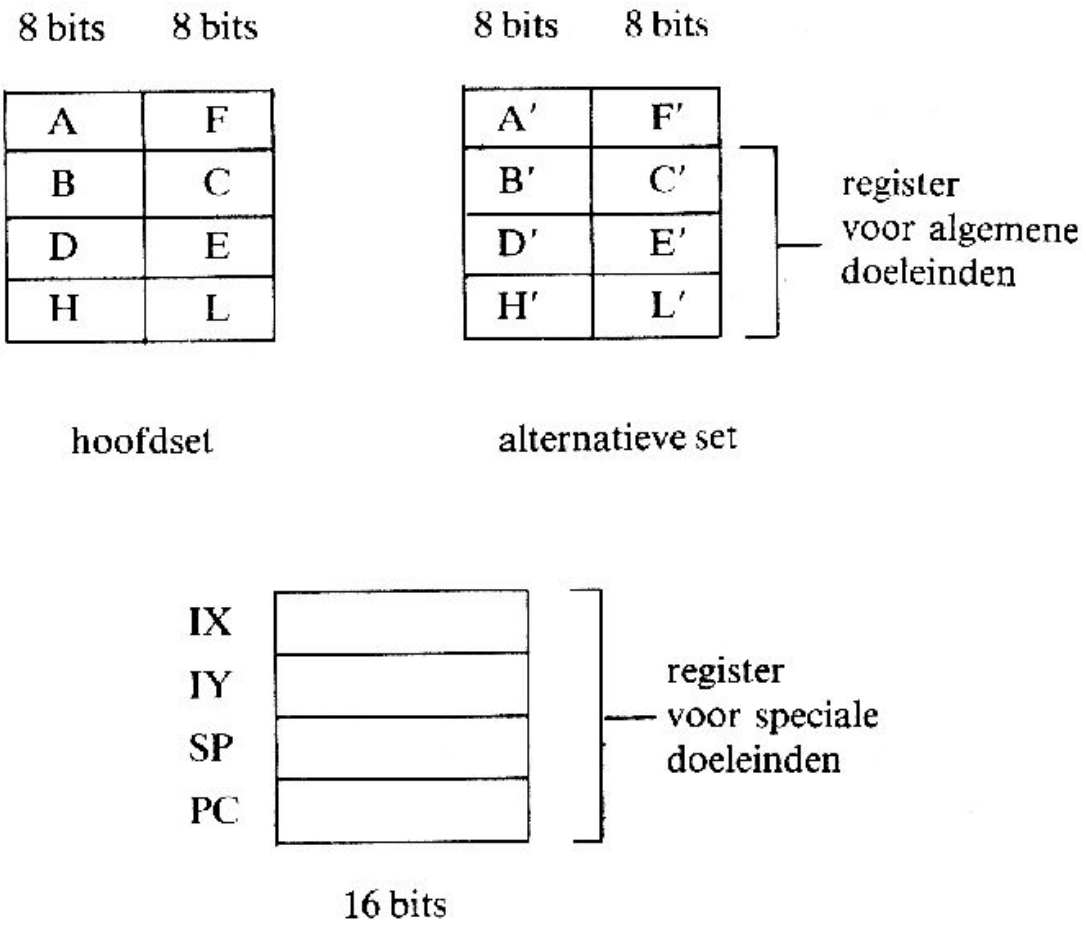
Ten tweede springt onze denkbeeldige machine nogal roekeloos om met het geheugen. Bepaalde instructies maken geen gebruik van het adresveld (HLT, LDI, STI) zodat een opeenvolging van dergelijke instructies in elk woord 1Ø bits verspillen. De Z8Ø lost dit probleem op door niet alle instructies even lang te maken; sommige instructies hebben geen adresveld en hebben een lengte van precies 1 byte, andere hebben een adresveld van 1 byte en zijn dus 2 bytes lang, nog andere hebben een adresveld van 2 bytes en zijn dus in totaal 3 bytes lang, er zijn zelfs instructies die opcodes met een lengte van 2 bytes hebben! Deze eigenschap heeft tot gevolg dat de PC niet voor elke instructie met 1 kan worden vermeerderd. Hij moet dus worden geïncrementeerd met de lengte van de instructie.

Ten derde brengt ons model met zich dat we steeds 16-bit woorden moeten "meesleuren" en dit is niet erg geschikt als we met karakters werken (die normaal ieder één byte in beslag nemen). Het zou dus erg handig zijn indien 2-bit *én* 16-bit operaties mogelijk waren.

Ten vierde is het vervelend dat er slechts één register voor algemene doeleinden (het A-register) is. Dit houdt dikwijls in dat tussenresultaten tijdelijk terug in het geheugen moeten worden opgeslagen tijdens de uitwerking van een andere berekening. De Z8Ø beschikt over een reeks registers voor algemene doeleinden, nochtans hangt hun preciese aantal af van hun gebruik.

DE REGISTERS

Hier ziet u een voorstelling van de wijze waarop de registers georganiseerd zijn.



Trek u voorlopig niets aan van de alternatieve set.

De registers komen in paren voor wat op zich zelf reeds aanduidt dat ze kunnen worden gebruikt als 8-bit of als 16-bit registers. Op deze manier kunnen we bijvoorbeeld naar het B register (8 bits) of naar het C-register (8 bits) of naar het BC-register (16 bits) verwijzen. De B-, C-, D-, E-, H- en L-registers kunnen alle op deze manier worden gebruikt (slechts de paren BC, DE en HL zijn mogelijk) maar de A- en F-registers zijn strikt 8-bit registers die niet kunnen worden gecombineerd. Wat de 16-bit paren betreft zult u wel hebben verwacht dat de linkse byte (B, D, H) de senior byte is.

Er zijn twee index registers, het IX- en het IY-register, een stackpointer (SP) en een programmateller (PC). En de indirectie dan? Hiervoor kan in principe elk van de 16-bit registerparen voor algemene doeleinden (BC, DE of HL) worden gebruikt, maar voor de eenvoud zullen we voor de indirectie steeds het HL-register reserveren.

Er zijn twee instructiesets: één voor 8-bit operaties en de andere voor 16-bit operaties. We beginnen met de 8-bit "load"-instructies.

8 Adresseermodes en de LD-instructies

Om een voorbeeld te hebben van de 8-bit operatiegroep bespreken we de "load" (LD) operatie. Deze lijkt sterk op de LD-instructie van onze denkbeeldige machine, maar de Z8Ø kent twee extra adresseermodes: *register-naar-register* en *onmiddellijk*. Dat brengt met de *directe*, de *indirecte* en de *geïndexeerde* het totaal van adresseermethodes op vijf.

1. *Direct adresseren*
   Dit heeft veel weg van het denkbeeldig adresseerequivalent maar, omdat de Z8Ø meer dan één register heeft, moeten we specificeren welk register we willen laden:

   LD A, (ØF1C)

   Deze operatie laadt de inhoud van geheugenlocatie ØF1C in het A-register. Denk er steeds aan dat de instructie van links naar rechts wordt uitgevoerd. We kunnen dus ook schrijven:

   LD (ØF1C), A

   wat dan betekent "sla de inhoud van het A-register in ØF1C op".
   (Het A-register is het enige *8-bit* register dat rechtstreeks kan worden geadresseerd.)

2. *Indirect adresseren*
   Dit is ook eenvoudig. Omdat we het HL-register reserveren voor de indirectie is het instructieformat:

   LD A, (HL)

   dit betekent "laadt het A-register *via* (d.i. met de inhoud van het adres uit) het HL-register". In de andere richting wordt het:

   LD (HL), A

   en nu wordt de inhoud van A geladen in het adres dat HL *bevat*. (Voor deze instructie zijn, behalve A, ook andere registers toegelaten.)

3. *Geïndexeerd adresseren*
   Bij deze adresseermode moeten we aangeven welk index-register in gebruik is én bovendien moeten we vermelden hoeveel de "offset" bedraagt:

LD A, (IX + 2E)

Merk op dat we bij direct adresseren een adres gaven dat bestond uit 4 hex cijfers, omdat voor het adres 16 bits (2 bytes) voorzien zijn. De offsetwaarde in een geïndexeerde adresseerinstructie moet echter in 1 byte worden opgeslagen, daarom bestaat deze waarde uit slechts twee hex getallen.

4. *Register-naar-register adresseren*
We kunnen gegevens van het ene register doorspelen naar een ander:

LD D, B

Deze instructie betekent: "laad D met de inhoud van B."

5. *Onmiddellijk (immediate) adresseren*
Bij deze adresseermethode worden de gegevens zelf, in plaats van het adres van de data, in het adresveld geplaatst. Dus betekent:

LD B, Ø7

"plaats het getal Ø7 in B". Alweer bestaat het getal uit twee hex cijfers omdat het moet worden opgeslagen in de ene byte van het B-register. Merk ook op dat een "LD"-instructie in werkelijkheid een *copieer*instructie is: de getallen blijven in hun oorspronkelijke adressen of registers behouden maar er wordt een copie van geplaatst in de bestemming.

HEX CODES

We gaan nu even nakijken hoe deze adresseerinstructies er in hex uit zien; de volledige listing vindt u steeds terug in appendix 6.

1. LD A, (ØF1C)

Eerst zoeken we de opcode voor de instructie LD A, (nn) op. (De nn duidt een algemeen 2-byte adres aan.) Deze code is 3A, zodat u verwacht dat deze instructie wordt gecodeerd als:

3A ØF1C

Jammer genoeg levert de Z8Ø, die getallen op een heel andere manier bekijkt, een kleine complicatie op. De Z8Ø moet de minst significante (junior) byte van een adres éérst zien. We moeten dus de *adresbytes van plaats verwisselen:*

3A 1CØF

Dit is wel een beetje vervelend maar u raakt er vlug aan gewend. De vaste regel voor 2-byte getallen in Z8Ø-instructies is dus: *eerst de junior byte, daarna de senior.* Dit is ook de reden waarom u in het Sinclair *Handboek* dikwijls de

instructie PEEK X + 256 * PEEK (X + 1) tegenkomt.
De LD (nn), A instructie heeft code 32, dus wordt:

LD (ØF1C) gecodeerd als 32 1CØF

2. LDA, (HL)

Dit is gemakkelijk omdat er geen adresgedeelte in voorkomt, deze instructie is dus een 1-byte opcode. Indien u deze opzoekt vindt u 7E.
Analoog vindt u voor LD (HL), A de code 77.

3. LD A, (IX + 2E)

De algemene vorm van de instructie is LD A, (IX + d), waarbij d een 1-byte verschuiving voorstelt (in twee complement notatie); de code voor de instructie is DD 7E (een 2-byte opcode!) De instructie wordt:

DD 7E 2E

waarbij de 2E byte de verschuiving is die in dit geval werd gekozen.

4. LD D, B

Geen probleem, de code is 5Ø.

5. LD B, Ø7

De opcode is Ø6, dus wordt de instructie Ø6Ø7.

9 Machinecode opslaan, runnen en saven

Hoofddoelstelling van dit hoofdstuk is het uitwerken van een eenvoudig BASIC gebruiksprogramma (LOADER) dat de ongemakken van machinecode grotendeels elimineert. In appendix 7 wordt hiervoor een meer gesofisticeerde versie gegeven.

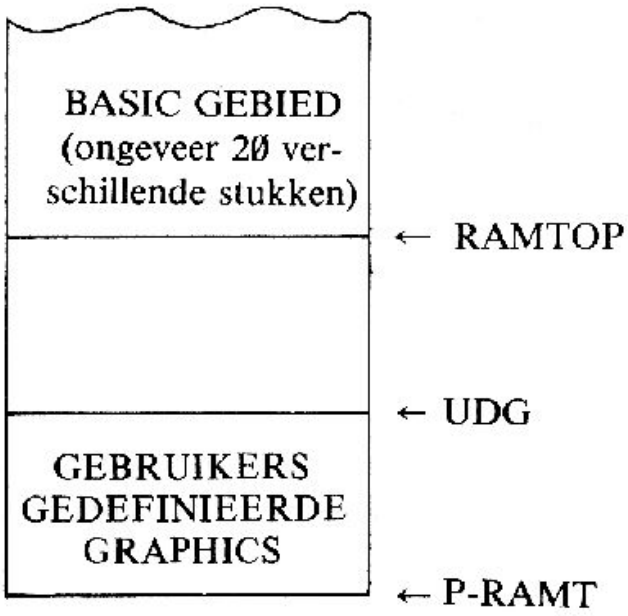
We beginnen met het geheugen van de Spectrum dat tweevoudig is:

Type	Startadres		Eindadres		
	Decimaal	Hex	Decimaal	Hex	
ROM	0	0	16383	3FFF	
RAM	16384	4000	32767	7FFF	(16K machine)
			65535	FFFF	(48K machine)

De ROM bevat het "operating system", de BASIC interpreter, enzovoort: indien we hieraan zouden knoeien zou dat rampzalige gevolgen hebben, daarom is de ROM zo ontworpen dat dit niet mogelijk is; ROM betekent immers "Read Only Memory" (of leesgeheugen). De ROM kan gePEEKt maar niet gePOKEt worden.

In de RAM ("Random Access Memory") kunnen we naar hartelust schrijven en lezen en het is precies in deze RAM dat de machinecode moet worden opgeslagen en uitgevoerd. Eén mogelijk ongemak is dat ook het BASIC-systeem de RAM zal gebruiken; dus kan het BASIC-systeem onze machinecode uitwissen of overschrijven indien we ze op een oude plaats opslaan. We moeten onze machinecode dus bewaren op een plaats die niet door het BASIC-systeem wordt gebruikt (of die er ten minste niet door wordt gewijzigd).

De best geschikte plaats (later zullen we zien dat er nog andere zijn) is het bovenste uiteinde van het geheugen. Hier ziet u een voorstelling van het bedoelde geheugengebied:



Drie systeemvariabelen bevatten de adressen van de grenzen tussen deze gebieden. Deze adressen zijn *dezelfde* voor de 16K- en de 48K-machine:

RAMTOP	23730 – 1
UDG	23675 – 6
P-RAMT	23732 – 3

Aan de hand van PEEK-instructies kan men de waarde van deze 2-byte variabelen vinden. Voor RAMTOP gebeurt dit als volgt:

```
PRINT PEEK 23730 + 256 * PEEK 23731
```

Wanneer de machine pas werd ingeschakeld, krijgen deze variabelen standaardwaarden:

Variabele	16K		48K	
	Decimaal	Hex	Decimaal	Hex
RAMTOP	32599	7F57	65367	FF57
UDG	32600	7F58	65368	FF58
P-RAMT	32767	7FFF	65535	FFFF

Het BASIC-systeem gebruikt het gebied onder het adres dat in de RAMTOP-variabele zit, maar ook dit adres wordt gebruikt. Het stuk vanaf de UDG tot aan de P-RAMT (dit gebied noemt men de top van de fysische RAM) slaat het format op van grafische symbolen die door de gebruiker worden gedefinieerd (user-defined-graphics). Door de waarde van UDG aan te passen kan men dit laatste gebied inkrimpen of zelfs volledig laten verdwijnen; maar om de zaken niet ingewikkelder te maken, veronderstellen we dat het gebied van de user-defined-graphics zich in haar standaardpositie bevindt.

We maken ruimte vrij voor de machinecode door de waarde van RAMTOP te verkleinen. Om bijvoorbeeld 600 bytes vrij te laten en de machineco-

de te beginnen op adres 32000, moeten we RAMTOP veranderen in 32000 − 1 = 31999.

Op deze manier hebben we meer dan genoeg ruimte vrijgemaakt voor alles wat in dit boek voorkomt. Het startadres voor de machinecode is een rond getal en dit zowel in decimale (32000) als in hex notatie (7D00), dus nemen we deze waarde als standaard. Indien u andere waarden wenst hoeft u slechts de getallen 32000 en 31999 − of 7D00 en 7CFF − te vervangen door de door u gekozen waarden. Om te vermijden dat we een aparte uiteenzetting voor de 48K-machine moeten houden, gebruiken we voor die machine *eveneens* de waarde 32000 als startadres, hoewel we hierdoor ongeveer 32K RAM verspillen. Indien u geen RAM verloren wenst te laten gaan kunt u 64000 gebruiken wat in hex wordt genoteerd als FA00. Zolang men de machinecode *leert* is deze verspilling van geheugenruimte onbelangrijk, maar in echte programma's moet u ernaar streven de RAMTOP zo efficiënt mogelijk te kiezen.

Om de RAMTOP kleiner te maken gebruiken we het commando:

```
CLEAR 31999
```

Dit commando brengt de volgende effecten teweeg:

het "cleart" alle variabelen
het cleart de display file net als een CLS-commando
het reset de PLOT-positie op onderaan links: (0, 0)
het RESTOREt de DATA-pointer naar het begin
het ruimt de GO SUB-stack op maar houdt daarbij de adressen bij van de tot dan toe gebruikte subroutines
het reset RAMTOP op 31999 en maakt daarbij de adressen vanaf 32000 "safe" van stoornissen
het plaatst de nieuwe GO SUB-stack net onder deze nieuwe waarde voor RAMTOP.

Het commando:

```
CLEAR 31447
```

doet precies hetzelfde maar reset RAMTOP op 31447 waardoor veel meer plaats wordt gelaten: nu zijn de adressen vanaf 31448 safe.

Het spreekt vanzelf dat men CLEAR het best gebruikt voordat variabelen worden toegekend, voordat iets op het scherm verschijnt en voordat subroutines worden aangeroepen. Onderstaande "loading"-routine reset RAMTOP bij het begin op de juiste waarde, maar wanneer u zelf uw machinecode schrijft is het een goede gewoonte CLEAR te gebruiken *vóór* u iets anders doet.

Let op!
Gebruik CLEAR adres −1 om het gebied vanaf adres "adres" vrij te maken en stel u *niet* tevreden met CLEAR adres. De waarde in een CLEAR-commando is immers het laatste *niet veilige* adres en niet het eerste veilige adres. Dit is een ongemak, maar slechts een kleintje: hou er

MACHINECODE OPSLAAN

We hebben een programma nodig dat hex codes accepteert en dat de corresponderende bytes in volgorde boven de RAMTOP opslaat. Onderstaande routine beantwoordt aan deze vereisten; het zal later in dit hoofdstuk worden aangevuld zodat het veelzijdiger wordt.

```
10  CLEAR 31999
20  PRINT "Basisadres" 32000"
30  PRINT "Aantal databytes"  □";
40  INPUT d: PRINT d
50  LET b = 32000
60  FOR i = 0 TO d
70  POKE b + i, 0
80  NEXT i
90  LET a = b + d
100 DIM h$ (2)
110 PRINT "CODE:"
120 INPUT c$
130 IF c$ = "s" THEN GO TO 200
140 PRINT c$ + "□";
150 LET hs = CODE c$ (1) - 48 - 39 * (c$ (1) > "9")
160 LET hj = CODE c$ (2) - 48 - 39 * (c$ (2) > "9")
170 POKE a, 16 * hs + hj
180 LET a = a + 1
190 GO TO 120
200 POKE a, 201
```

Regel 30 laat ons een aantal adressen voor de machinecode reserveren waarin gegevens kunnen worden bewaard: deze werkwijze blijkt dikwijls erg nuttig te zijn. Regels 60-80 vullen de ongebruikte spaties dan met nullen op; de actuele gegevens kunnen dan achteraf – wanneer ze nodig zijn – worden ingePOKEt Regels 150-160 zetten de hex code om in twee (decimale) getallen tussen 0 en 15 (in plaats van 0-9, A-F) zodat 16 * hs + hj de actuele decimale versie van de hex code geeft. Regel 170 POKEt dit in het juiste adres.

De input "s", die geen hex code is, speelt de rol van delimiter en vertelt de machine wanneer ze moet ophouden met het accepteren van codes. Regel 200 zorgt ervoor dat het laatste commando van het machinecode programma de RET-instructie (return naar BASIC) is, deze heeft de hex code C9 (decimaal

201). In het *Handboek* kunt u vaststellen dat het van belang is de machinecode met deze instructie te beëindigen: op deze manier voert bovenstaande "loader" (zo noemt men een dergelijke routine) de machinecode automatisch in voor het geval u mocht vergeten te returnen. (Een extra RET kan geen kwaad zodat het geen rol speelt indien u tweemaal dit commando invoert.)

MACHINECODE RUNNEN

Om de routine te runnen wordt het USR-commando gebruikt. Om technische redenen is USR een *functie*. Het argument van deze functie is het adres vanwaar de machinecode start en de waarde van de functie is de inhoud van het BC-register van de Z80 wanneer de routine wordt beëindigd. De Spectrum zal het machinecode programma uitvoeren zodra deze functie in het BASIC-programma aan de beurt komt. Dit is zo gemaakt opdat de machinecode gemakkelijk via een BASIC-programma toegankelijk is. Indien de machinecode begint op adres 32000 dan hebben we een commando nodig als:

```
LET y = USR 32000
```

maar de waarde van y aan het eind zal ons weinig interesseren. Elk commando dat USR 32000 bevat en dat een correcte syntaxis heeft, zal ons toegang tot de machinecode routine verschaffen; bijvoorbeeld:

```
RANDOMIZE USR 32000
```

(merk op dat dit heel efficiënt is omdat men voor RANDOMIZE slechts één enkele toets moet indrukken) of

```
RESTORE USR 32000
```

of gelijk welk ander correct commando...

Indien er databytes voorkomen dan moet de 32000 worden vermeerderd om te vermijden dat ze zouden worden gezien als een deel van het programma. Indien we het volgende stukje code voegen bij wat we reeds hadden, kunnen we machinecode runnen. Dit stukje code werd zo samengesteld dat we later nog andere opties kunnen inbouwen.

```
300  INPUT  "Optie?" ;o$
310  IF o$ = "r" THEN GO SUB 400
399  STOP
400  LET y = USR (b + d)
410  RETURN
```

MACHINECODE "SAVEN"

We kunnen een machinecode programma bewaren (SAVE) door *byte storage* toe te passen, zie *Handboek*. Stel dat onze routine 77 bytes lang is, het eind-commando RET is hierin begrepen, dan SAVEn we ze onder de naam (bij-voorbeeld) "m-code" door middel van volgend commando.

```
SAVE "m-code" CODE 32000, 77
```

hierin is 32000 de startbyte en 77 de lengte. Eigenlijk moeten we ons van de lengte niets aantrekken want we hebben de positie gestandaardiseerd, dus kunnen we bij wijze van standaardprocedure de 77 vervangen door 600. (Dit neemt iets meer plaats in op de tape).

Het is even eenvoudig om een aantal bytes die op deze manier opgeslagen werden opnieuw te laden (LOAD); het commando:

```
LOAD "m-code" CODE
```

volstaat. Indien u de naam van de code vergeten bent, kunt u gebruik maken van:

```
LOAD " " CODE
```

We kunnen beide commando's in onze loading utility routine als opties in-bouwen:

```
320  IF o$ = "s" THEN GO SUB 500
330  IF o$ = "l" THEN GO SUB 600
500  INPUT "Te SAVEn routine?" n$
510  IF n$ = " " THEN LET n$ = "m-code"
520  SAVE n$ CODE b, 600
530  RETURN

600  INPUT "Te LOADen routine?" n$
610  LOAD n$ CODE
620  RETURN
```

CHECKEN EN UITPRINTEN

Een laatste toevoeging stelt ons in staat de listing op het scherm te checken of, indien we over een printer beschikken, het machinecode programma uit te printen. Indien er vergissingen in de listing voorkomen kunnen deze worden gecorrigeerd door de betreffende adressen te POKEn. De HELPA-routine uit appendix 7 corrigeert op een meer doeltreffende manier, maar voorlopig is volgend stukje code voldoende.

```
340  IF o$ = "z" THEN GO SUB 700
350  IF o$ = "p" THEN GO SUB 700

700  FOR i = b TO a
710  LET j = PEEK i: LET js = INT (j/16):
     LET jj = j - 16 * js
720  LET h$ (1) = CHR$ (js + 48 + 7 * (js > 9) )
730  LET h$ (2) = CHR$ (jj + 48 + 7 * (jj > 9) )
740  IF o$ = "p" THEN PRINT i; "□ □" + h$ + "□ □"; j
750  IF o$ = "z" THEN LPRINT i; "□ □" + h$ + "□ □"; j
760  NEXT i
770  RETURN
```

Op deze manier krijgen we een decimale én een hex listing van het programma.

Merk op dat de originele input de letters a tot en met f als *kleine* letters uitprint, maar dat bovenstaande code *hoofdletters* gebruikt. Indien dit u hindert, moet u bij wijze van oefening eens uitzoeken hoe u dit kunt vermijden. Het is echter gebruikelijker ervoor te zorgen dat kleine letters en hoofdletters afwisselend kunnen worden gebruikt, in de toekomst zullen we dit dan ook doen.

Indien we één regeltje toevoegen, kunnen we deze optie gebruiken om zowel voor als na de run de listing na te kijken:

```
370 GO TO 300
```

nu krijgen we dus alle opties opnieuw. Om te STOPpen kunt u regel 360 toevoegen:

```
360 IF o$ = "a" THEN STOP
```

De mogelijke opties zijn nu:

a	STOP
l	LOAD
p	PRINT (op het scherm)
r	RUN (Machinecode)
s	SAVE
z	COPY (= PRINT op de printer)

Alvorens u verder leest, moet u nu bovenstaand programma copiëren en op tape SAVEn zodat het klaar is om gebruikt te worden met de machinecode listings die volgen.

10 Rekenkunde

Dankzij het ADD- en SUB-commando kunnen we gaan rekenen. Beide commando's verwijzen naar het A-register en beide kunnen gebruik maken van alle adresseermodes behalve van de directe.

Om te beginnen schrijven we een programma dat de getallen 4 en 7 optelt: we moeten een getal in het A-register laden, en eentje in het B-register, beide optellen en het resultaat ergens opslaan. Indien we ons LOADER-programma met 1 databyte gebruiken dan kunnen we de som in adres 32ØØØ (7DØØ) stoppen. Eerst zetten we elke stap in opcode mnemonics om:

Plaats 4 in het A-register:	LD A, Ø4
Plaats 7 in het B-register:	LD B, Ø7
Tel ze op (waarbij de som in het A-register terecht komt)	ADD A, B
Sla het resultaat op:	LD (7DØØ), A

Nu zoeken we de opcodes in het appendix op, en we vinden de hex codering:

LD A, Ø4	3EØ4
LD B, Ø7	ØØØ7
ADD A, B	8Ø
LD (7DØØ), A	32ØØ 7D

Weet u nog hoe het komt dat we van 7DØØ komen tot ØØ7D?

Nu gaan we deze machinecode inladen. Eerst moet u het LOADER-programma in de Spectrum inbrengen en RUNnen. Wanneer u naar databytes wordt gevraagd, voert u 1 in. Daarna voert u de hex codes achtereenvolgens als volgt in:

```
3e  Ø4  Ø6  Ø7  8Ø  32  ØØ  7d
```

(dit omdat de LOADER in kleine letters uitprint) en u eindigt met

```
s
```

als delimiter (en vergeet niet het belangrijke RET-commando toe te voegen!)

Als optie kiest u "r" om te runnen.

Mooi, maar waar zit nu het resultaat? De gemakkelijkste manier om dit zichtbaar te maken is optie "p" kiezen (een listing). Het resultaat hiervan is:

32000	0B	11
32001	3E	62
32002	04	4
32003	06	6
32004	07	7
32005	80	128
32006	32	50
32007	00	0
32008	7D	125
32009	C9	201

Vanaf 32001 zien we het programma en het RET-statement, C9, aan het eind zitten. We zien ook het resultaat van de som, 11, zitten op adres 32000, dit is de plaats die we ervoor hadden gereserveerd.

Alvorens u verder gaat, kunt u machinecode programma's schrijven die de volgende sommen uitwerken:

18 + 66
13 + 17
23 + 6

hiertoe copiëert u bovenstaand programma en vervangt u de 04 en de 07 door het nieuwe getallenpaar; kijk na of de som telkens in adres 32000 terecht komt.

DATA BETER GEBRUIKEN

Dat ging prima, maar het is niet erg comfortabel dat we voor elk getallenpaar een nieuw programma moeten schrijven.

We kunnen de 04 en de 07 vervangen door algemene getallen m en n door gebruik te maken van:

```
POKE 32002, m: POKE 32004, n
```

maar als we dit doen, halen we programma en data door elkaar omdat de twee getallen werkelijk gegevens moeten zijn. In dit geval is het eigenlijk van weinig belang, maar in een ingewikkelder programma is het wel nadelig indien het programma zelf tijdens een run wordt gewijzigd (dit noemt men een *self-modifying program*). Dit is niet wenselijk om twee redenen: ten eerste omdat we het programma niet opnieuw kunnen gebruiken en ten tweede om-

dat we nergens het originele programma kunnen terugvinden.

Een betere werkwijze stelt de twee op te tellen getallen voor als data, laadt ze door middel van een stukje programma in de registers, telt ze bij elkaar op en laadt het resultaat opnieuw in de data.

Hiervoor reserveren we drie databytes:

32000	Eerste getal
32001	Tweede getal
32002	Totaal

die dus in hex adressen 7D00, 7D01 en 7D02 komen. Het programma moet er vanaf 7D03 als volgt uitzien:

Laad eerste getal in A:	LD A, (7D00)
Laad tweede getal in B:	LD B, (7D01)
Tel deze op:	ADD A, B
Plaats de som in 32002:	LD (7D02), A

Dat is mooi, maar wanneer u de hex codes opzoekt stelt u vast dat er geen commando LD B (nn) bestaat. Pech!

Men kan dit probleem op verschillende manieren omzeilen. Eén ervan is het gebruik van indirectie via het HL-register, zodat we in plaats van LD B (7D01) schrijven:

Laad HL met het adres:	D HL, 7D01
Laad B via HL:	LD B, (HL)

Nu krijgt het programma volgende vorm:

LD A, (7D00)	3A 00 7D
LD HL, 7D01	21 01 7D
LD B, (HL)	46
ADD A, B	80
LD (7D02), A	32 02 7D

Laad dit in en reserveer drie databytes; daarna STOPt u het laadprogramma en brengt u de data in het geheugen door:

```
POKE 32000, 44: POKE 32001, 33
```

te gebruiken om (in dit geval) 44 en 33 bij elkaar op te tellen.

Nu toetst U een GO TO 300 in om terug in de LOADER te komen, en RUN optie "r". Dan voert U "p" in voor een listing. De eerste drie geheugenlocaties worden:

32000	2 C	44
32001	21	33
32002	4 D	77

We vinden, zoals we verwachtten, de som 77 in adres 32002.

Wijzig de te POKEn gegevens en kijk na welke resultaten u aldus krijgt. Probeer in het bijzonder:

POKE 32000, 240: POKE 32001, 100

U zult vaststellen dat de computer denkt dat 240 + 100 = 84! Als u dit eigenaardig vindt is dat uw eigen fout: u moet eraan denken dat het ADD-commando de overdrachtsbit verwerpt! Bekijk deze som even binair:

```
240        1 1 1 1 0 0 0 0
100    +   0 1 1 0 0 1 0 0
          ---------------
          0 1 0 1 0 1 0 0    = 84
          ---------------
          1 1
          |
          1
```

De som brengt een negende bit voort die niet in 8-bit byte kan worden vastgehouden, dus valt hij er aan het eind af en is het uitgevoerde resultaat de waarde van de negende bit – dit is 256 – te klein. (Inderdaad 256 + 84 = 340, het "correcte" antwoord.) Er werd geen proef gemaakt en er werd ook geen foutmelding uitgeprint: wanneer u machinecode schrijft bent u volledig op uzelf aangewezen en wat u zelf niet uittest, daarover komt u ook nooit iets anders te weten.

In werkelijkheid bestaan er methodes om deze overdrachtscijfers en deze vorm van "overflow" aan te pakken; kijkt u maar onder de paragraaf "flags" waartoe we later wel komen.

DE AFTREKKING

Probeer nu het programma zodanig te wijzigen dat het het tweede getal van het eerste *aftrekt*. Alles wat u moet doen is:

ADD A, B 80

veranderen in

SUB A, B 90

en verder gaat alles in zijn werk als voorheen. Experimenteer en overtuig u ervan dat 44 – 33 = 11, en kijk na welke effecten overflows (zoals 17 – 99) teweeg brengen.

11 Een uittreksel uit de Z8Ø instructieset

We zijn niet van plan alle 694 opcodes van de Z8Ø te beschrijven, dit zou al te vervelend en bovendien nutteloos zijn. (Raadpleeg hiervoor appendix 4.) We zullen in plaats daarvan een deelverzameling van 3Ø-types van instructies onder de loep nemen (die ongeveer 23Ø commando's omvatten). Jammer genoeg kunnen ze niet alle gebruik maken van alle adresseermodes, daarom vindt u hieronder een handige tabel waarin u vlug kunt opzoeken welke instructies op welke manier kunnen worden geadresseerd; de opcodes worden gegeven in appendix 6.

opcode / Adres-mode	LD	ADD ADC SUB SBC AND OR XOR CP	INC DEC SLA SRA SRL	JR JRC JRNC JRZ JRNZ DJNZ	JP	JPZ JPNZ JPC JPNC JPP JPM	LD	ADD ADC SBC	INC DEC PUSH POP
Register	LD r, s	ADD A, r	INC r					ADD HL, r	INC r
Onmiddellijk	LD r, n	ADD A, n			JP nn	JPZ nn	LD r, nn		
Direct	LD A, (nn) LD (nn), A						LD HL, (nn) LD (nn), HL		
Indirect	LD A, (HL) LD (HL), A	ADD A, (HL)	INC (HL)		JP (HL)				
Geïndexeerd	LD A, (IY + d) LD (IY + d), A	ADD A, (IY + d)	INC (IY + d)	JR d					

8-bit operaties

16-bit operaties

We geven een kleine toelichting bij de tabel. Een aantal opcodes zullen ons vreemd voorkomen, later meer daarover. Voor de andere zijn de conventies als volgt:

1. In deze tabel wordt telkens een voorbeeld van het format van het instructietype getoond. Alle andere opcodes die in eenzelfde kolom voorkomen kunnen worden gesubstitueerd.
2. "r" of "s" duidt een willekeurig register aan. Of het gaat om een 8-bit

of om een 16-bit register hangt af van in welk deel van de tabel de instructie te vinden is. Een voorbeeld: bij de LD r, s-instructie zijn r en s willekeurige 8-bit registers (dus A, B, C, D, E, H of L), maar bij de ADD HL, r-instructie is "r" één van de registers BC, DE, HL of SP.

3. "n" is een willekeurig 8-bit getal. "nn" is een willekeurig 16-bit getal.
4. Indien een register expliciet wordt genoemd, zoals dat met LD A, (nn) het geval is, dan is dit het enige register dat voor dit doel kan worden gebruikt.

   Dit is een verregaande vereenvoudiging want soms kunnen ook andere registers worden gebruikt, maar waar het om gaat is dat de afgebeelde instructieset *altijd* OK is. (U kunt uw instructievocabularium steeds uitbreiden wanneer u deze materie onder de knie hebt.)
5. "d" is een willekeurig 8-bit getal dat steeds bij één of andere 16-bit waarde wordt opgeteld. Met andere woorden, "d" is een "index verplaatser".

We behandelen nu de nieuwe opcodes:

AND

Deze operatie neemt de inhoud van het A-register en die van een ander 8-bit veld en vergelijkt ze bit per bit. Slechts indien beide overeenkomstige bits "1" zijn plaatst deze operatie een "1" terug op deze positie in het A-register, in alle andere gevallen wordt het een "Ø".

Bijvoorbeeld, AND A, Ø7 heeft het volgende effect:

A-register voor de operatie:	Ø Ø 1 1 Ø 1 Ø 1
Ø7:	Ø Ø Ø Ø Ø 1 1 1
A-register na de AND:	Ø Ø Ø Ø Ø 1 Ø 1

Ziet u dat in dit geval de drie junior bits worden overgebracht? Men kan dus een AND gebruiken om een stuk van een byte te selecteren.

OR

Deze operatie lijkt op de AND, maar nu is de resulterende bit een "1" indien één van de bits een "1" is. Dus, OR A, Ø5 geeft:

A-register vooraf:	Ø 1 Ø Ø 1 Ø 1 1
Ø5:	Ø Ø Ø Ø Ø 1 Ø 1
A-register nadien:	Ø 1 Ø Ø 1 1 1 1

In dit geval ziet u dat bepaalde bits tot "1" werden gedwongen ongeacht hun originele waarde.

XOR

Bij deze operatie moeten de initiële bitwaarden verschillend zijn opdat het resultaat een "1" bevat. XOR A, B3 geeft:

A-register vooraf:	Ø 1 Ø 1 1 Ø 1 Ø
B3	1 Ø 1 1 Ø Ø 1 1
A-register nadien:	1 1 1 Ø 1 Ø Ø 1

Deze operatie is erg nuttig om de inhoud van een register te doen omslaan van Ø naar 1 en terug naar Ø. Indien het A-register bij het begin Ø bevat dan slaat de waarde in het A-register om telkens als de instructie XOR A, Ø1 wordt uitgevoerd. (Van Ø naar 1, terug naar Ø, terug naar 1 enzovoort.)

CP

Dit is de vergelijkingsinstructie (CP staat voor "compare"). De inhoud van het A-register wordt vergeleken met die van een ander 8-bit veld. Deze instructie schept een probleempje: hoe wordt het resultaat van de vergelijking meegedeeld?

Daarvoor dient het F (of flag) register. Elke bit (flag) van het F-register bevat bepaalde informatie over het effect van de laatste instructie. (Niet elke instructie brengt een wijziging teweeg in het flagregister).

De flags die ons het meest interesseren zijn de "Carry", "Zero", "Overflow" en "Sign" flags. De CP-instructie kan deze flags wijzigen, maar de belangrijkste flag is voor ons de Zero flag en die wordt één indien de twee vergeleken waarden gelijk zijn.

Indien de inhoud van het A-register *kleiner* is, dan die van de ermee vergeleken byte dan wordt de sign flag één; dit is hetzelfde als zeggen dat het "resultaat negatief is". Voor het ogenblik is dit alles wat u over flags weten moet: (Meer informatie hierover vindt u in hoofdstuk 16 en in appendix 5.)

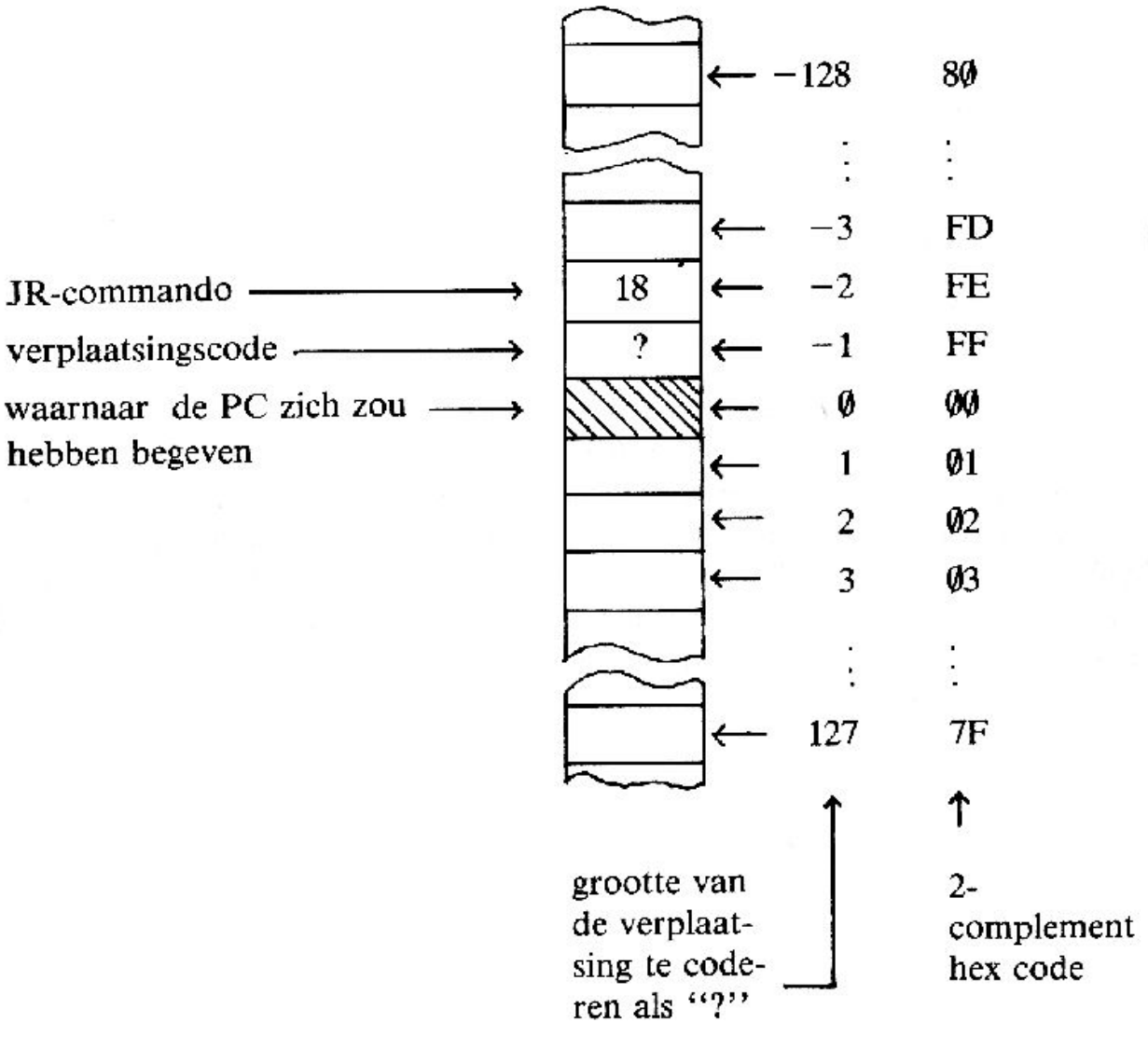
DE SPRONGEN

Of een voorwaardelijke sprong aanleiding geeft tot een vertakking (of niet) is afhankelijk van de waarde van de flags. Zo betekent JPZ bijvoorbeeld "maak een sprong indien de Zero flag één is". Nu wordt het duidelijk hoe we de CP-instructie kunnen gebruiken. Stel bijvoorbeeld dat we willen te weten komen of een bepaalde byte, die door HL wordt aangeduid, de waarde 1E in hex bevat of niet, indien dit zo is wensen we een sprong te maken naar 447B. De code is:

LD A, 1E	3E 1E
CP A, (HL)	BE
JPZ 447B	CA 7B 44

Alle andere sprongen worden op analoge manier gemaakt; JPNZ betekent "maak een sprong indien het resultaat niet nul is" (Zero flag gelijk aan nul), JPP betekent "maak een sprong indien het resultaat positief is" (Sign flag gelijk aan nul), JPM betekent "maak een sprong indien het resultaat negatief is" (Sign flag gelijk aan één), JPNC betekent "maak een sprong indien er geen overdracht heeft plaatsgehad" (Carry flag gelijk aan nul), enzovoort. Al deze sprongen hebben één ding gemeen: het adres van de sprong ligt vast. Met andere woorden, indien we om één of andere reden een routine willen laten draaien die zich ergens anders in het geheugen bevindt dan waar we hem oorspronkelijk hebben geladen, dan moeten alle sprongadressen worden aangepast. Maar ook voor dit probleempje heeft de Z8Ø een oplossing door "relatieve sprongen" (JR) mogelijk te maken. Met deze sprongen kunt u vanaf de plaats waar u zich bevindt een sprong voorwaarts (of achterwaarts) maken over een aantal bytes. Deze verplaatsing wordt vastgehouden (in 2 complement notatie) in 1 byte, dus is de afstand waarover gesprongen kan worden nooit groter dan 128 bytes achterwaarts of 127 bytes voorwaarts.

De verplaatsing wordt berekend aan de hand van het adres waar de PC zich nu naartoe *zou* begeven indien er geen sprong was, en dit adres is dat van het volgend commando in het programma. Dus als volgt:



Hier volgt een voorbeeld. We willen de bytes van het geheugen één na één onderzoeken totdat de eerste maal 1E hex voorkomt. Voor de eenvoud veronderstellen we dat het startadres zich reeds in HL bevindt. We kunnen als volgt te werk gaan:

```
        LD A, 1E
lus:    CP A, (HL)
        INC HL
        JRNZ lus
```

Twee zaken vereisen enige verklaring. Ten eerste hebben we een nieuwe instructie, INC, ingevoerd: deze instructie is een INCrement; ze doet niets anders dan 1 bijtellen bij de inhoud van het gespecificeerd register. Hierdoor kijkt de vergelijkingsoperatie altijd de volgende geheugenbyte na; HL wordt immers na iedere lusdoorgang met 1 vermeerderd. Er bestaat ook een DECrement instructie die precies het tegenovergestelde doet. Ten tweede is er geen duidelijk onderscheid tussen een JRNZ-*lus* en een JPNZ-*lus*. Dit onderscheid wordt pas duidelijk wanneer we de instructies in machinecode *assembleren*. Stel dat de code, zoals gewoonlijk, vanaf 7DØØ hex wordt geladen:

Adres		Instructie	Hex code
7DØØ		LD A, 1E	3E 1E
7DØ2	lus:	CP A, (HL)	BE
7DØ3		INC HL	23
7DØ4		JRNZ lus	2ØFC

Waar komt die FC vandaan in het adresgedeelte van de JRNZ-instructie? De verklaring is als volgt: de PC wordt vermeerderd met 2, nadat de JRNZ-instructie wordt uitgevoerd, omdat dit een 2-byte instructie is. Dus bevindt de PC zich nu op 7DØ6. We willen een sprong maken naar de *lus*, die zich op 7DØ2 bevindt; en dat is 4 bytes terug (of -4 bytes verder om even de redeneringswijze van de ZØ te gebruiken). Nu is 4 in het binair talstelsel ØØØØØ1ØØ en we maken er -4 van door de bits te flippen en bij dit laatste 1 op te tellen (het 2-complement weet u nog?) Dus:

```
Ø Ø Ø Ø Ø 1 Ø Ø
                      flip de bits
1 1 1 1 1 Ø 1 1
            + 1       tel er 1 bij op
-----------------
1 1 1 1 1 1 Ø Ø
                      zet om in hex
    F     C
```

INC HL verandert de flags niet, we kunnen dus zonder bezwaar ná het increment testen.

Hetzelfde programma met absolute sprongen zou er zo uitzien:

Adres		Instructie	Hex code
7DØØ		LD A, 1E	3E 1E
7DØ2	lus:	CP A, (HL)	BE
7DØ3		INC HL	23
7DØ4		JPNZ lus	C2 Ø2 7D

Bemerk dat de JPNZ-instructie 3 bytes heeft omdat ze een volledig 16-bit adres bevat, en vergeet niet de twee bytes van dat adres van plaats te verwisselen!

In de groep van de sprongen bestaat een heel belangrijke instructie die we nog niet hebben genoemd, de DJNZ-instructie. Zij decrementeert het B-register met 1 en maakt slechts een (relatieve) sprong indien het resultaat niet nul is.

Stel dat ons kort programmaatje waarmee we 1E opzochten slechts een gebied van honderd (hex 64) bytes moet afzoeken waarna de lus zou worden verlaten of 1E nu werd gevonden of niet:

```
        LD B, 64        Ø6, 64
        LD A, 1E        3E 1E
lus:    CP A, (HL)      BE
        JPZ hebbes      CA —— (adres voor hebbes)
        INC HL          23
        DJNZ lus        1Ø79
```

De lus wordt honderd maal uitgevoerd, behalve wanneer 1E wordt gevonden

dan grijpt de sprong naar *hebbes* plaats. De DJNZ-lus werkt dus zoals de eenvoudige FOR-lus in BASIC.

Denk eraan dat voor *alle* relatieve sprongcommando's, JR, JRC, JRNC, JRNZ en JRZ, de grootte van de sprong op dezelfde manier wordt berekend. Om het met de hand coderen van sprongen te vergemakkelijken vindt u in appendix 1 een tabel met de 2-complement hex codes; tijdens het schrijven van de code zal onze utility routine, HELPA, in appendix 7 zeker van pas komen want zij werkt de relatieve sprongen automatisch uit.

ADC en SBC

Dit zijn de "ADD with Carry" (Tel op met carry) en "SUB with Carry" (Trek af met carry) instructies. We hebben reeds gesproken over het bestaan van een Carry flag in het flagsregister. Deze flagbit wordt één wanneer een rekenkundige instructie een overdracht veroorzaakt die buiten het register valt. De ADC-instructie werkt net als de ADD maar zal 1 meer meerekenen indien de Carry flag door een vorige operatie werd geset. De SBC-instructie doet ongeveer hetzelfde, maar zij trekt de Carry flag af.

DE SHIFTS

De shiftinstructies, SLA, SRA en SRL verschuiven de bitpatronen.

SLA verschuift het patroon 1 bit naar links, dus indien het B-register het volgende bevat:

Ø	Ø	1	Ø	1	1	Ø	Ø

en SLA wordt uitgevoerd, dan is het resultaat na deze instructie:

Ø	1	Ø	1	1	Ø	Ø	Ø

(Merk op dat een nul wordt gebruikt om het "gat" rechts op te vullen.)

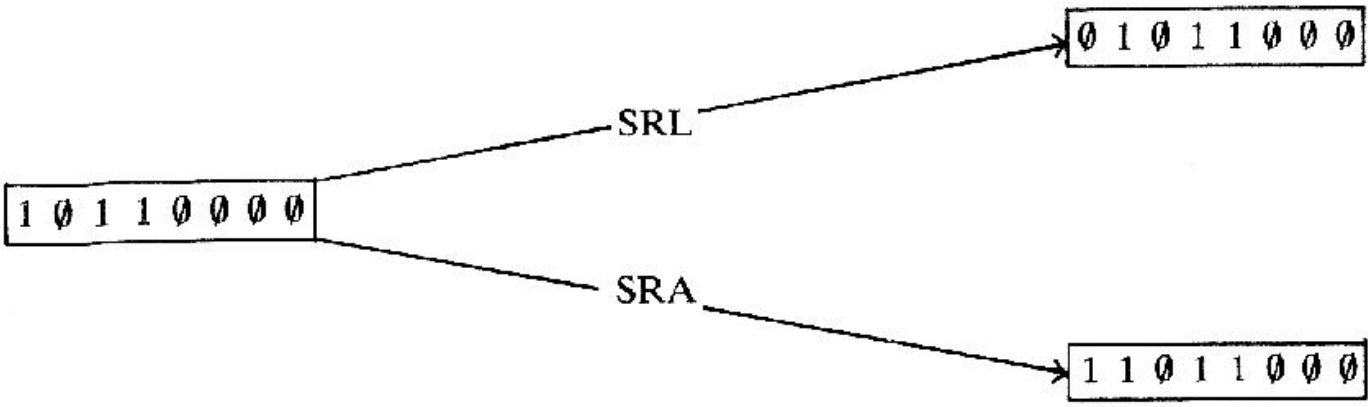
Omdat ØØ1Ø11ØØ = 44 en Ø1Ø11ØØØ = 88 (decimaal) ziet u dat het effect een vermenigvuldiging is met 2.

Nogmaals SLA B geeft:

1	Ø	1	1	Ø	Ø	Ø	Ø

Omdat de senior byte nu 1 is, zal dit getal als negatief worden beschouwd en wordt de Sign flag geset.

Rechtse shifts werken op dezelfde manier, maar hierbij moeten we één belangrijke opmerking maken: SRL vult de senior bit op met een nul, maar SRA vult deze op met de bit die er daarvoor was. Een voorbeeld:



De verklaring hiervoor is de volgende: SRL is een *shift right logical*-instructie die het bitpatroon eenvoudig naar rechts verschuift zonder het te wijzigen. SRA is een *shift right arithmetic*-instructie die werkt als een "deling door 2", en omdat bij de deling van een negatief getal het resultaat negatief moet blijven, moet ook de tekenbit behouden blijven.

PUSH en POP

U zult zich deze benamingen nog wel herinneren van onze uiteenzetting over stacks. Ze worden hier op precies dezelfde manier gebruikt: ze maken het mogelijk de machinestack te manipuleren zonder een subroutine-aanroep te moeten gebruiken.

Dit kan erg handig zijn wanneer waarden tijdelijk moeten worden bewaard. Stel dat we in BC een waarde hebben zitten die we later nodig hebben, maar we willen BC op dit moment graag voor iets anders gebruiken. We kunnen schrijven:

```
PUSH BC
.........
Code waarin
BC
wordt gebruikt
.........
POP BC
```

Dit doet men ook dikwijls vóór een subroutine CALL omdat het op deze manier geen rol speelt welke registers de subroutine gebruikt: er is geen interferentie mogelijk met de aangeroepen programmagegevens. We zien ook wel eens codes als:

```
PUSH BC
PUSH DE      — save de registers
PUSH HL
CALL 4FA1
POP HL
POP DE       — restore de registerwaarden
POP BC         (let op de volgorde!)
```

in dit geval wordt het A-register verondersteld niet te worden gemanipuleerd door de routine, daarom moeten we het niet saven.

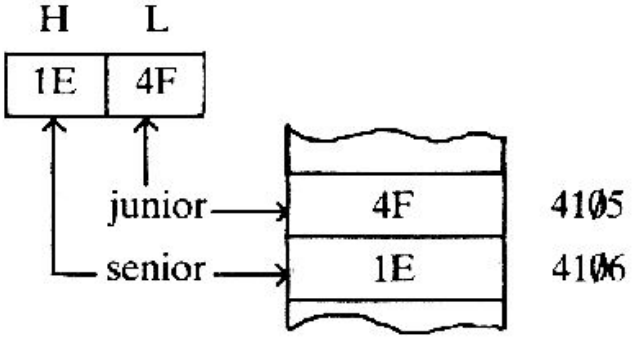
Opgelet
De inhoud van de stackpointer, SP, wordt door het operating system van de Spectrum geïnitialiseerd, tenzij u dit met opzet zou willen veranderen. Het kan geen kwaad de SP op die waarde te behouden maar dan moet u er steeds voor zorgen dat de PUSH- en de POP-instructies in paren tegen elkaar wegvallen zodat de SP bij het verlaten van de machinecode routine terug op zijn initiële waarde staat. CALL's en RET's moeten op dezelfde manier op elkaar inspelen. (USR brengt een CALL voort die past bij de laatste RET-instructie die door de LOADER-routine aan het eind wordt vastgemaakt.)

EEN PROBLEEM VOOR 16-BIT OPERATIES

Voor de 16-bit operaties (PUSH, POP en LD in het bijzonder) is het van belang dat men weet in welke volgorde de bytes van het register naar het geheugen en vice versa worden overgebracht. Dit gaat als volgt:

LD (41Ø5), HL

zal, indien HL de waarde 1E4F bevat, het volgende effect hebben:



We zien dat de minst significante of "junior" byte uit het register wordt geladen in het gespecificeerd adres, en dat de meest significante of "senior" byte wordt geladen in het adres dat erop volgt. Anderzijds zal:

LD HL, (41Ø5)

precies het omgekeerde effect hebben. (NB: volgens de standaardovereenkomst wordt dit gecodeerd als 2A Ø5 41!) Op dezelfde manier wordt:

LD HL, 1ØØØ

(om de hex waarde 1ØØØ in HL te laden) gecodeerd als

21 ØØ 1Ø

waarbij de bytes zoals gewoonlijk worden omgewisseld hoewel 1ØØØ data zijn en geen adres!

CRASHES

Wanneer een BASIC-programma "ontspoort" (crasht), kunt u er steeds op één of andere manier uit geraken met een BREAK zodat het programma niet verloren is. Machinecode crashes werken echter meer op de zenuwen van de gebruiker: ofwel vegen ze alles uit wat in het geheugen staat (zoals NEW) ofwel loopt de machine volledig vast zodat ze moet worden gereset door de stekker enkele seconden uit het stopcontact te trekken en dit heeft natuurlijk hetzelfde resultaat. Indien u zo een crash wilt meemaken dan volgt hier een heel eenvoudige die het geheugen niet uitwist:

RANDOMIZE USR 996

Geluiden, kleuren en geen respons op een aanslag van het toetsenbord. RANDOMIZE USR 14ØØ biedt nog somberder vooruitzichten.

U zult reeds vlug inzien dat crashes niet te vermijden zijn wanneer men in machinecode werkt. De stekker uittrekken is de enige oplossing, daarna moet u maar weer van voren af aan beginnen. Om de kansen op crashes te reduceren kunnen er echter wel enkele voorzorgen worden genomen.

1. Kijk alle machinecode listings nauwgezet na en zorg ervoor dat u ze correct invoert.
2. Gebruik *nooit* de HALT-instructie (opcode 75 hex). Deze is *niet* equivalent aan het BASIC-commando STOP – HALT veroorzaakt zelf een CRASH.
3. Zorg ervoor dat alle CALL's en RET's en alle PUSH's en POP's bij elkaar passen.
4. Roep het *correcte* startadres aan.
5. Alvorens u de routine voor het eerst uitprobeert, kunt u best zoveel mogelijk op tape SAVEn.

12 Een machinecode vermenigvuldiger

Nu gaan we enkele eenvoudige routines schrijven. Misschien herinnert u zich nog dat we hebben gezegd dat er geen Z8Ø vermenigvuldigingsinstructie bestaat. Laten we dus een subroutine voor dit doel schrijven.

EEN VOORBEELD

Eerst moeten we de aard van het probleem onderzoeken. De beste manier om dit te doen is door naar een voorbeeld te kijken. We houden alles zo eenvoudig mogelijk en we werken in 8-bit registers. Indien we 9 met 13 willen vermenigvuldigen, dan ziet dit er als volgt uit:

```
    Ø Ø Ø Ø 1 Ø Ø 1
×   Ø Ø Ø Ø 1 1 Ø 1
```

We kunnen dit nu uitwerken als een traditionele lange vermenigvuldiging, maar omdat het in binair staat, is het eigenlijk eenvoudiger: indien het cijfer waarmee we vermenigvuldigen 1 is dan kunnen we de bovenste regel copiëren, indien het nul is doen we niets:

```
    Ø Ø Ø Ø 1 Ø Ø 1      P
×   Ø Ø Ø Ø 1 1 Ø 1      Q
-----------------
    Ø Ø Ø Ø 1 Ø Ø 1
    Ø Ø 1 Ø Ø 1 Ø Ø
    Ø 1 Ø Ø 1 Ø Ø Ø
-----------------
    Ø 1 1 1 Ø 1 Ø 1
```

Natuurlijk hebben we op elke regel rechts de ontbrekende nullen ingevuld, zoals we dat ook in een lange decimale vermenigvuldiging kunnen doen. In machinecode komt dit neer op een shift naar links. Voor de eenvoud hebben we de twee getallen P en Q genoemd.

Terwijl we P naar links schuiven zal het ook gemakkelijk zijn Q naar rechts te schuiven omdat we op deze manier slechts de junior bit van Q moeten onderzoeken om vast te stellen of we P aan de som moeten toevoegen of niet.

WERKWIJZE

Stel dat P zich in het D- en Q zich in het E-register bevindt. De werkwijze is als volgt:

1. Set het A-register op nul.
2. Indien de junior bit van E een 1 is, tel dan D op in A
3. Schuif D naar links
4. Schuif E naar rechts

] Herhaal deze stappen acht maal

Hier volgt het eerste stukje van de code:

```
LD A, ØØ

LD B, Ø8
```

De eerste stap is evident; de tweede laat B de rol van lusteller spelen in samenwerking met een DJNZ die aan het eind komt. Nu volgt de test van de junior bit van E. De enige manier waarop we dat op dit moment kunnen doen is door gebruik te maken van een bitpatroonmasker (ØØØØØØØ1) en een AND-operatie, we initialiseren C met dat patroon:

```
LD C, Ø1        [de hex codering volgt straks]
```

We kunnen de AND-operatie slechts met het A-register uitvoeren waardoor de actuele inhoud van A verloren zou gaan, dus saven we het A-register eerst in L.

```
lus:    LD L, A
```

daarna halen we de junior bit uit E en restoren we het A-register:

```
LD A, C

AND A, E

LD A, L
```

Indien het resultaat van de AND-operatie nul is, moeten we een sprong maken over het "tel D op in A"-gedeelte van stap 2, dus:

```
JRZ shift
```

(Merk op dat de JRZ nog steeds betrekking heeft op de AND omdat LD geen invloed heeft op de flags.) In het andere geval wordt de ADD uitgevoerd.

```
ADD A, D
```

Nu voeren we de verschuivingen uit:

```
shift:  SLA D

        SRA E
```

en we kijken na of de lus reeds een voldoende aantal keren werd doorlopen:

DJNZ lus

(RET)

DE CODE

Hier volgt heel het programma:

	LD A, ØØ	3E ØØ
	LD B, Ø8	Ø6 Ø8
	LD C, Ø1	ØE Ø1
lus:	LD L, A	6F
	LD A, C	79
	AND A, E	A3
	LD A, L	7D
	JRZ shift	28 Ø1
	ADD A, D	82
shift:	SLA D	CB 22
	SRA E	CB 2B
	DJNZ lus	1Ø F3

Indien u dit programmaatje wilt testen moet u ervoor zorgen dat de D- en E-registers de te vermenigvuldigen getallen bevatten. U kunt het programma dus laten voorafgaan door iets in de trant van:

LD HL, 7DØØ	21 ØØ 7D
LD D, (HL)	56
INC HL	23
LD E, (HL)	5E

en daarna 7DØØ (hex) en 7DØ1 (hex) met de te vermenigvuldigen waarden te POKEn alvorens het programma aan te roepen. Deze twee bytes zullen op het begin van de subroutine staan, waardoor de instructie LD HL, 7DØØ op adres 7DØ2 zal komen. Merk op dat we geen actuele adressen aan het programma hebben toegekend; dit hebben we niet gedaan omdat alle sprongen relatief zijn zodat de actuele adressen geen rol spelen, alleen de verplaatsingen zijn van belang.

U zult ook het antwoord moeten *uitvoeren* want in dit stadium bevindt het zich in het A-register. Dit kunnen we doen door nog een databyte, 7DØ2, voor het antwoord te reserveren zoals we eerder hebben gedaan in het optelprogramma. Hiertoe moet u nog een stukje code aan het programma toevoegen:

```
LD (7DØ2), A        32Ø2 7D
```

en dus 3 databytes gebruiken wanneer u de code inlaadt. Om het antwoord zichtbaar te maken, gebruikt u:

```
PRINT PEEK 32ØØ2
```

Eventueel kunt u als volgt het resultaat uit het A-register naar het C-register overbrengen:

```
LD B, Ø             Ø6 ØØ
LD C, A             4F
```

en dc machinecode aanroepen door middel van:

```
PRINT USR 32ØØ2
```

Denk eraan dat USR een functie is die de waarde van het BC-register bewaart bij een terugkeer naar BASIC; dit commando runt dus de machinecode én print het antwoord uit! (We veronderstelden in dit commando dat we weer twee databytes hebben, daarom zijn we vertrokken vanaf 32ØØ2.)

BIT

Er bestaat een eenvoudigere manier om te testen of de junior bit van E één is of niet. Men heeft hiervoor de instructie, BIT Ø, E. Dus:

```
lus:    LD L, A         6F
        LD A, C         79
        AND A, E        A3
```

wordt vervangen door:

```
lus:    BIT Ø, E        CB 43
```

en de LD A, L moet eveneens verdwijnen.

We vertellen u dit nu pas omdat we beloofd hadden slechts de instructies uit de tabel te gebruiken; die belofte hebben we dus nu gebroken. Maar er is nog een tweede reden: door op deze manier te werk te gaan hebben we bewezen dat men dingen op een bevredigende manier kan doen zonder de volledige instructieset te kennen.

Dit was een nogal akademisch voorbeeld dat we gekozen hebben omdat het gebruik maakt van verschillende gewone instructies op conventionele – maar niet altijd evidente – manier. We zijn er ons echter wel van bewust dat u niet belust bent op een paar dozijn 8-bit vermenigvuldigingen met gehele getallen.

13 De schermdisplay

De informatie die de schermdisplay stuurt, wordt in twee RAM-segmenten bewaard die bekend staan als de *display file* en de *attributes file*. De eerste stuurt de karakters en de hoge resolutie grafieken, de tweede stuurt de kleuren en zaken als FLASH en BRIGHT. Deze files zijn totaal verschillend georganiseerd. De attributes file is de eenvoudigste, daarom beginnen we daarmee.

ATTRIBUTES FILE

Zij beslaat 24 * 32 = 768 bytes in de volgende locaties:

	Decimaal	Hex
Begin v.d. attributes file	22528	5800
Einde v.d. attributes file	23295	5AFF

De eerste 22 * 32 = 704 bytes nemen het gebruikelijke PRINT-gebied van het scherm voor hun rekening, de rijen 0-21 dienen voor PRINT AT-commando's. De laatste 2 * 32 = 64 bytes beslaan het onderste gebied dat uit twee regels bestaat en dat meestal wordt gebruikt voor foutmeldingen en voor het editten. Het PRINT-gebied eindigt op adres 23231 (decimaal) of 5ABF (hex); het onderste schermgedeelte begint op adres 23232 (5AC0).

Indien we de rijen zoals gewoonlijk van 0 tot en met 21 nummeren (waarbij we het onderste gebied kunnen beschouwen als extra regels 22 en 23) en indien we de kolommen een nummer geven van 0 tot en met 31, komt de positie: rij r, kolom c overeen met adres:

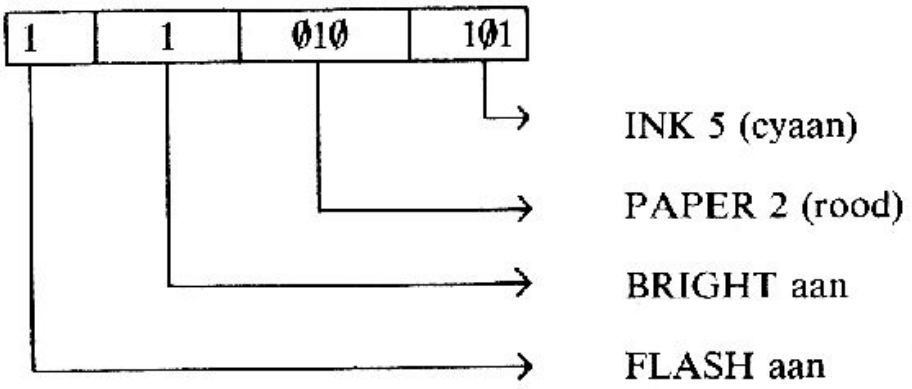
```
22528 + 32 * r + c
```

en dit adres bevat het *attribute* (toekenning) voor die positie.

Elk attribuut is één enkele byte waarvan de acht bits als volgt zijn opgesplitst:

FLASH	BRIGHT	drie	voor	PAPER	drie	voor	INK
aan/uit	aan/uit	bits		kleur	bits		kleur

waarbij Ø staat voor "uit" en 1 voor "aan". Dus betekent byte 11Ø1Ø1Ø1 het volgende:



Indien u dit attribute wilt toepassen op de positie: rij 5, kolom 7, moet u bovenstaande byte opslaan in de locatie met adres:

22528 + 32 * 5 + 7

en dit is:

22695.

Het attribute zelf is 213 (decimaal), dus zal het commando:

POKE 22695, 213

de byte in de juiste locatie opslaan. Natuurlijk zult u de INK-kleur niet zien indien er slechts een blanco spatie wordt uitgeprint, dus moet u nog iets invoeren zoals:

PRINT AT 5, 7; "*"

om er zeker van te zijn dat alles naar wens verloopt.

Omdat er 32 kolommen in elke rij voorkomen, wordt de plaats onmiddellijk onder een gegeven plaats gevonden door 32 (decimaal) of 20 (hex) op te tellen bij het adres van deze laatste plaats. De attributes liggen dus als volgt uitgespreid.

bovenste rij	58Ø0	58Ø1	58Ø2	. . .	581D	581E	581F	⎫
1ste rij	582Ø	5821	5822	. . .	583D	583E	583F	⎪
2de rij	584Ø	5841	5842	. . .	585D	585E	585F	⎪
.	.	.	.	. . .	.	.	.	⎪ PRINT gebied
.	.	.	.	. . .	.	.	.	⎪
.	.	.	.	. . .	.	.	.	⎪
21ste rij	5AAØ	5AA1	5AA2	. . .	5ABD	5ABE	5ABF	⎭
22ste rij	5ACØ	5AC1	5AC2	. . .	5ADD	5ADE	5ADF	⎫ Berichtjes
23ste rij	5AEØ	5AE1	5AE2	. . .	5AFD	5AFE	5AFF	⎭

DISPLAY FILE

Deze is veel ingewikkelder omdat elk karakter van de display wordt opgeslagen als een list van *acht* bytes. Elke byte komt overeen met één rij van de 8 × 8-array van hoge resolutie pixels die het karakter inneemt. Alsof dit nog niet moeilijk genoeg is worden deze bytes *niet* in de voor de hand liggende volgorde opgeslagen.

Wanneer u reeds door middel van LOAD SCREEN$ een tekening in de Spectrum hebt geladen dan zult u wel al hebben opgemerkt dat de tekening in een eigenaardige volgorde wordt "ingekleurd". Dit is nu juist de volgorde van de bytes in de display file. U kunt dit zeer goed zien door het scherm helemaal zwart te maken, te SAVEn en opnieuw te LOADen:

```
10 FOR r = 0 TO 21
20 PRINT "[32 inverse spaties]"
30 NEXT r
40 SAVE "blanco" SCREEN$
```

Zet dit op tape en voer dan het volgende in en ENTER:

```
CLS: LOAD "blanco" SCREEN$
```

en kijk hoe het scherm wordt zwart gemaakt.

Enkele numerieke specificeringen; de adressen zijn:

	Decimaal	Hex
Begin v.d. display file	16384	4000
Einde v.d. display file	22527	57FF

en het totaal aantal bytes is 32 * 24 * 8 = 6144 (decimaal) of 1800 (hex).

We kunnen de display file het best begrijpen door het scherm in drie blokken te verdelen:

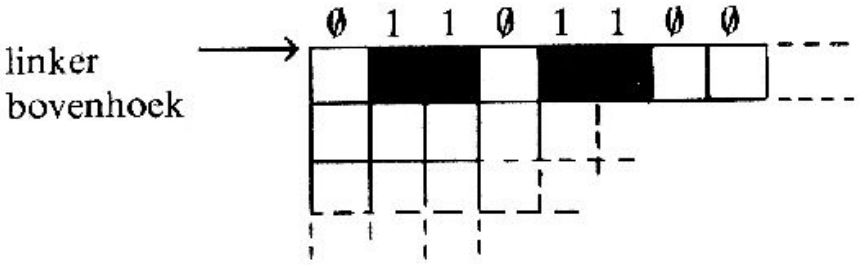
BLOK 1	Rijen 0 tot en met 7 van het scherm
BLOK 2	Rijen 8 tot en met 15 van het scherm
BLOK 3	Rijen 16 tot en met 23 van het scherm

Noteer dat blok 3 de "mededelingen" rijen 22 en 23 bevat. Elk blok vereist 6144/3 = 2048 bytes (800 hex). De eerste 2048 bytes van de display file nemen blok 1 voor hun rekening, de volgende blok 2 en de laatste blok 3. Elk blok wordt op dezelfde manier gerangschikt. In hex samengevat hebben we:

Blok	Beginadres	Eindadres
1	4000	47FF
2	4800	4FFF
3	5000	57FF

We zullen alleen blok 1 behandelen om de organisatie binnen een blok toe te lichten: de andere blokken zijn op dezelfde manier ingericht (maar denk eraan dat blok 3 ook nog een "berichten"gebied bevat). Om alles zo eenvoudig mogelijk te maken werken we in het hoge resolutierooster van 256 bij 176 pixels. Nummer de rijen van Ø tot en met 175 en de kolommen van Ø tot 256; u neemt voor rij 175 de bovenste en voor kolom Ø de linkse, zoals dit gebruikelijk is voor PLOT *kolom, rij*-commando's. Nu wordt blok 1 samengesteld uit rijen 175 tot en met 112, dus een totaal van 8 * 9 = 64 rijen elk met 256 pixels.

De eerste 32 bytes van de display file bevatten de informatie voor rij 175. Elke byte bepaalt een segment van 8 kolommen. De eerste byte neemt de kolommen Ø tot en met 7 voor zijn rekening en wel zo dat een bit overeenkomt met een pixel. Indien bijvoorbeeld adres 4ØØØ de byte Ø11Ø11ØØ bevat dan is de configuratie van de eerste acht pixels bovenaan links op het scherm:



er ontstaat dus een spatie voor "Ø" en een zwart vierkantje voor "1".

De volgende byte zorgt voor de kolommen 8 tot en met 15, de volgende voor de kolommen 16 tot en met 23 enzovoort totdat 32 bytes de volledige rij 175 hebben afgewerkt.

Om dit te zien gebeuren, geeft u een CLS en dan

```
POKE 16384, BIN Ø11Ø11ØØ
```

u zult twee korte streepjes zien in de bovenste rij (de 11-tjes en uit het binair getal). Probeer ook eens met:

```
POKE 16384, BIN 1Ø1Ø1Ø1Ø
```

nu is het resultaat vier puntjes. Verander nu de 16384 in 16385, 16386, . . ., tot 16415; op deze manier vult u rij 175.

De volgende byte bevindt zich in adres 16416; probeer dus eens:

```
POKE 16416, BIN 1Ø1Ø1Ø1Ø
```

en u zult vaststellen dat dit *niet* de eerste acht kolommen van rij 174 zijn, maar wel de eerste acht kolommen van rij 167. Bemerk dat 175 − 8 = 167; m.a.w. er worden *acht* hoge resolutie rijen overgeslagen. Deze byte en de 31 die erop volgen vullen rij 167. Daarna verplaatst de machine zich naar 167 − 8 = 159 en telkens worden 8 rijen overgeslagen totdat de bodem van blok 1 wordt bereikt. In stappen van 32 bytes komen de rijen dus als volgt tevoorschijn:

```
175   167   159   151   143   135   127   119
```

Indien we van rij 119 nog eens acht aftrekken dan hebben we rij 111 en die bevindt zich niet meer in blok 1 (ze is de top van blok 2). Nu springt de Spectrum naar de rijen die zich onmiddellijk onder de vorige bevinden: dus de volgende 8 secties met 32 bytes zijn de rijen:

174 166 158 150 142 134 126 118

Daarna de rijen daaronder:

173 165 157 149 141 133 125 117

enzovoort totdat ten slotte de bodem van het blok wordt bereikt:

168 160 152 144 136 128 120 112

Nu is blok 1 volledig. De volgende 2Ø48 bytes doen hetzelfde met blok 2 en tenslotte is blok 3 aan de beurt. De volgorde is in elk blok dezelfde: van het ene blok tot het andere is er een verschuiving in de adressen van 2Ø48 en een verschuiving in de posities van 64 rijen naar beneden.

Dit ziet er erg ingewikkeld uit; indien u met de display file werkt, kunt u best eerst blok per blok uitwerken, daarna de banden van 8-rijen en tenslotte de individuele rijen. Deze organisatie komt ons al heel wat zinvoller voor als we ons herinneren dat een karakter een vierkantje beslaat van 8 bij 8 pixels (in hoge resolutie) en dit is één PRINT AT-positie. Nu wordt het duidelijk dat de acht rijen van het karakter worden gevormd door de overeenkomstige acht bytes (op juist dezelfde manier als de door-de-gebruiker-gedefinieerde-grafische-karakters, zie *Basic met de ZX-Spectrum,* pagina 50). Samengevat wordt een display van karakters, in blok 1 als volgt georganiseerd in de display file:

32 bytes voor de bovenste rij van elk karakter op regel Ø
32 ,, ,, ,, ,, ,, ,, ,, ,, ,, ,, 1
. . .
32 ,, ,, ,, ,, ,, ,, ,, ,, ,, ,, 7

en daarna:

32 bytes voor de *tweede* rij van elk karakter op regel Ø
32 ,, ,, ,, ,, ,, ,, ,, ,, ,, ,, 1
. . .
32 ,, ,, ,, ,, ,, ,, ,, ,, ,, ,, 7

Daarna volgt de derde, de vierde, . . ., de achtste rij. Het ziet er nog een beetje rommelig uit maar het is toch niet meer zo erg als het op het eerste gezicht leek.

In het volgende hoofdstuk zullen we u beter vertrouwd maken met de attributes file, in het hoofdstuk dat daar op volgt zullen we dan de display file aan pakken.

14 De attributes file

We schrijven een machinecode routine die een vierkantje van een bepaalde kleur in de linker bovenhoek van het scherm uitprint (begin van de file, adres 5800 in hex). Dit is de eenvoudigste manier om de attributes (= toeken) file te leren gebruiken. We zullen één databyte (7D00 zoals gewoonlijk) gebruiken die de kleur (attribute byte) bevat. De code is dan:

LD A, attribute	3A 00 7D
LD (5800), A	32 00 58

Laadt dit in, STOP en set de data door:

POKE 32000, 32

(omdat 32 = 4 * 8 is dit de PAPER 4 attribute.) Nu typt u GO TO 300 en voert u "r" in om te runnen...

...Prachtig, het werkt! Hier volgen enkele oefeningetjes om uw vaardigheid te testen: de antwoorden volgen aan het eind van het hoofdstuk:

1. Maak van het groene vierkantje een rood.
2. Maak er een magenta vierkantje van.
3. Maak het blauw en laat het FLASHen.
4. Maak het geel en BRIGHT.
5. Plaats het op rij 0, kolom 1.
6. Plaats het op rij 3, kolom 7.

Voor oefening 1 tot en met 4 is het voldoende om het adres 32000 met de juiste waarde te POKEn; voor 5 en 6 moet u adres 5800 (hex) aanpassen aan de correcte plaats in de attributes file.

Laten we nu de bovenste rij groen maken. We zullen een lus nodig hebben met een teller die op 32 (dec) wordt geset en die bij elke doorgang wordt gedecrementeerd: we zorgen voor een test die nakijkt of de teller nul is en die een sprong maakt indien dit niet het geval is.

	LD HL, a-file	21 00 58
	LD B, 32 dec	06 20
lus:	LD (HL), 32 dec	36 20
	INC HL	23
	DEC B	05
	CP B	B8

Deze keer zijn er geen databytes dus nemen we onmiddellijk de optie "r".

Het is heel eenvoudig om twee, drie, . . . , rijen te kleuren door de teller B groter te maken. Indien we LD B, 64 dec [Ø64Ø] nemen, worden *twee* rijen groen; [Ø6 6Ø] geeft er drie; enzovoort tot [Ø6 EØ] waardoor zeven rijen groen worden. Door nog eens 2Ø hex (32 dec) aan B toe te voegen beginnen we bij ØØ, de eerste decrementering van B maakt B = 255 (er worden geen overdrachtsbits onthouden!) dus is dit hetzelfde alsof we vertrokken zouden zijn van 256 als waarde voor B. Indien u de tweede regel uit bovenstaande routine vervangt door:

```
        LD B, Ø          Ø6 ØØ
```

kunt u vaststellen dat de bovenste *acht* rijen groen worden ingekleurd. Verder kunnen we niet gaan met het aanpassen van de waarden die in het B-register moeten worden geladen, want dit is slechts een 1-byte register. Alvorens we deze moeilijkheid omzeilen, trachten we onze routine wat efficiënter te maken. We kunnen de teller, B, *automatisch* met het DJNZ-commando laten functioneren:

```
        LD HL, a-file    21 ØØ 58
        LD B, Ø          Ø6 ØØ
lus:    LD (HL), 32 dec  36 2Ø
        INC HL           23
        DJNZ lus         1Ø FB
```

Dit spaart enkele bytes uit en werkt even goed. Probeer maar eens.

OGENBLIKKELIJKE INKLEURING

Om alle 24 rijen op het scherm van kleur te doen veranderen kunnen we ofwel BC als teller gebruiken (en laten starten op Ø3ØØ), een CPC en nog een JRNZ *lus* toevoegen (om na te kijken of BC nul is moeten B en C afzonderlijk worden nagekeken) ofwel bovenstaande routine in een lus leggen door een ander register als teller te gebruiken. Er is nog een derde, eenvoudige oplossing: we kunnen drie copies van het programma aan elkaar vastmaken en ze telkens in een ander deel van de attributes file laten starten.

Deze oplossing zullen we eerst proberen:

```
        LD HL, 58ØØ      21 ØØ 58
        LD B, Ø          Ø6 ØØ
lus 1:  LD (HL), 32 dec  36 2Ø
        INC HL           23
        DJNZ lus 1       1Ø FB
```

	LD HL, 5900	21 00 59
	LD B, 0	06 00
lus 2:	LD (HL), 32 dec	36 20
	INC HL	23
	DJNZ lus 2	10 FB
	LD HL, 5A00	21 00 5A
	LD B, 0	06 00
lus 3:	LD (HL), 32 dec	36 20
	INC HL	23
	DJNZ lus 3	10 FB

Run dit zonder databytes en u zult zien dat het scherm ogenblikkelijk groen wordt gekleurd; zelfs het mededelingengebied. Om dit laatste te vermijden veranderen we de derde "LD B, 0" in "LD B, 192 dec" of [06C0] en nu krijgen we bij de derde lusdoorgang slechts zes groene rijen.

Misschien hebt u opgemerkt dat het niet nodig is om B iedere keer op nul te resetten, omdat dit de waarde is die B reeds heeft! Deze regels kunnen we dus schrappen (behalve de eerste maal die steeds noodzakelijk is; en de laatste maal indien u slechts 22 groene regels wenst).

Het ligt voor de hand dat er een snellere methode is, maar bovenstaande heeft een belangrijk voordeel; ze laat ons toe de kleuren tijdens de run te veranderen. Indien u ervoor zorgt dat de tweede maal dat [36 20] voorkomt dit wordt veranderd in [36 10] en de derde maal in [36 30] (en dit kan het eenvoudigst met POKE 32016, 16: POKE 32026, 48) dan krijgt u drie gekleurde blokken:

GROEN
ROOD
GEEL

daarbij komt nog dat u andere, gelijksoortige, effecten kunt verkrijgen door de attributes bytes te veranderen.

Nu proberen we even de oplossing met de extra lus. We gebruiken D als lusteller en voor de afwisseling maken we het scherm magenta.

	LD HL, a-file	21 00 58
	LD D, 03	16 03
buitenste	LD B, 0	06 00
lus	LD (HL), 24 dec	36 18
	INC HL	23

	DJNZ lus	1Ø FB
	DEC D	15
	CP D	BA
	JRNZ buitenste lus	2Ø F5

Bemerk dat HL in de buitenste lus niet wordt geïncrementeerd omdat HL zich omhoog werkt door de attributes file in de binnenste lus.

DEFLASH EN REFLASH

De volgende routine doorloopt de attributes file en schakelt alle FLASH-es uit, maar laat voor het overige de attributes onveranderd. De FLASH bit is de linkerbit van het attribute, m.a.w. de senior bit. Deze bit moeten we nul maken en hierbij moeten we er voor zorgen dat de rest niet verandert.

We kunnen dit doen door een *bitpatroonmasker* te gebruiken: we stoppen de attribute byte in het A-register en voeren dan een AND-operatie uit met het bitpatroon Ø1111111. Hierdoor wordt de senior bit Ø en blijven de andere bits ongewijzigd. De code wordt in dit geval.

	LD BC, 768 dec	Ø1 ØØ Ø3
	LD HL, a-file	21 ØØ 58
lus :	LD A, (HL)	7E
	AND 127 dec	E6 7F
	LD (HL), A	77
	INC HL	23
	DEC BC	ØB
	LD A, Ø	3E ØØ
	CP B	B8
	JRNZ lus	2Ø F5
	CP C	B9
	JRNZ lus	2Ø F2

Bemerk dat BC dienst doet als lusteller; 768 dec is de lengte van de attributes file. B en C moeten beide op de waarde nul worden getest om de lus te beëindigen.

Laad dit programma zonder databytes en STOP dan. We zullen dit even testen. Voeg dus volgende regels toe:

```
1000  CLS: FOR i = 1 TO 704: PRINT FLASH
      (INT (2 * RND) ); CHR$ (65 + i - 20 * INT (i/20) ); :
      NEXT i
1010  GO TO 300
```

Gebruik GO TO 1000 en het scherm vult zich met letters waarvan er een aantal "flashen". Voer als optie "r" in en u zult merken dat het aan- en uitflitsen ophoudt.

Herhaal alles van voren af aan maar probeer INK en PAPER andere kleuren te geven. U zult vaststellen dat de kleuren niet veranderen.

Het omgekeerde vraagstuk bestaat erin het hele scherm te laten aan- en uitflitsen. In plaats van AND 127 dec als bitmasker te gebruiken, moeten we nu OR 128 dec (128 is 10000000 in het binair talstelsel; OR maakt nu de senior bit 1 en wijzigt de andere bits niet) als masker toepassen. We kunnen dus hetzelfde programma gebruiken, maar we moeten regel:

AND 127 dec	E6 7F

veranderen in:

OR 128 dec	F6 80

Om het programma te testen gebruiken we regel 1000 zoals voorheen. Wat gebeurt er indien u XOR 128 gebruikt?

SET EN RES

We kunnen de bitpatronen uit het voorgaande programma met veel minder omhaal wijzigen. Het SET-commando *set* een gegeven bit in een bepaald register op 1 en RES *reset* hem (op 0).

De bits worden als volgt in een byte genummerd:

7	6	5	4	3	2	1	0

een hoger getal heeft dus betrekking op een meer significante bit ten opzichte van de lagere bits. Het commando:

```
SET 4, D
```

zorgt ervoor dat het nummer 4 in het D-register 1 wordt, en

```
RES 6, E
```

maakt bit nummer 6 in het E-register nul.

In plaats van AND 127 dec te gebruiken, kunnen we dus even goed schrijven:

RES 7, A	CB BF

en in plaats van OR 128 dec:

SET 7, A	CB BF

Deze commando's beslaan evenveel bytes; het is dus niet nodig in het oude programma de relatieve sprongen aan te passen. Pas bovenstaande routines eens aan en kijk of ze nog werken.

EEN KOLOM SCROLLEN

We schrijven een routine die een kolom (voorlopig kolom 31, uiterst rechts) met attributes scrollt. Herinner u dat de attributes van een vierkantje onmiddellijk onder een gegeven vierkantje zich in een adres bevinden dat 32 dec groter is. We moeten dus indexeren met een verplaatsing van 32. Het idee is als volgt: we brengen de attributes van het lagere vierkantje over naar het D-register en daarna dragen we ze over naar het hogere vierkantje; om de hele kolom te hebben doen we dit 21 maal. De code wordt:

	LD BC, 32 dec	Ø1 2Ø ØØ
	LD IX, start	DD 21 1F 58
	LD A, 21 dec	3E 15
lus:	LD D, (IX + 32)	DD 56 2Ø
	LD (IX), D	DD 72 ØØ
	ADD IX, BC	DD Ø9
	DEC A	3D
	CP B	B8
	JNRZ lus	2Ø F4

Bemerk dat de 1F in de tweede instructie het nummer (31) is van de kolom die we wensen te scrollen, dit nummer wordt in hex geschreven; indien we een andere kolom wensen te scrollen moeten we dit nummer aanpassen.

We testen onze routine uit door deze eerst in te laden en te STOPpen en daarna de volgende regels toe te voegen:

```
1ØØØ FOR e = Ø TO 21: PRINT PAPER e - 7 * INT (e/7);
     "[32 spaties]"; : NEXT e
1Ø1Ø GO TO 3ØØ
```

Nu geeft u het commando GO TO 1ØØØ en u verkrijgt een mooi gestreept patroon, waarmee de scrollroutine zal worden uitgetest; u voert nu de optie "r" in om te runnen.

U zult kolom 31 één spatie opwaarts zien scrollen. Voor meer spaties moet u de routine in BASIC in een lus leggen; voor een andere kolom verandert u 1F.

U zult ook vaststellen dat het onderste attribute uit de kolom niet wordt veranderd. Om er een wit vierkantje van te maken (attribute 56 = 7 * 8) moet u na de JRNZ de volgende regel aan de machinecode toevoegen:

LD (IX), 56 dec	DD 36 ØØ 38

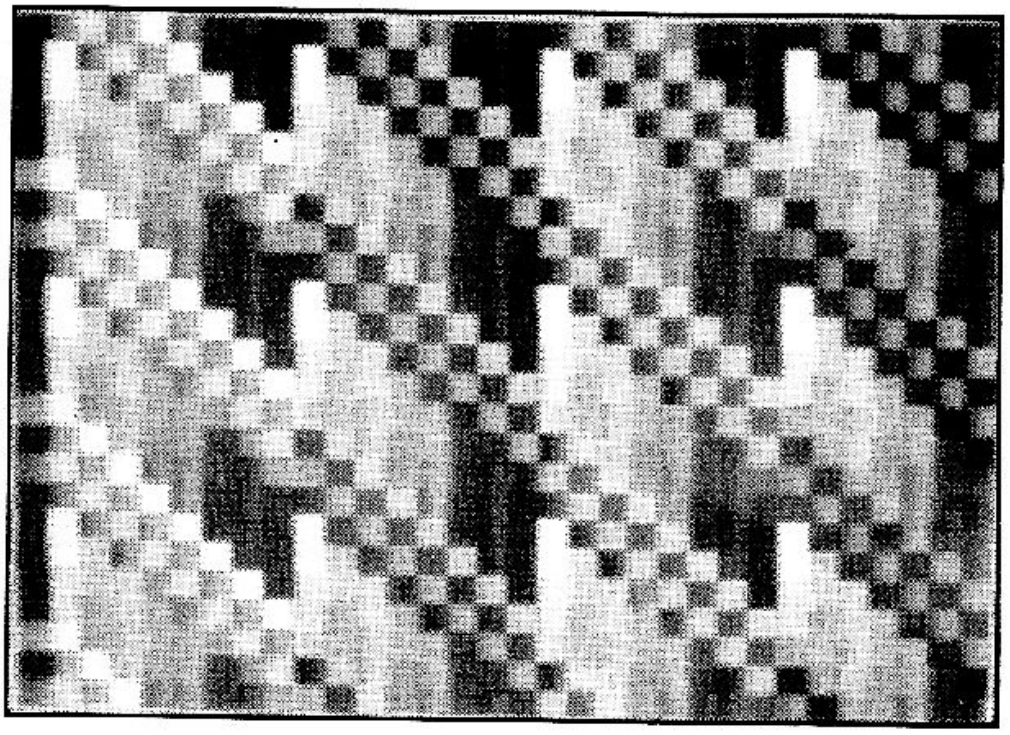
Bij wijze van oefening kunt u proberen een volledig kolommenblok (bijvoorbeeld de kolommen 5 tot en met 17) te scrollen; hiervoor moet u de machinecode routine in een lus leggen en telkens het startadres voor het IX-register incrementeren.

PATROONGENERATOR

Tenslotte volgt hier een routine die de OGENBLIKKELIJKE INKLEURING-routine opvrolijkt door mooie patronen te geven. We laten het aan u over om uit te maken hoe deze routine werkt; er komen geen databytes in voor.

LD B, Ø	Ø6 ØØ
LD D, Ø	16 ØØ
LD HL, 58ØØ	21 ØØ 58
LD (HL), D	72
INC HL	23
LD A, 7	3E Ø7
ADD A, D	82
LD D, A	57
DJNZ ?	1Ø F8
LD HL, 59ØØ	21 ØØ 59
LD (HL), D	72
INC HL	23
LD A, 7	3E Ø7
ADD A, D	82
LD D, A	57
DJNZ ?	1Ø F8
LD HL, 5AØØ	21 ØØ 5A
LD (HL), D	72
INC HL	23
LD A, 7	3E Ø7
ADD A, D	82

LD D, A	57
DJNZ ?	10 F8



Figuur 14.1 Voor dit geruit patroon geldt LD A, 31. Wat zijn de resultaten met andere waarden voor A? Er zijn 255 verschillende mogelijkheden.

U verkrijgt andere patronen indien u de regels LD A, 7 aanpast zodat andere getallen in A worden geladen: bijvoorbeeld LD A, 8, of LD A, 32 of LD A, 31 (decimaal!). In plaats van de 07 kan men willekeurige hexadecimale getallen van twee hex digits gebruiken; bovendien kan men – in de drie gevallen dat LDA voorkomt – andere getallen gebruiken.

Om het scherm te wissen stopt u (d.m.v. optie "a"). Pas daarna geeft u een CLS, dan GO TO 300 en tenslotte gebruikt u optie "r".

Kunt u er achter komen wat de routine doet en hoe ze het doet?

OPLOSSINGEN

1. POKE 32000, 16.
2. POKE 32000, 24.
3. POKE 32000, 136.
4. POKE 32000, 112.
5. Verander 5800 in 5801 [3A 00 7D 32 01 58].
6. Verander 5800 in 5867 [3A 00 7D 32 67 58].

15 De display file

Wat u vooral moet onthouden is dat de display file begint op 4000 hex, dat hij 1800 bytes lang is, en dat hij wordt opgedeeld in drie blokken die respectievelijk gaan van 4000 tot en met 47FF, van 4800 tot en met 4FFF en van 5000 tot en met 57FF.

Om vertrouwd te raken met de display file kunt u het best experimenteren door veranderingen aan te brengen in het display file gedeelte van het geheugen en te kijken wat het resultaat ervan is.

Steek bijvoorbeeld identieke waarden in de adressen; wat gebeurt er?

Om dit te weten te komen nemen we de byte 10101110 (binair) of AE (hex), of 174 (dec), gebruiken we HL om het adres in de display file aan te wijzen en maken we van BC een lusteller. We krijgen dan het volgende.

```
        LD HL, d-file       21 00 40
        LD BC, 1800         01 00 18
lus:    LD (HL), 174 dec    36 AE
        INC HL              23
        DEC BC              0B
        CP B                B8
        JRNZ lus            20 F9
        CP C                B9
        JRNZ lus            20 F6
```

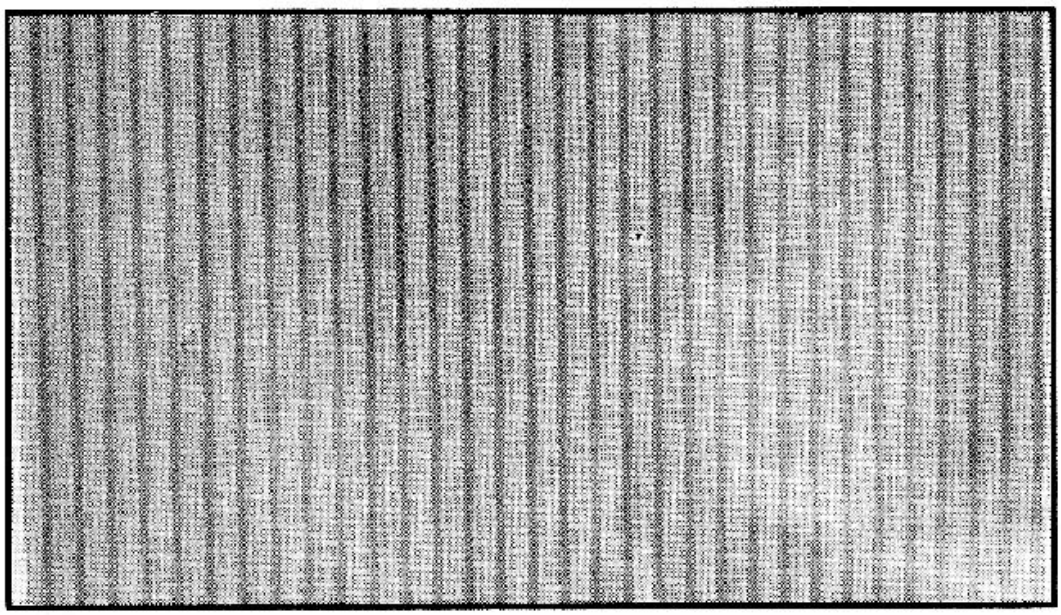


Fig. 15.1. Identieke bytes geven vertikale strepen.

Dit levert ons een patroon met vertikale strepen op die worden gevormd door het bitpatroon van het getal 174, d.i. 1Ø1Ø111Ø in binaire notatie: er komt een zwarte streep met breedte 1, dan een witte met breedte 1, dan weer een zwarte met breedte 1 en nog een witte met breedte 1, vervolgens een zwarte met breedte 3 en tenslotte een witte met breedte 1 in voor. Deze configuratie herhaalt zich 32 maal van links naar rechts over het scherm: dit komt doordat elke rij van de display met hetzelfde bitpatroon wordt geladen en dit herhaalt zich 32 keer; al deze patronen sluiten vertikaal bij elkaar aan en leveren strepen op.

Verander de 174 in andere getallen en kijk na welke patronen tevoorschijn komen. Probeer in het bijzonder de waarden 1, 15 en 17Ø die in hex als Ø1, ØF en AA worden geschreven.

HORIZONTALE LIJNEN

De volgende routine tekent horizontale lijnen (door 255 in de bytes van bepaalde rijen van de display te stoppen). U kunt alvorens u optie "r" gebruikt het scherm wissen zoals dat aan het eind van hoofdstuk 14 werd gedaan.

	LD A, 3	3E Ø3
	LD B, Ø	Ø6 ØØ
	LD HL, d-file	21 ØØ 4Ø
lus:	LD (HL), 255 dec	36 FF
	INC HL	23
	DJNZ lus	1Ø FB
	DEC A	3D
	CP B	B8
	JRZ sla over	28 Ø8
	PUSH AF	F5
	LD A, 7	3E Ø7
	ADD A, H	84
	LD H, A	67
	POP AF	F1
	JR lus	18 EF
sla over	(RET instruction	C9)

PATRONEN

Men kan verbazingwekkende configuraties verkrijgen door verschillende bytes in delen van de display file te laden. Volgend voorbeeld gebruikt het C-re-

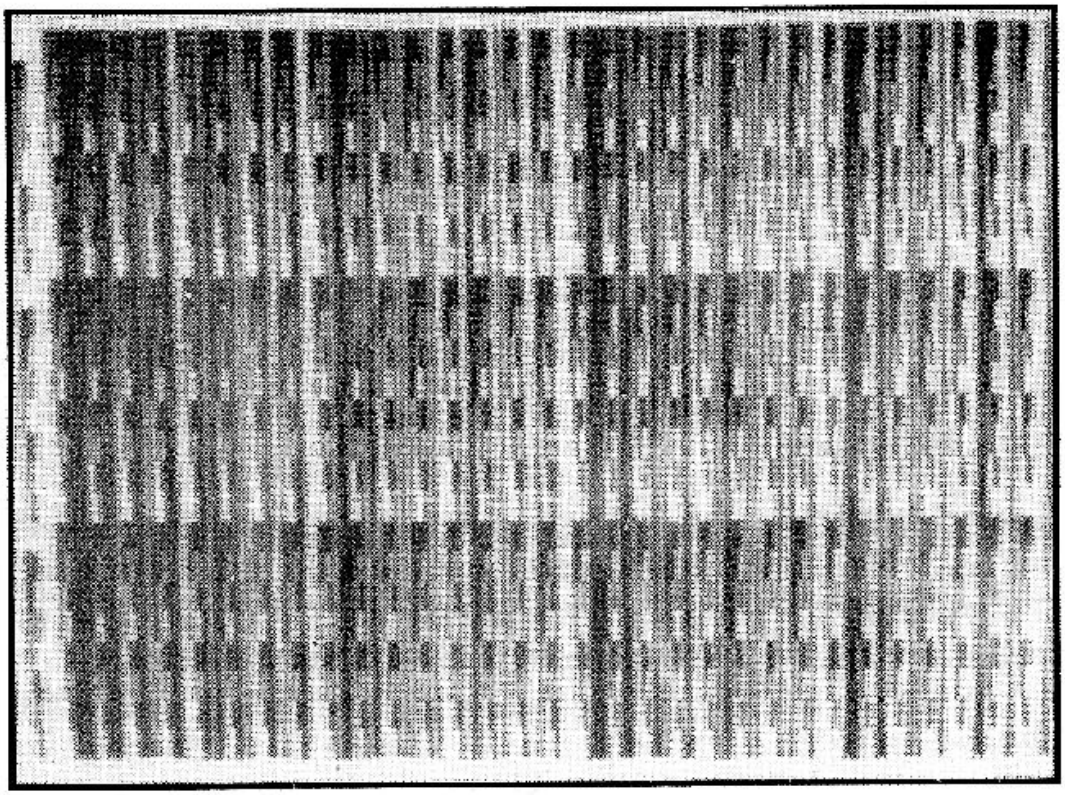
gister dat begint bij ØØ en gedecrementeerd wordt, dus: FF, FE, FD, . . ., enzovoort tot ØØ (in het totaal gebeurt dit driemaal); op deze manier wordt de te laden byte bepaald.

```
        LD BC, 768 dec      Ø1 ØØ 18
        LD HL, d-file       21 ØØ 4Ø
lus:    LD (HL), C          71
        INC HL              23
        DEC BC              ØB
        CP B                B8
        JRNZ lus            2Ø FA
        CP C                B9
        JRNZ lus            2Ø F7
```

Bij wijze van variatie kunt u de *lus* regel veranderen door nu het B-register te laden:

```
lus:    LD (HL), B          7Ø
```

Nu ziet u een ander patroon. Deze routine doorloopt de lus 256 keer en verandert daardoor telkens de waarde van B (die de senior byte is van de lusteller) wat overeenkomt met acht hoge resolutie rijen van het scherm. Uit het patroon blijkt dat de bovenste acht rijen verschillende waarden hebben voor



Figuur 15.2 Variabele bytes geven eigenaardige patronen. Merk de drie blokken op. Kunt u de binaire getallen ontdekken die verantwoordelijk zijn voor deze configuratie?

B. Dit betekent dat de eerste 256 bytes *niet* overeenkomen met de rijen 175 tot en met 168 (zoals dit wel het geval zou zijn indien de rijen in de normale volgorde van boven naar beneden zouden worden genummerd). De manier waarop het patroon zich binnen een 8-rijen blok herhaalt, toont aan dat de eerste 256 bytes in werkelijkheid de lijnen 175, 167, 159 bepalen. Zoals we eerder reeds hebben verklaard. Uit dit patroon blijkt ook zeer duidelijk de drievoudige blokstructuur.

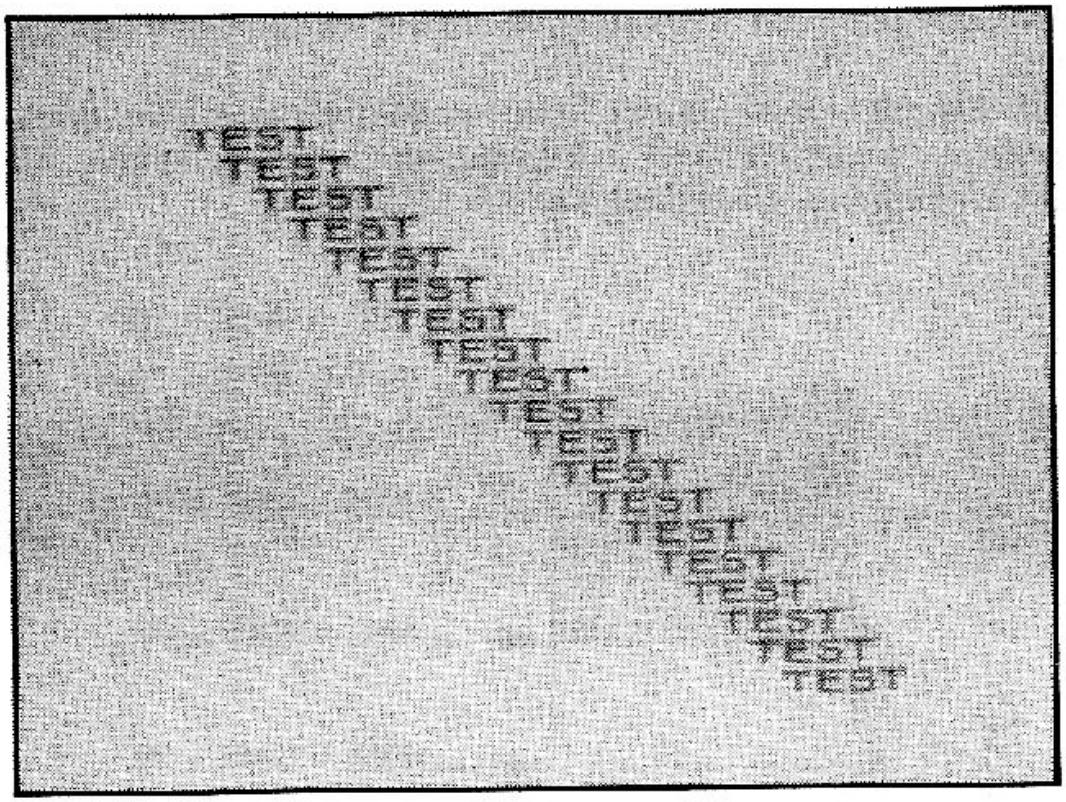
Dergelijke routines geven ons vertrouwen dat we met de display file, interessante dingen *kunnen* doen; maar voor de rest zijn deze routines niet erg praktisch *bruikbaar*. We stappen nu over naar een onderwerp dat wel nuttig is.

ARCEREN

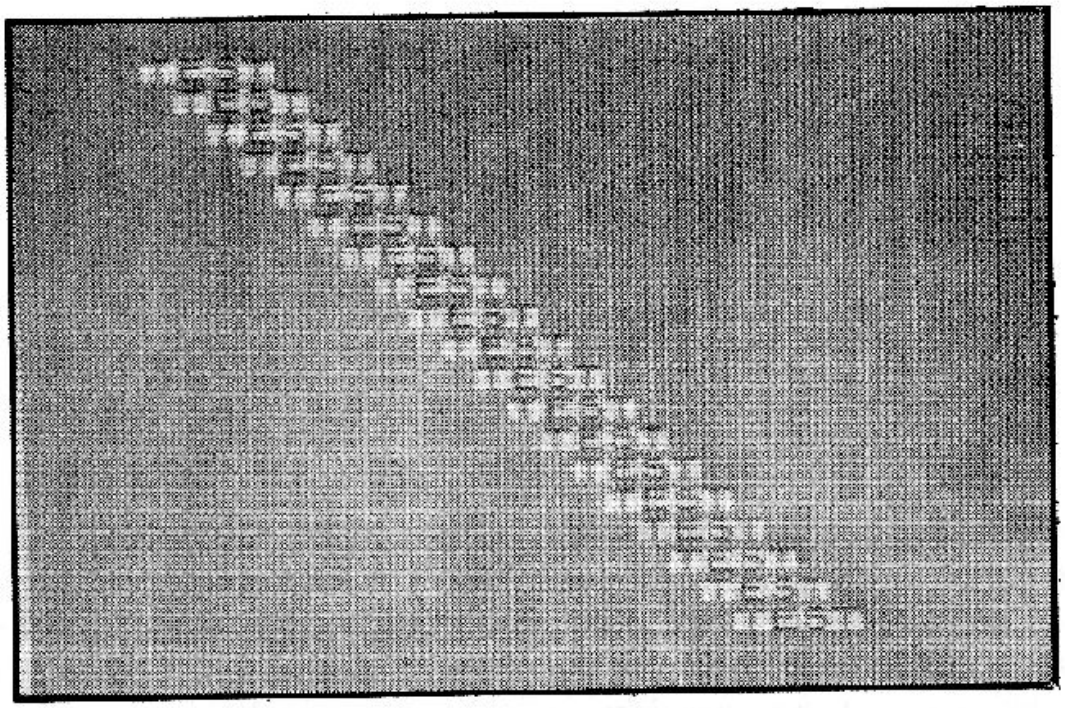
Deze routine heeft als voordeel dat ze de gedetailleerde structuur van de display file niet gebruikt. Ze doorloopt de file en vervangt elke zerobyte, 00000000, door de byte 10101010 (AA hex). Het resultaat hiervan is dat de blanco gebieden van scherm met fijne vertikale streepjes worden gearceerd.

```
        LD HL, d-file          21 00 40
        LD BC, 768 dec         01 00 18
        LD A, 0                3E 00
lus:    CP A, (HL)             BE
        JRNZ sla over          20 02
        LD (HL), AA hex        36 AA
sla over: INC HL               23
        DEC BC                 0B
        CP B                   B8
        JRNZ lus               20 F6
        CP C                   B9
        JRNZ lus               20 F3
```

Het commando LD A, 0 is eigenlijk overbodig omdat de accumulator A reeds nul is, tenzij A met iets anders werd geladen; maar op deze manier wordt het wel duidelijker hoe het programma werkt. U kunt nakijken of het inderdaad overbodig was door het te schrappen. (De snelste manier om dit te doen is d.m.v. POKE 32006, 0; deze instructie verandert de regel in 00 00. De code 00 betekent NOP – No OPeration – en doet niets anders dan tijd verspillen. Deze operatie is ideaal voor het experimenteel verwijderen van een commando omdat de andere hex codes niet in het geheugen verplaatst hoeven te worden.)



Figuur 15.3 Vóór de arcering...



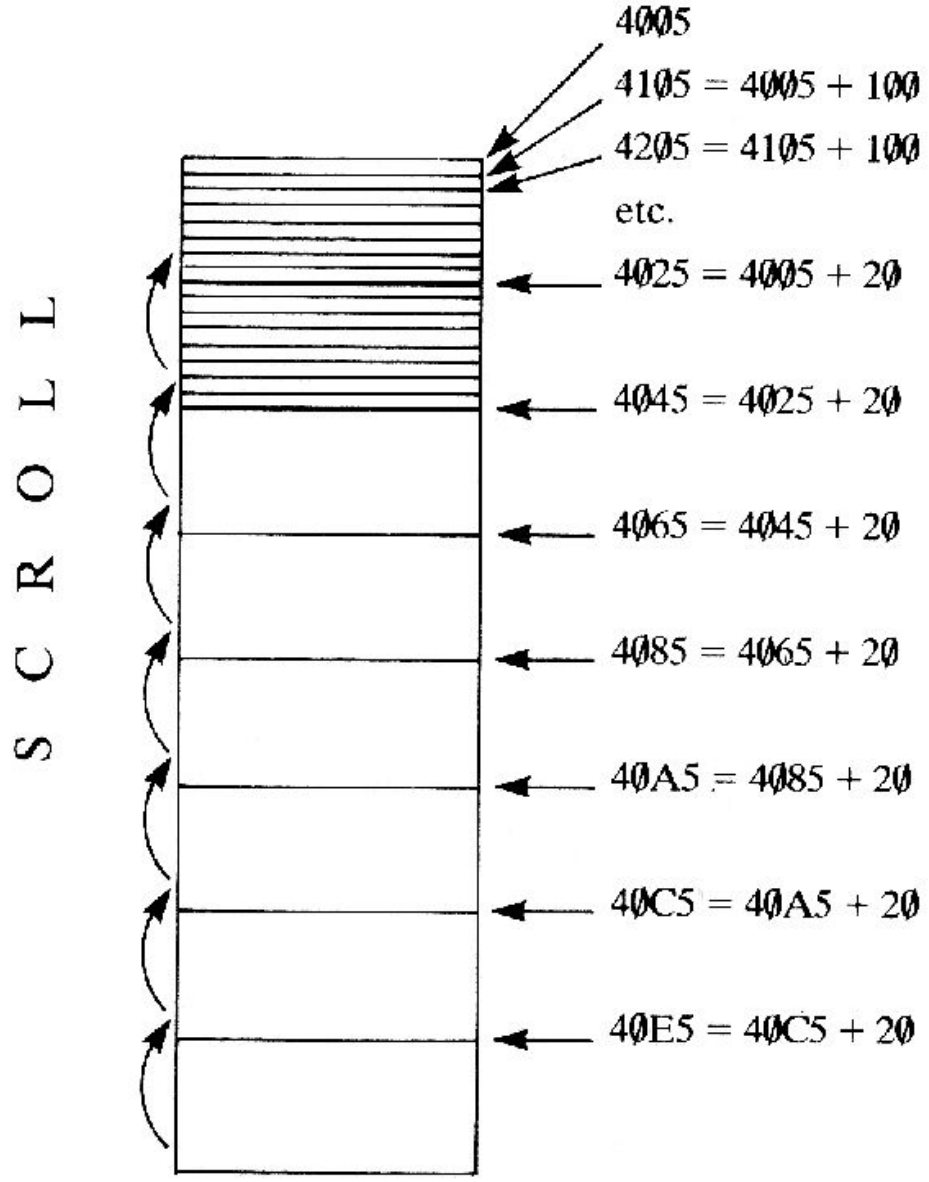
Figuur 15.4 ...en erna.

EEN ENKELE KOLOM SCROLLEN

Het scrollen van één enkele kolom op het scherm is iets ingewikkelder. Om het probleem enigszins te vereenvoudigen zullen we slechts in één van de drie blokken scrollen; dit komt neer op het scrollen van een stukje van de kolom

dat bestaat uit acht rijen. Voor de duidelijkheid werken we in blok 1, en met kolom 5.

Kolom 5 wordt in stukjes van 8-lijnen verdeeld, elk stuk komt overeen met één karakter (en met 1 PRINT AT-geheugenlocatie) en elke lijn wordt opgeslagen in een adres, zoals hieronder wordt getoond:



Om spraakverwarring te vermijden, noemen we elk vierhoekje van 8 lijnen een *rij* (zoals dit het geval is voor een PRINT AT-statement) en noemen we elke lijn afzonderlijk een *lijn*. In dit geval zit de bovenste lijn van rij Ø vervat in adres 4ØØ5 (hex) van de display file. Door 32 (decimaal) (of 2Ø hex) erbij te tellen, krijgen we de bovenste lijn van rij 1, en zo gaat het verder naar onder tot rij 7.

Om de tweede lijn van rij Ø te verkrijgen, moeten we 256 (dec) of 1ØØ (hex) bij het originele adres 4ØØ5 optellen; dus als volgt:

4ØØ5 + Ø1ØØ = 41Ø5

Door achtereenvolgende keren 2Ø (hex) op te tellen bij dit laatste adres verkrijgen we de tweede lijnen van de zeven andere rijen. Vervolgens komen de derde lijnen aan de beurt, . . . en tenslotte eindigen we met de achtste lijnen.

Dat is de structuur van de kolom (blok 1). Om een willekeurige byte met één rij opwaarts te scrollen, verplaatsen we de byte naar het adres dat 32 (decimaal) plaatsen vroeger voorkomt. We wensen onze scroll zo uit te voeren dat de bovenste rij volledig verloren gaat en we willen aan het eind de zevende rij blanco laten.

Deze opdracht heeft veel weg van de kolomscroll van de attributen waarvoor we reeds een routine hebben geschreven, maar verschilt ervan omdat we nu acht maal een lus moeten maken opdat alle acht lijnen van een vierkantje verplaatst worden. (Een routine die slechts het bovenste achtste deel van een karakter scrollt heeft weinig nut – en heeft helemaal geen zin indien de karakters blanco's zijn!)

We moeten dus, zoals voorheen, indexeren met een verplaatsing van Ø of 32.

In de hoop dat alles er duidelijker op zal worden, geven we eerst een BASIC-versie van de routine.

```
2ØØØ  LET ix = 256 * 64 + 5
2Ø1Ø  FOR l = 1 TO 8
2Ø2Ø  FOR a = 1 TO 7
2Ø3Ø  LET b = PEEK (ix + 32)
2Ø4Ø  POKE ix, b
2Ø5Ø  LET ix = ix + 32
2Ø6Ø  NEXT a
2Ø7Ø  POKE ix, Ø
2Ø8Ø  LET ix = ix + 32
2Ø9Ø  NEXT l
```

We hebben in dit geval variabelen met kleine letters, zoals ix, gebruikt die overeenkomen met inhouden van registers die in hoofdletters worden genoteerd, zoals IX. De a-lus schuift alle bovenste lijnen naar boven op; regel 2Ø7Ø POKEt de blanco op de zevende rij; en de l-lus herhaalt de hele werkwijze voor de 2de, 3de, ..., 8ste lijn van een karakter.

Maak een programmaatje waarmee u deze routine kunt testen en gebruik hierbij LIST 2ØØØ: GO TO 2ØØØ.

Indien u deze routine probeert, zult u vaststellen dat ze inderdaad werkt, maar dat ze nogal traag is: u kunt de karakters werkelijk "druppelsgewijs" opwaarts zien verschuiven. Maar we zijn op het goede pad. (Deze werkwijze is een nuttig truukje bij het debuggen: eerst schrijft u een BASIC-versie van de routine; aan de hand van deze routine kunt u nakijken of het wel een verstandige redenering was; indien er fouten in zaten kunt u eerst *die* routine debuggen; daarna pas stapt u over naar de machinecode.) Als we de BASIC-versie in machinecode omzetten, geeft dit:

	LD DE, 0020	11 20 00
	LD IX, 4005	DD 21 05 40
	LD L, 08	2E 08
lus 2:	LD A, 07	3E 07
lus 1:	LD B, (IX + 20)	DD 46 20
	LD (IX), B	DD 70 00
	ADD IX, DE	DD 19
	DEC A	3D
	CP D	BA
	JRNZ lus 1	20 F4
	LD (IX), 00	DD 36 00 00
	ADD IX, DE	DD 19
	DEC L	2D
	PUSH AF	F5
	LD A, 0	3E 00
	CP L	BD
	POP AF	F1
	JRNZ lus 2	20 E4

Om dit te testen voegen we een BASIC-stukje toe:

```
1000  FOR i = 0 TO 21: FOR j = 0 TO 31:
      PRINT CHR$ (65 + i); : NEXT j: NEXT i
```

Daarna geven we GO TO 1000 en dan runnen we de routine (ofwel via GO TO 300 ofwel via een directe RANDOMIZE USR 3200-instructie). Het bovenste blok van kolom 5 scrollt inderdaad opwaarts en met een aanzienlijk tempo. Indien we de machinecode in BASIC in een lus leggen, krijgen we een redelijk snelle scroll:

```
3000  FOR k = 1 TO 8: RANDOMIZE USR 32000: NEXT k
```

Indien we de lus in machinecode leggen dan gaat alles zo snel dat we de karakters nauwelijks kunnen zien verdwijnen.

MEERDERE KOLOMMEN SCROLLEN

Nu deze routine werkt is het maar een kleine stap meer om ze uit te breiden tot een routine die een hele zone van kolommen uit een blok aan het scrollen

brengt: het voorbereidend werk zal echter iets meer tijd in beslag nemen.

Reserveer vier databytes (vanaf 7DØØ tot en met 7DØ3), die de startkolom, het blok, de breedte van de zone waarover moet worden gescrolld en een dummy nul die we nodig hebben om esthetische redenen, bevatten:

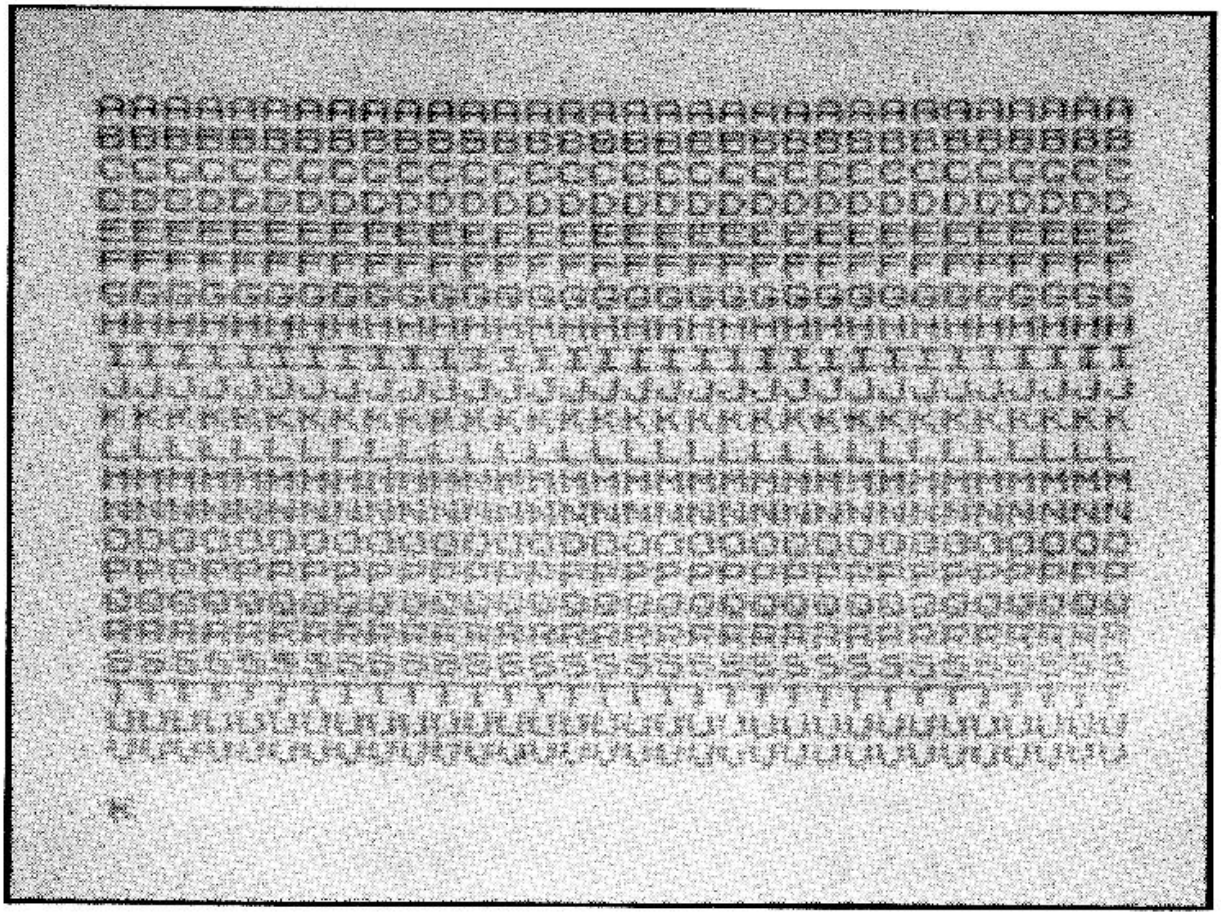
7DØØ	kolom	(bijv. Ø5 om kolom 5 te activeren)
7DØ1	blok	(4Ø, 48 of 5Ø voor de drie blokken)
7DØ2	breedte	(bijv. 11 voor een breedte van 17 (dec))
7DØ3	ØØ	(dummy)

De machinecode start op 7DØ4 (en wordt aangeroepen d.m.v. RANDOMIZE USR 32ØØ4). De routine ziet er als volgt uit (bemerk dat het middenstuk een herhaling is van voorgaande routine):

	LD BC, (7DØ2)	ED 4B Ø2 7D
	LD DE, ØØ2Ø	11 2Ø ØØ
	LD IX, (7DØØ)	DD 2A ØØ 7D
lus 3:	PUSH IX	DD E5
	LD L, Ø8	2E Ø8
lus 2:	LD A, Ø7	3E Ø7
lus 1:	LD B, (IX + 2Ø)	DD 46 2Ø
	LD (IX), B	DD 7Ø ØØ
	ADD IX, DE	DD 19
	DEC A	3D
	CP D	BA
	JRNZ lus 1	2Ø F4
	LD (IX), ØØ	DD 36 ØØ ØØ
	ADD IX, DE	DD 19
	DEC L	2D
	PUSH AF	F5
	LD A, Ø	3E ØØ
	CP L	BD
	POP AF	F1
	JRNZ lus 2	2Ø E4
	POP IX	DD E1
	DEC C	ØD
	PUSH AF	F5
	LD A, Ø	3E ØØ

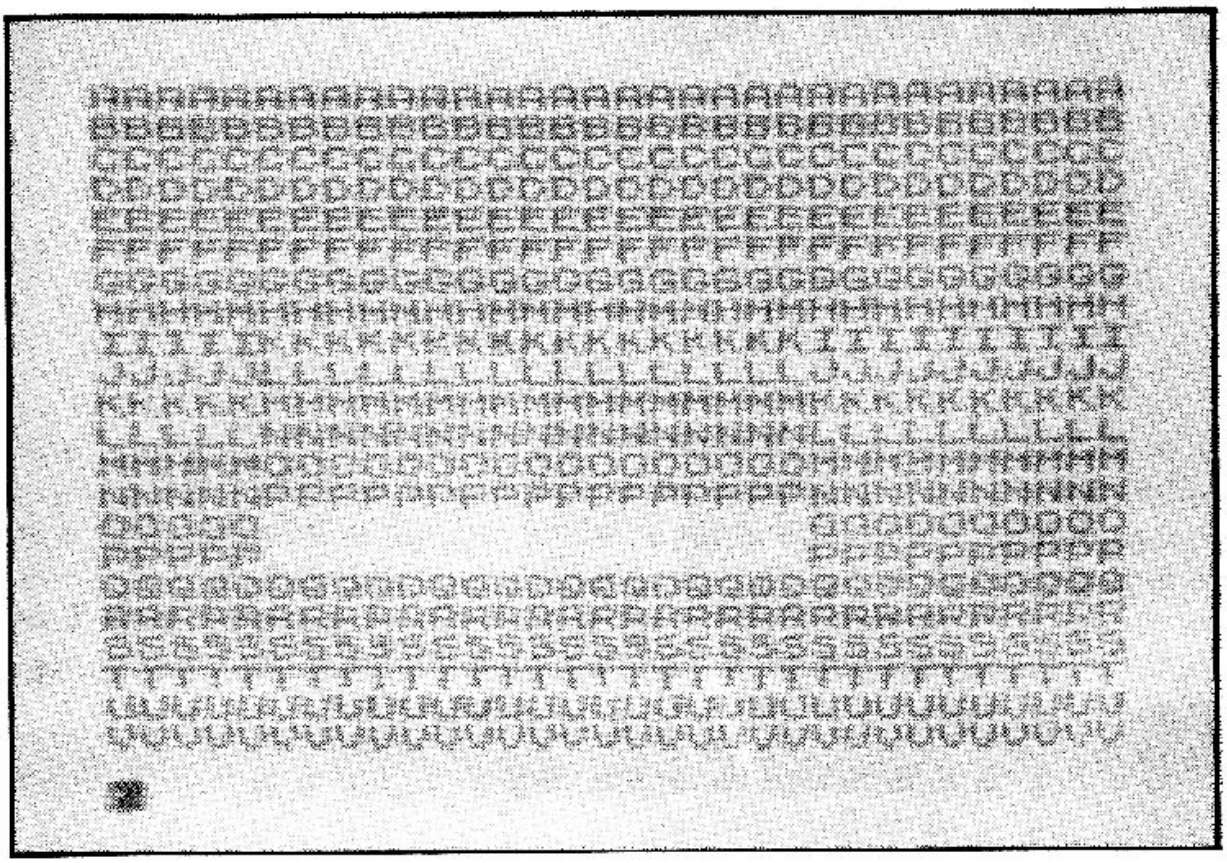
	CP C	B9
	POP AF	F1
	JRZ sla over	28 Ø4
	INC IX	DD 23
	JR lus 3	18 D2
sla over	(RET	C9)

Om deze routine te kunnen gebruiken moeten de juiste getallen in de databytes gePOKEt worden (bijv. 5, 64, 17, Ø – vergeet niet dat POKE met decimale getallen en niet met hexadecimale getallen werkt!) Daarna gebruikt u GO TO 1ØØØ samen met het kleine testprogramma uit de vorige paragraaf. Tenslotte GO TO 3ØØ met optie "r", of RANDOMIZE USR 32ØØ4. Daar gaan we dan!



Figuur 15.5 Vóór de scroll van een zone van het scherm...

Probeert u maar de lus in BASIC te leggen: dit blijkt erg effectief te zijn. U kunt een dergelijke routine gebruiken voor blok 2 en er bijvoorbeeld een mooi "jackpot"-programma mee maken...



Figuur 15.6 ...en het resultaat, twee scrolls later.

Een laatste opmerking. Toen we deze routine voor de eerste maal uitprobeerden was er een klein foutje in de laatste twee regels geslopen: we hadden POP AF *na* de JRZ-*sla over* geplaatst. Het resultaat was dat:

1. Het scherm op een onberispelijke manier scrollde, maar
2. Dat de foutmelding "C: Nonsense in BASIC Ø: 1" verscheen.

Dit kan u ook overkomen. Het wijst erop dat er iets is mis gelopen met de *stack;* en, inderdaad, indien de code in die volgorde staat springt de JR-*sla over* aan het einde over het POP AF-commando. Dus bevat het return adres in de stack (dit is een 2-byte adres) – wanneer de Spectrum terug naar BASIC probeert over te gaan – nog steeds de overblijvende bit van het AF-register, en hierdoor wordt het systeem in de war gebracht.

16 Meer over flags

Tot nu toe hebben we getracht de technische details over het F-register in het midden te laten; toch hebben we, telkens als we een voorwaardelijke sprong maakten, flags gebruikt. Het is dus verantwoord dat we dit onderwerp weer uitdiepen, we zullen echter niet alle details behandelen want de exacte werking van het F-register is één van de meest ingewikkelde materies van de Z8Ø.

In het F-register is er plaats voor acht bits maar er worden er slechts zes van gebruikt. Deze zijn:

C	Carry-flag (overdrachtvlag)
Z	Zero-flag (nulvlag)
S	Sign-flag (tekenvlag)
P/V	Parity/Overflow-flag (pariteit/overflowvlag)
H	Half-carry-flag (halve overdrachtvlag)
N	Subtract flag (aftrekvlag)

Ze komen als volgt in het F-register voor:

S	Z	X	H	X	P/V	N	C

X betekent "niet gebruikt".

De Carry-flag wordt hoofdzakelijk beïnvloed door de commando's: add (tel op), subtract (trek af), rotate (roteer) en shift (schuif).

De Zero-flag wordt beïnvloed door bijna alle commando's! Ruwweg kunnen we stellen dat indien iets (behalve LD, INC en DEC) de inhoud van A verandert de Zero-flag wordt geset op 1 indien A nul is of dat ze wordt gereset (op Ø) in het andere geval. BIT set de flag indien de gespecificeerde bit nul is. CP (vergelijk) set of reset de flag naargelang het resultaat van de vergelijking.

De Sign-flag slaat de tekenbit op van het resultaat van een willekeurige operatie die het laatst werd uitgevoerd: 1 staat voor negatief, Ø voor positief.

De P/V-flag werkt op twee verschillende manieren, afhankelijk van de operatie (deze kan een rekenkundige of een logische operatie zijn). Indien er rekenkunde wordt uitgevoerd dan wordt de P/V-flag op 1 geset indien er een *overflow* is in de 2-complement rekenkunde (bijv. indien de som van twee positieve getallen te groot is voor de accumulator dan geeft dit schijnbaar negatieve resultaten.) Bij logische operaties wordt de P/V -flag op Ø gereset indien

de byte in A een *even* aantal bits die 1 zijn bevat; en ze wordt op 1 geset indien het aantal bits die 1 zijn *oneven* is. Men noemt dit even/oneven karakter van de bits de *pariteit* van de byte. Bijvoorbeeld:

A bevat Ø11Ø11ØØ: even pariteit, P/V-flag Ø
A bevat 1ØØ1ØØØ1: oneven pariteit, P/V-flag 1.

De H- en de N-flag worden slechts gebruikt in binair gecodeerde decimale berekeningen en het is onwaarschijnlijk dat u ze op de Spectrum nodig zult hebben. Meestal dienen ze om uitwendige uitvoerapparaten te sturen. In appendix 5 vindt u een opsomming van de operaties die eventueel de C- en de Z-flag beïnvloeden. Als u pas met de Spectrum begint te werken zijn dit de belangrijkste flags.

We hebben in een aantal routines de Zero-flag gebruikt, bijvoorbeeld telkens wanneer we JRZ, JRNZ of DJNZ nodig hadden. Meestal worden deze voorwaardelijke sprongen voorafgegaan door een *vergelijkings*-commando zoals:

CP B

Deze instructie *set de flags* op dezelfde manier als SUB A, B maar ze wijzigt de inhoud van A niet. Indien A en B dezelfde byte bevatten is het resultaat van de aftrekking Ø en wordt de Zero-flag *geset* (op 1). Indien A en B verschillend zijn dan wordt de Zero-flag gereset.

De Carry-flag hebben we tot nu toe niet dikwijls gebruikt. Deze flag is vooral nuttig in een lusteller wanneer u niet zeker weet of u het luseinde precies zult bereiken: de kans bestaat dat u dit einde voorbijschiet. Dit moeten we even verduidelijken.

Stel dat u wilt weten of het getal dat in het HL-register zit groter is dan dat in het BC-register. Het commando:

SBC HL, BC ED 42

zal dan de Carry-flag setten indien BC groter is dan HL, en het zal de flag resetten indien BC kleiner dan of gelijk is aan HL. Laat op dit commando een voorwaardelijke sprong volgen:

JRC ergens naartoe 38 ??

U zult dan vaststellen dat de sprong wordt gemaakt indien BC groter is dan HL. Indien het SBC-commando wordt gevolgd door:

JRNC ergens naartoe 3Ø ??

dan zal de sprong worden gemaakt indien BC gelijk is aan of kleiner is dan HL. (Om de grootte van de sprong te bepalen worden alle getallen als positief beschouwd: dus geen 2-complement rekenkunde!)

HET OPNIEUW NUMMEREN VAN DE REGELS

Het voorgaande kunnen we illustreren aan de hand van een rudimentaire rou-

tine die de regelnummers opnieuw nummert. Deze routine loopt gewoon door het BASIC-programmadeel en zet de regelnummers in de regelmatige volgorde 1Ø, 2Ø, 3Ø, . . . enzovoort in stappen van 1Ø tot het einde wordt bereikt.

Voor we dit kunnen doen, moeten we weten op welke manier de BASIC-regels in het geheugen zitten opgeslagen.

Het BASIC-programma wordt in het geheugen opgeslagen tussen de adressen die in PROG en VARS zitten. Zonder disc drives is PROG 23755. Elke regel wordt op de volgende manier opgeslagen:

NS	NJ	LJ	LS	Code voor regel	ENTER

NS en NJ zijn respectievelijk de senior en de junior byte van het regelnummer. LJ en LS zijn respectievelijk de junior en de senior byte van het getal dat het aantal karakters op de regel aangeeft. (In dit aantal is de code voor ENTER *wel* maar de eerste vier bytes NS, NJ, LJ en LS *niet* begrepen.) Bemerk dat in het regelnummer de senior byte eerst voorkomt, dit is *geen* zetfout. Volgend testprogramma:

```
1 REM 12345678901
5 PRINT a
```

wordt als volgt in het geheugen opgeslagen:

Adres	Inhoud	Commentaar
23755	Ø	Senior byte van eerste regelnummer
23756	1	Junior byte van eerste regelnummer
23757	13	Junior byte van de lengte van de regel
23758	Ø	Senior byte van de lengte van de regel
23759	234	Code voor REM
23760	49	Code voor 1
23761	50	Code voor 2
23762	51	Code voor 3
23763	52	Code voor 4
23764	53	Code voor 5
23765	54	Code voor 6
23766	55	Code voor 7
23767	56	Code voor 8

23768	57	Code voor 9
23769	48	Code voor Ø
23770	49	Code voor 1
23771	13	Code voor ENTER
23772	Ø	Senior byte van 2de regelnummer
23773	5	Junior byte van 2de regelnummer
23774	3	Junior byte van de lengte van de regel
23775	Ø	Senior byte van de lengte van de regel
23776	245	Code voor PRINT
23777	97	Code voor a
23778	13	Code voor ENTER
23779		Begin van het VARIABELEN gebied

U kunt dit nakijken door de PEEK-routine te gebruiken die u vindt in *Basic met de ZX-Spectrum,* op p. 88, of – beter nog – schrijf er zelf eens een routine voor!

Nu we dit allemaal weten ligt de werkwijze voor het hernummeren van de programmaregels blijkbaar voor de hand: we zoeken in het programmagebied naar ENTER-bytes (code 13), de twee bytes die daar op volgen zijn de regelnummers, rest ons nog slechts deze bytes aan te passen.

Dit is prachtig, máár het zal niet werken omdat de byte 13 ook elders tevoorschijn kan komen; met name in de bytes die de lengte van de regel aangeven. Dit is inderdaad het geval op adres 23757. (Daarom hebben we juist dit testprogramma uitgekozen).

De bytes die de lengte van de programmaregel aangeven *vertellen* ook hoe ver we moeten gaan om het volgende regelnummer tegen te komen. Het is dus overbodig op zoek te gaan naar de ENTERs!

HET PROGRAMMA

We beginnen bij de eerste twee bytes van het programmagebied: het eerste regelnummer; deze bytes kunnen we aanpassen; dan gebruiken we de volgende twee bytes die de lengte van de programmaregel aangeven om een sprong te maken naar het volgende regelnummer, en zo gaan we door tot we het VARS-gebied bereiken.

Eerst en vooral moeten we de adressen opslaan. We gebruiken BC om de VARS-waarde in op te slaan (daar zullen we stoppen), HL om PROG in op te slaan (daar zullen we van vertrekken) en DE om het nieuwe regelnummer in op te slaan. Dit wordt als volgt geïnitialiseerd:

LD BC, (VARS)	ED 4B 4B 5C
LD HL, (PROG)	2A 53 5C
LD DE, 10 dec	11 0A 00

Raadpleeg appendix 3 voor de adressen van de systeemvariabelen PROG en VARS.

Vervolgens wensen we na te kijken (check) of het HL-register (dat we zullen gebruiken als pointer die het programmagebied doorloopt omdat HL zich uitstekend leent tot indirectie) de waarde overtreft die in het BC-register zit opgeslagen. Indien dit het geval is stoppen we. En nu komt de Carry-flag op de proppen:

check:	PUSH HL	E5
	SBC HL, BC	ED 42
	POP HL	E1
	JRNC end	30 15

(De 15 aan het eind kan eigenlijk pas worden uitgewerkt wanneer het volledige programma geschreven is; dan zult u vaststellen dat het 15 moet zijn.)

Nu volgt de eigenlijke hernummering:

hernummer:	LD (HL), D	72
	INC HL	23
	LD (HL), E	73

Vervolgens moeten we de twee volgende bytes lezen om de lengte van de regel te weten te komen. We moeten het resultaat ergens opslaan; het HL-register wordt in beslag genomen voor berekeningen, dus lijkt het BC-register het best geschikt. Jammer genoeg is het in gebruik, maar dat kunnen we oplossen door de lopende waarde in de stack weg te zetten en ze er later weer uit op te halen.

PUSH BC	C5
INC HL	23
LD C, (HL)	4E
INC HL	23
LD B, (HL)	46

Nu verschuiven we HL naar het volgende regelnummer:

ADD HL, BC	09
INC HL	23

en we zorgen ervoor dat we in BC de oude waarde terug krijgen; die we eerder op de stack hadden geplaatst:

POP BC	C1

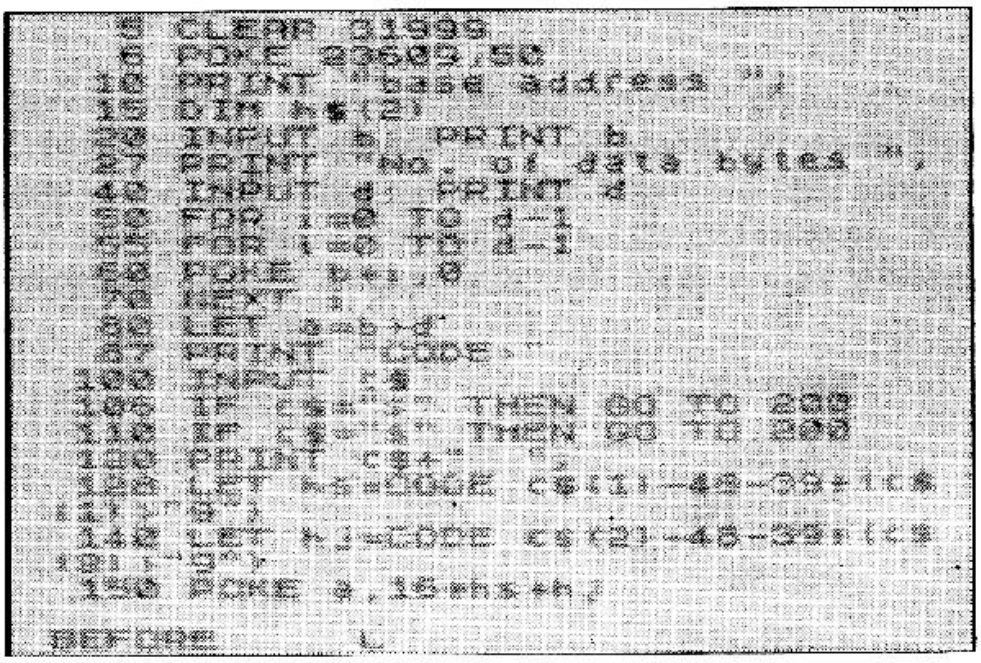
Tenslotte willen we DE op de correcte waarde voor de volgende regel setten door er 10 bij op te tellen; dit houdt in dat we nog wat meer in de stack moeten onderbrengen:

PUSH HL	E5
LD HL, 10 dec	21 0A 00
ADD HL, DE	19
LD D, H	54
LD E, L	5D
POP HL	E1

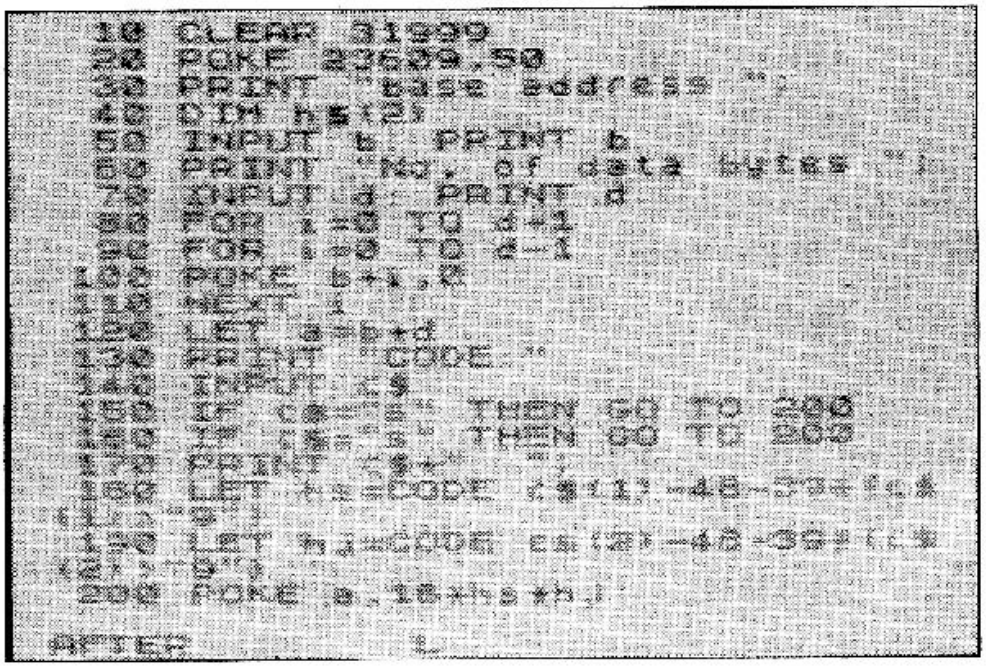
Nu leggen we een lus terug naar het *check*-niveau zodat alles wordt herhaald totdat HL te groot is:

JR check 18 E5

Tenslotte verschijnt het eindcommando, RET, op de plaats die we *end* hebben genoemd.



Figuur 16.1 Voor de hernummering...



Figuur 16.2 ...en erna. Bemerk dat de GO TO's niet werden veranderd.

Laad de hele routine in de gegeven volgorde (vergeet het finale RET-commando niet). Indien u nu het volgende invoert:

RANDOMIZE USR 32000

zal elk programma dat zich in het BASIC-gebied bevindt, worden voorzien van nieuwe regelnummers.

Het zijn natuurlijk alleen maar de regelnummers die veranderen; GO TO's en GO SUB's zullen niet meer overeenstemmen.

17 Bloksearch en bloktransfer

Enkele routines die we tot dusver hebben gegeven, werden niet in hun meest efficiënte vorm geschreven. We hebben er naar gestreefd alles zo eenvoudig mogelijk te houden. Machinecode is niet het gemakkelijkste onderwerp en door alle facetten ervan op een rijtje te zetten schept men verwarring. Indien u reeds andere boeken over Z8Ø machinecode of listings in tijdschriften hebt ingekeken; zult u zich misschien afvragen waarom we bepaalde zaken op een nogal stuntelige manier hebben opgelost. In dit en in het volgende hoofdstuk zullen we trachten de balans weer in evenwicht te brengen.

BLOKSEARCH

Om te beginnen zijn er enkele zeer krachtige instructies die een volledig geheugenblok onderzoeken. Een voorbeeld van zo een instructie is CPDR, dit betekent compare, decrement and repeat (vergelijk, decrementeer en herhaal).

Wat gebeurt er indien we een stukje code op de volgende manier schrijven?

```
        LD BC, Ø1ØØ          Ø1 ØØ Ø1
        LD HL, 5ØØØ          21 ØØ 5Ø
        LD A, Ø5              3E Ø5
        CPDR                  ED B9
next:   — —
```

Wanneer de CPDR-instructie wordt tegengekomen, wordt de waarde uit het A-register vergeleken met de inhoud van de byte waarnaar HL wijst. Indien ze gelijk zijn, wordt overgegaan naar *next*. Indien beide waarden verschillend zijn, worden BC en HL met 1 gedecrementeerd en wordt de "compare"-operatie herhaald totdat de inhoud waarnaar HL wijst aan A gelijk is of totdat BC nul bevat. Met andere woorden komen deze 4 instructies neer op: "Zoek de adressen vanaf 5ØØØ (hex) tot 4FØØ af naar de eerste byte die Ø5 bevat en laat HL er naar wijzen. Indien zo'n byte niet wordt gevonden, maak dan BC nul".

Dus had het eerste voorbeeld waarin we sprongen hebben gebruikt – en dat een kleine "compare"-lus was – veel eenvoudiger gekund, maar in dat

geval was het natuurlijk geen illustratie van sprongen geweest!

Er bestaat een gelijksoortig commando, CPIR, dat het HL-register incrementeert maar voor de rest op dezelfde manier werkt. Dit commando onderzoekt dus een geheugenblok vanaf de andere kant.

BLOKTRANSFER

De bloktransfercommando's (bloktransport) LDIR en LDDR verschuiven blokken gegevens in het geheugen. Indien u LDIR gebruikt dan:

1. Laadt u HL met het adres van de eerste byte die moet worden getransporteerd.
2. Laadt u DE met het adres van de eerste bestemmingsbyte.
3. Laadt u BC met het aantal te transporteren bytes.

Als dit gebeurd is verplaatst de instructie LDIR de eerste byte, incrementeert ze HL en DE, decrementeert ze BC en blijft ze dit herhalen totdat BC de waarde Ø bereikt.

Het is belangrijk dat men weet dat bij de laatste stap HL en DE wel worden geïncrementeerd maar dat geen transport meer plaatsvindt. Dus wijst HL aan het eind de byte aan die zich onmiddellijk achter het transportgebied bevindt, en wijst DE het bovenste uiteinde van het transportgebied aan.

LDDR werkt op analoge wijze maar decrementeert HL en DE en blijft BC decrementeren zoals LDIR dit doet – BC is slechts een teller.

PRESET ATTRIBUTES

Een manier om het bloktransport te gebruiken is een "valse" attributes file in de RAM samenstellen en deze dan transporteren naar de echte attributes file – daarbij worden alle attributes ogenblikkelijk veranderd in een nieuw, vooropgesteld (preset) patroon. U heeft hiervoor 7Ø4 databytes nodig die de nieuwe attributen bevatten. Om dit te verkrijgen moeten we het LOADER-programma aanpassen. Edit regel 1Ø en verander deze in:

```
1Ø CLEAR 31599
```

zodat voldoende geheugenruimte vrijkomt. Nu RUNt u en deelt u mee dat er 7Ø4 databytes zijn. De code is:

LD HL, 316ØØ dec	21 7Ø 7B
LD DE, a-file	11 ØØ 58
LD BC, 7Ø4 dec	Ø1 CØ Ø2
LDIR	ED BØ

316ØØ (7B7Ø hex) is het begin van het nieuwe datagebied. De routine zelf wordt geladen vanaf het startadres 323Ø4.

Nu moeten we de "valse" file samenstellen. Bijvoorbeeld:

```
5000  FOR w = 31600 TO 32303
5010  IF w < 31900 THEN POKE w, 48
5020  IF w > = 31900 THEN POKE w, 32
5030  NEXT w
```

Typ GO TO 5000 (en wacht!) om deze file samen te stellen. Zorg er dan voor dat u een display op het scherm krijgt (dit is zeer gemakkelijk te doen d.m.v. LIST) en tenslotte typt u RANDOMIZE USR 32304. U krijgt nu gele en groene papergebieden die overeenkomen met de attributes die het programma op regel 5000 heeft ingePOKEt. U kunt ongetwijfeld meer spectaculaire BASIC-routines bedenken om de valse file samen te stellen. (Strepen of blokpatronen bijvoorbeeld.)

WAAR WE MEE BEGONNEN ZIJN...

Nu moet u in staat zijn uit te zoeken wat de inleidende programma's uit hoofdstuk 1 doen en hoe ze precies werken. De machinecode commando's van deze programma's zitten in de DATA-statements, en ze werden in decimale en niet in hex notatie geschreven omdat ze gemakkelijker konden worden ingevoerd. Indien u de decimale getallen in hex getallen omzet en ze daarna opzoekt (in de tabel van de opcodes in het *Handboek* staan ze in numerieke volgorde gelist) zult u vaststellen dat ze alle bloktransporten naar de attributes- of de display files bevatten. De overgehevelde bytes worden uit de ROM genomen, vandaar dat ze er meestal erg willekeurig uitzien. Bepaalde delen van de ROM (welke?) zijn echter meer gestructureerd dan andere, daarom ziet u hier en daar patronen in de displays.

ZIJDELINGS SCROLLEN VAN ATTRIBUTES

Edit het LOADER-programma zodat regel 10 opnieuw 31999 bevat. (We hebben geen behoefte aan zoveel geheugenruimte).

Een zeer eenvoudige toepassing van LDIR preset de attributes. Maar nu volgt een meer geraffineerde toepassing: een rij attributes één spatie naar links laten scrollen en daarbij het uiterst linkse attribute aan de rechterkant plaatsen; dus net alsof het scherm rond een draaiende cilinder gewikkeld is. Om het niet te moeilijk te maken, doen we dit slechts voor rij 0:

```
        LD HL, a-file          21 00 58
lus:    LD D, H                54
        LD E, L                5D
        INC HL                 23
        LD BC, 31 dec          01 1F 00
```

LD A, (DE)	1A	
LDIR	ED B0	
LD (DE), A	12	

Stel iets samen om dit te testen:

```
6000  FOR s = 0 TO 31: PAPER s/6: PRINT AT 0, s; "□"; : NEXT s
```

en typ daarna RANDOMIZE USR 32000 om de scroll te zien. Om een echte scroll te zien moet u een BASIC-lus maken:

```
FOR t = 1 TO 500: RANDOMIZE USR 32000: NEXT t
```

Kijk maar eens hoe alles nu ronddraait!

Door dit 22 maal met geschikte startvoorwaarden in een lus te leggen, kunt u een volledige schermbreedte met attributes een zijdelingse scroll laten uitvoeren. We zullen nu een analoog probleem aanpakken, namelijk het zijdelings scrollen van een lijn van de *display* file. (Hierbij is de lus reeds ingebouwd.)

ZIJDELINGSE SCROLL VAN DE DISPLAY

Hier volgt de code:

	LD A, 8 dec	3E 08	[lustelling]
	LD DE, start	11 00 40	[start lijn]
set:	LD H, D	62	
	LD L, E	6B	
	INC HL	23	
	PUSH DE	D5	[save start regel]
	LD BC, 31 dec	01 1F 00	
	PUSH AF	F5	[save teller]
	LD A, (DE)	1A	[save eerste byte]
	LDIR	ED B0	[scroll links]
	LD (DE), A	12	[laad eerste byte opnieuw]
	POP AF	F1	[herneem lustelling]
	DEC A	3D	[vermeerder telling]
	CP B	B8	[test voor einde]
	JRZ skip	28 04	
	POP DE.	D1	

```
        INC D           14          [volgende regel]
        JR  set         18 EB
sla over:POP DE         D1
```

Integreer dit in nog een lus (of lussen, zowel blok-per-blok als in een blok van 8 rijen) en u hebt een routine die de hele display zijdelings scrollt. Natuurlijk zijn er vele varianten mogelijk. Bestudeer de routine aandachtig en zoek naar eventuele wijzigingen die het proberen waard zijn.

18 Enkele dingen waar we nog niet over gesproken hebben

BIT-FLIPPING

Misschien is het nuttig enkele eenvoudige commando's te geven die de bits veranderen. Het eerste is (change Carry-flag):

CCF 3F

dat de Carry-flag van Ø in 1 verandert of omgekeerd. Om de Carry-flag op 1 te setten, gebruiken we (set Carry flag):

SCF 37

Om de Carry-flag op Ø te *resetten* gebruiken we achtereenvolgens SCF en CCF.

Om het volledige A-register te complementeren (d.i. iedere bit in zijn tegengestelde bit te doen omslaan) gebruiken we:

CPL 2F

VARIATIES IN MNEMONISCHE CODES

Sommigen gebruiken licht verschillende mnemonische opcodes dan wij hebben gedaan. Waar wij LD A, (nn) schreven zult u bij anderen soms LD (nn) vinden omdat het A-register het enige register is dat direct kan worden geladen. Het is dus niet nodig te specificeren dat het A-register wordt geladen hoewel we dit een nuttig geheugensteuntje vinden.

ALTERNATIEVE REGISTERS

We hebben eerder gezegd dat er een alternatieve registerset bestaat, maar we hebben er nooit meer over verteld. In feite hebt u deze registers niet nodig en ze zijn ook niet erg nuttig. U kunt er geen rekenkunde in doen. Ze worden meestal gebruikt om er tijdelijk de inhoud in te bewaren van de hoofdset terwijl u een of andere routine uitvoert die de inhoud van het hoofdregister verandert op een manier die u niet wenst. In dit geval verwisselt u de inhouden van de hoofd- en de alternatieve registersets vóór en na de bewuste routine:

EX AF, AF'	08	verwissel AF met AF'
EXX	D9	verwissel BC, DE, HL met BC', DE', HL'
CALL . . .	CD ——	aanroep v.d. bewuste routine
EX AF, AF'	08	restore de registers
EXX	D9	

Natuurlijk kunt u hetzelfde resultaat bereiken door voor de CALL de te bewaren registerinhoud op de stack te PUSHen en hem erna weer vanaf te POPpen.

Gebruik *nooit* het IY-register, want de Spectrum heeft dit register nodig.

ROM-ROUTINES

De BASIC-interpreter in de ROM doet soms een beroep op routines zoals wij hebben gemaakt. Waarom *roepen* we deze routines dan niet eenvoudigweg *aan* in plaats van ze zelf te schrijven; in het algemeen zou dit veel moeite en geheugenruimte sparen. We hebben dit niet gedaan omdat we ons in *dit* boek hebben beperkt tot de machinecode van de Z80 en we zoveel mogelijk de speciale kenmerken van de Spectrum buiten beschouwing hebben gelaten. Indien alle voorbeelden een aantal adressen van de ROM zouden aanroepen dan zou u niet veel hebben geleerd!

BASIC + MACHINECODE SAVEN

Stel dat u een BASIC-programma hebt dat gebruik maakt van een machinecode die boven de RAMTOP zit opgeslagen (bijvoorbeeld op 32000). Hoe kunt u nu beide op een gemakkelijke manier SAVEn?

Een manier is achtereenvolgens:

```
SAVE "programma"

SAVE "m-code" CODE 32000, 600
```

te schrijven (600 is de lengte van het machinecode gebied: de eigenlijke lengte van de code schrijven is efficiënter, maar dat moet u zelf uitmaken). Het LOADen gebeurt als volgt:

```
LOAD "programma"
```

en als dat erin zit,

```
LOAD "m-code" CODE
```

Vergeet ook niet CLEAR 32000.

Een betere manier om deze procedure uit te voeren is: een korte BASIC-routine schrijven die dit allemaal voor u doet. Zoek een vrije regel – aan het eind bijvoorbeeld – en schrijf

9800 LOAD "m-code" CODE

(of wat ook de startregel van het BASIC-programma is). Nu voert u met de hand het volgende in:

SAVE "programma" LINE 9800

SAVE "m-code" CODE 32000, 600

Indien u dan, zoals gewoonlijk,

LOAD "programma"

intypt, wordt het programma geladen, begint het op regel 9800 te lopen, laadt het de machinecode en gaat het verder.

Indien u dit wenst, kunt u zelfs de SAVE-operatie als een BASIC-routine schrijven. Experimenteer hiermee.

MULTIPELE ROUTINES

Indien u aan het eind van verschillende machinecode routines RET-commando's voorziet, dan kunt u deze routines "staart-kop" aan elkaar schakelen. Elke routine kan worden aangeroepen d.m.v. RANDOMIZE USR (startadres van de gewenste routine.) U kunt een dergelijk blok machinecode in één keer SAVEn: u hoeft dus *niet* routine per routine te saven.

MACHINECODE EFFICIËNT GEBRUIKEN

In deze paragraaf willen we twee zaken verduidelijken die we bij het begin maar even hebben aangeraakt. We hebben gezegd dat er redenen kunnen zijn – andere dan de BASIC-beperking die inhoudt dat statements moeten worden geïnterpreteerd telkens als ze worden uitgevoerd – om een machinecode programma vlugger te doen lopen dan BASIC. We verduidelijken dit met een voorbeeld:

BASIC		Machine Code
10 FOR i = 20 TO 1 STEP −1		LD B, 14
.....	lus:	.....
50 NEXT i		DJNZ lus

In beide gevallen wordt een variabele met 1 gedecrementeerd telkens als de lus wordt uitgevoerd. Dit proces is echter veel ingewikkelder in BASIC dan het in machinecode is. De reden is de volgende: omdat BASIC een gedeelte van de tijd met decimale waarden te doen heeft, neemt de machine aan dat dit gedurende de hele tijd het geval is, dus trekt de machine in werkelijkheid 1.000000000 af en dit is even moeilijk als 1.58712684 af trekken. In werkelijkheid is dit een erg complexe en tijdrovende procedure. De machinecode ge-

bruikt echter één enkele, voor dit doel geconcipieerde, instructie. Het resultaat is dat dit ongeveer 100 maal vlugger gaat.

Het andere item dat we hebben verwaarloosd is dat de machinecode meer geheugen kan innemen dan haar BASIC-equivalent. Hier volgt een voorbeeld dat illustreert waarom dit zo is:

BASIC	MC	Aantal bytes
30 IF r = p AND p = q THEN		
LET p = w	LD HL, 5000	3
	LD A, (HL)	1
	LD HL, 5001	3
	SUB A, (HL)	1
	JRNZ nextbit	2
	LD HL, 5001	3
	LD A, (HL)	1
	LD HL, 5002	3
	SUB A, (HL)	1
	JRNZ nextbit	2
	LD HL, 5001	3
	LD A, (5003)	3
	LD (HL), A	1
volgende bit:		27 TOTAAL

De machinecode gaat ervan uit dat r, p, q and w respectievelijk worden opgeslagen in de bytes 5000, 5001, 5002 en 5003. In werkelijkheid is het niet zo eenvoudig omdat elk getal 5 bytes in beslag neemt en omdat de SUB in werkelijkheid een CALL zal zijn aan een floating-point-aftrekroutine. In elk geval zal de werkelijke code minstens evenveel, en waarschijnlijk meer dan 27 bytes in beslag nemen. De equivalente BASIC-regel heeft hier slechts 18 bytes voor nodig: 1 voor elk symbool (IF, =, w, AND, enzovoort), 4 voor het regelnummer en 1 voor de regeldelimiter. Hoe complexer het BASIC-statement wordt, hoe meer de machinecode versie het geheugen belast.

ANDERE PLAATSEN OM MACHINECODE OP TE SLAAN

Het opslaan van code boven de RAMTOP heeft het grote nadeel dat u de code niet direct als deel van een BASIC-programma kunt bewaren. Het voor-

deel is evenwel dat u andere zaken onder de machinecode *kunt* inladen. Maar er *zijn* alternatieve plaatsen.

Een geliefde truuk is alles opslaan in een REM-statement dat als eerste regel in het BASIC-programma voorkomt. Het eerste karakter na REM heeft normaal gesproken adres 23755. Dus begint u uw BASIC-programma als volgt:

```
1  REM XXXXXX...X
```

met voldoende X-tekens zodat hierin de code kan worden opgeslagen. Daarna POKEt u de machinecode in. De code wordt toegankelijk d.m.v. RANDOMIZE USR 23755 (of LET y = USR 23755, enzovoort), en ze kan worden geSAVEd, bovendien wordt ze niet vernietigd door RUN.

Een andere plaats om machinecode in op te slaan is de karakterstring. Deze karakterstring kan gemakkelijk worden gelocaliseerd (via de systeemvariabele VARS) op voorwaarde dat u deze string als eerste declareert – begin bijvoorbeeld met een stringarray die groot genoeg is:

```
1  DIM a$ (79)          [om 79 bytes op te slaan]
```

dan plaatst u de machinecode in a$ (1), a$ (2), enzovoort, terwijl u de string opbouwt. Het belangrijkste nadeel van deze methode is dat RUN of CLEAR de code uitvegen. Bovendien raakt het startadres wel eens zoek.

Tenslotte kunt u de code als DATA opslaan en boven de RAMTOP laden als deel van de BASIC, maar deze methode verspilt, zoals in hoofdstuk 1 reeds werd duidelijk gemaakt, teveel plaats.

DEBUGGEN

In machinecode zijn er geen ingebouwde debug mogelijkheden en hoewel HELPA u bij het editten van code zal helpen, werd dit programma toch niet ontworpen om te debuggen. (De verschillende andere programma's die de kwaliteit "nuttig om machinecode te debuggen" meekregen werden evenmin voor dit doel ontworpen!) De enige oplossing die er op dit ogenblik voor u in-zit is een *dry run* uitvoeren met het ouderwetse pen en papier (zie *BASIC met de ZX-Spectrum*). Natuurlijk kunt u "tracing statements" aan de machinecode toevoegen, maar let op de veranderingen in de adressen en de sprong-groottes.

Een bruikbare (maar schijnbaar stuntelige) truuk bestaat erin uw routine eerst in BASIC te schrijven en *dit* te debuggen: gebruik slechts BASIC-instructies die beantwoorden aan de machinecode. (Dit wil zeggen, *streef* de machinecode in BASIC *na.*) Indien het werkt, zal het zeer traagjes gaan, maar het is te debuggen.

Voor de echte ambitieuze lezers volgt nu een projectsuggestie: verbeter HELPA door een routine toe te voegen die de naam SINGLE-STEP draagt en die het programma commando per commando doorloopt en die hierbij de registers op het schema afbeeldt. U moet (a) een machinecode routine schrijven om alle registers in de stack te PUSHen en ze daarna in hex op het scherm

af te beelden; (b) hier een routine aan toe voegen die een invoer via het toetsenbord waarneemt; (c) een CALL aan deze routine tussen elke regel machinecode schrijven (gebruik de ** delimiters in HELPA om aan te duiden waar); (d) uitwerken hoe u naar BASIC kunt terugkeren indien u dit wenst.

Appendices

1 Hex/Decimaal conversie

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	-128	-127	-126	-125	-124	-123	-122	-121	-120	-119	-118	-117	-116	-115	-114	-113
9	-112	-111	-110	-109	-108	-107	-106	-105	-104	-103	-102	-101	-100	-99	-98	-97
A	-96	-95	-94	-93	-92	-91	-90	-89	-88	-87	-86	-85	-84	-83	-82	-81
B	-80	-79	-78	-77	-76	-75	-74	-73	-72	-71	-70	-69	-68	-67	-66	-65
C	-64	-63	-62	-61	-60	-59	-58	-57	-56	-55	-54	-53	-52	-51	-50	-49
D	-48	-47	-46	-45	-44	-43	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33
E	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17
F	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

2-complement

gewoon

2 Geheugenreserveringstabellen

Gebruik de volgende adressen om geheugen in blokken van 100-bytes boven de RAM te reserveren. Vergeet niet het adres net onder het gewenste adres te CLEARen.

Aantal gereser-veerde cijfers	16K		48K	
	Decimaal adres	Hex adres	Decimaal adres	Hex adres
100	32500	7EF4	65268	FEF4
200	32400	7E90	65168	FE90
300	32300	7E2C	65068	FE2C
400	32200	7DC8	64968	FDC8
500	32100	7D64	64868	FD64
600	32000	7D00	64768	FD00
700	31900	7C9C	64668	FC9C
800	31800	7C38	64568	FC38
900	31700	7BD4	64468	FBD4
1000	31600	7B70	64368	FB70
1100	31500	7B0C	64268	FB0C
1200	31400	7AA8	64168	FAA8

3 Adressen van de systeemvariabelen

POINTERS NAAR VERPLAATSBARE ADRESSEN:

	Decimaal	Hex
CHANS	23631	5C4F
PROG	23635	5C53
VARS	23627	5C4B
E-LINE	23641	5C59
WORKSP	23649	5C61
STKBOT	23651	5C63
STKEND	23653	5C65
RAMTOP	23730	5CB2
UDG	23675	5C7B
P-RAMT	23732	5CB4
CHARS	23606	5C36
BORDCR	23624	5C48
DFSZ	23659	5C6B
COORDS	23677–8	5C7D–E
DFCC	23684	5C84
DFCC-S	23685	5C85
SCR-CT	23692	5C8C
ATTR-P	23693	5C8D

VASTE ADRESSEN:

	Decimaal	Hex
DISPLAY FILE	16384	4000
ATTRIBUTES FILE	22528	5800
16K RAMTOP VALUE	32599	7F57
48K RAMTOP VALUE	65367	FF57
16K UDG VALUE	32600	7F58
48K UDG VALUE	65368	FF58
16K P-RAMT VALUE	32767	7FFF
48K P-RAMT VALUE	65535	FFFF
PRINTER BUFFER	23296	5B00
SYSTEM VARIABLES	23552	5C00
MICRODRIVE MAPS	23734	5CB6
CHARACTER SET	15360	3C00

4 Z8Ø commando's in een notedop

Hier volgt een opsomming van alle opcode mnemonics en de effecten die ze teweegbrengen. Bij de commando's die in de tekst uitvoeriger worden besproken, staat de bladzijde vermeld. De effecten die deze commando's op de flags hebben, werden niet opgenomen; raadpleeg hiervoor de Zilog Reference Card.

ADC	pag. 60	Tel op, de Carry-flag inbegrepen. Sla op in A of in HL.
ADD	pag. 50	Tel op, zonder de Carry-flag. Sla op in A of in HL.
AND	pag. 55	Logische AND op overeenkomstige bits: sla op in A.
BIT	pag. 67	BIT b, r set de Zero-flag overeenkomstig de waarde van de b-de bit van de byte in register r. De volgorde van de bits in elke byte is 76543210.
CALL	pag. 29	Roept een subroutine aan. Er bestaan voorwaardelijke aanroepen die worden aangeduid d.m.v. de letters: C (roep aan indien de Carry-flag geset is), M (indien de Sign-flag is geset – "het resultaat (na vergelijking) is negatief"), NC (indien de Carry-flag niet is geset), NZ (indien de Zero-flag niet is geset), P (indien de Sign-flag niet is geset – "het resultaat is positief"), PE (indien de Parity-flag is geset: vergeet dit maar); PO (indien de Parity-flag niet is geset: dit mag u ook vergeten); Z (indien de Zero-flag is geset). Wat de flags betreft, kunt u pag. 00, 00 raadplegen.
CCF	pag. 105	Complement Carry flag (d.i. verwissel Ø door 1 en omgekeerd).
CP	pag. 56	Vergelijk: de flags worden geset alsof iets van A wordt afgetrokken maar A zelf wordt niet veranderd.
CPD		Vergelijk en decrementeer. Vergelijk via HL en decrementeer daarna HL en BC.
CPDR		Vergelijk, decrementeer, herhaal: bloksearch. Werkt zoals CPD maar gaat door totdat ofwel het resultaat van het vergelijken Ø is of totdat BC Ø wordt.
CPI	pag. 100	Werkt zoals CPD maar nu wordt HL geïncrementeerd; BC wordt zoals bij CPD gedecrementeerd.
CPIR	pag. 101	Werkt zoals CPDR, maar HL wordt geïncrementeerd.
CPL	pag. 105	Complementeert (flipt de bits van) het A-register.

DAA		Decimaal aanpassen van de accumulator inhoud. Wordt gebruikt bij binair-gecodeerde decimale berekeningen: vergeet deze maar.
DEC	pag. 58	Decrementeer: verminder de waarde met 1.
DI		Disable interrupts: mag u vergeten.
DJNZ	pag. 59	Decrementeer, spring indien niet nul. Decrementeer B en maak behalve wanneer de Zero-flag is geset een relatieve sprong. Wordt in lussen net als de BASIC FOR/NEXT-statements gebruikt.
EI		Enable interrups: vergeet maar.
EX	pag. 106	Verwissel waardes. Instructies met (SP) verwisselen de registers HL, IX of IY met de top van de stack.
EXX	pag. 106	Verwissel de drie registerparen BC, DE en HL met hun alternatieven BC', DE' en HL'.
HALT		Wacht op een interrupt. GEBRUIK DIT NOOIT tenzij u hardware heeft aangesloten en u weet wat u doet, want anders zal het programma eeuwig blijven wachten.
IM		Interrupt mode: vergeet maar.
IN		Input via een apparaat: vergeet maar.
INC	pag. 58	Incrementeer: vermeerder de waarde met 1.
IND, INDR, INI en INIR		Inputcommando's die analoog zijn aan LDD, LDDR, LDI en LDIR. Vergeet ze maar.
JP	pag. 56	Spring. Varianten van JP verkrijgt men door C, H, NC, NZ, P, PE, PO of Z toe te voegen: in deze gevallen heeft men een voorwaardelijke sprong (de voorwaarden zijn dezelfde als voor CALL.)
JR	pag. 57	Relatieve sprong, gevolgd door een verplaatsing van 1 byte. Voorwaardelijke varianten zijn: C, NC, NZ, Z.
LD	pag. 40	Laad. Kan met elk van de vijf adresseermodes worden gebruikt.
LDD		Is niet hetzelfde als LD D! Laad datgene waar HL naar wijst in wat DE aanwijst: decrementeer BC, DE, HL.
LDDR	pag. 101	Laad, decrementeer, herhaal: bloktransport. Voer LDD uit totdat BC nul tegenkomt. Copieert een geheugenblok, waarvan de lengte in BC zit opgeslagen, van datgene wat HL aanwijst naar wat DE aanwijst.
LDI		Zoals LDDR behalve dat HL en DE worden geïncrementeerd.
LDIR	pag. 101	Zoals LDDR, behalve dat HL en DE worden geïncrementeerd.
NEG		Negatief: verander het teken van de inhoud van A.
NOP		Geen operatie. Doe niets gedurende één tijdscyclus (d.i. verspilde tijd.) Nuttig bij het debuggen om in-

OR	pag. 55	Logische OR op bits. In A opslaan.
OTDR, OTIR, OUT		
OUTD, OUTI		Verschillende outputs. Vergeet ze maar.
POP	pag. 61	Haal van de stack naar het aangeduide register.
PUSH	pag. 61	Breng van het register naar de stack.
RES	pag. 67	Reset een bit (d.i. maak hem nul).
RET	pag. 30	Terugkeer na een subroutine. Er zijn voorwaardelijke returns mogelijk die op dezelfde manier worden gevormd als bij CALL. (De voorwaarden bij een CALL hoeven niet noodzakelijk dezelfde zijn als die voor een RET!)
RETI, RETN		Return na interrupt routines. Vergeet maar.
RL		Roteer links: zoals een shift maar hier is de Carry-flag inbegrepen, deze wordt beschouwd als de achtste bit.
RLA		Roteer accumulator links. Zoals RL A maar met ander effect op de flags.
RLC		Is niet hetzelfde als RL C! Roteer links, maar stop bit 7 in Carry *én* in bit Ø.
RLCA		Zoals RLC A maar het effect op de flags is hetzelfde als bij RLA.
RLD		Helemaal niet wat u zou verwachten: roteer links decimaal. Wordt gebruikt bij binair gecodeerde decimale getallen: vergeet deze maar.
RR		Zoals RL maar naar links.
RRA		Zoals RLA.
RRC		Zoals RLC.
RRCA		Zoals RLCA.
RRD		Zoals RLD.
RST		Zoals CALL, maar slechts vanuit de adressen: Ø, 8, 1Ø, 18, 2Ø, 28, 3Ø, 38 (hex). Deze zitten bij de Spectrum alle in de ROM. RST Ø komt overeen met het tijdelijk uitschakelen van de stroom.
SBC	pag. 60	Trek af en houd hierbij rekening met de Carry-flag. Sla op in A of in HL.
SCF	pag. 105	Set Carry-flag (op 1).
SET	pag. 77	Set een bit (d.i. maak hem 1).
SLA	pag. 60	Schuif links rekenkundig: alle bits schuiven 1 plaats op; bit Ø wordt Ø.
SRA	pag. 60	Schuif rechts rekenkundig: verschuif de bits, copieer bit 7 in 6 *én* in 7.
SRL	pag. 60	Schuif rechts logisch: verschuif de bits en maak bit 7 nul.
SUB	pag. 50	Trek af zonder rekening te houden met de Carry-flag. Sla op in A. (Er bestaat geen SUB HL, r-commando, indien u er één nodig hebt, moet u de Carry-

flag resetten en SBC gebruiken.)

XOR	pag. 56	Exclusieve OR op elke bit. Sla op in A.

5 Zero en Carry-flags

Deze appendix toont hoe de instructies de Carry en de Zero-flags beïnvloeden. De symbolen hebben de volgende betekenis:

- – onveranderd
- * beïnvloed door het resultaat van de operatie
- 1 wordt op 1 geset
- Ø wordt op Ø gereset
- + wordt op 1 geset indien A = (HL), en in het andere geval wordt op Ø gereset.

Instructie	C	Z	Instructie	C	Z	Instructie	C	Z
ADC	*	*	IND	–	*	RET	–	–
ADD (8-bit)	*	*	INDR	–	1	RETI	–	–
ADD (16-bit)	*	–	INI	–	*	RETN	–	–
AND	Ø	*	INIR	–	1	RL	*	*
BIT	–	*	JP	–	–	RLA	*	–
CALL	*	*	JR	–	–	RLC	*	*
CCF	*	–	LD A, I	–	*	RLCA	*	–
CP	*	*	LD A, R	–	*	RLD	–	*
CPD	–	+	LD (all others)	–	–	RR	*	*
CPDR	–	+	LDD	–	–	RRA	*	–
CPI	–	+	LDDR	–	–	RRC	*	*
CPIR	–	+	LDI	–	–	RRCA	*	–
CPL	–	–	LDIR	–	–	RRD	–	*
DAA	*	*	NEG	*	*	RST	–	–
DEC	–	–	NOP	–	–	SBC	*	*
DI	–	–	OR	Ø	*	SCF	1	–
DJNZ	–	–	OTDR	–	1	SET	–	–
EI	–	–	OTIR	–	1	SLA	*	*
EX	–	–	OUT	–	–	SRA	*	*
EXX	–	–	OUTD	–	*	SRL	*	*
HALT	–	–	OUTI	–	*	SUB	*	*
IM	–	–	POP	–	–	XOR	Ø	*
IN A (n)	–	–	PUSH	–	–			
IN r, (C)	¬	*	RES	–	–			

6 Z8Ø Opcodes

Dit is een volledige lijst van de Z8Ø opcodes. Alfabetisch gerangschikt. In deze lijst staat het symbool n voor een willekeurig getal van één byte; nn voor een willekeurig getal van 2 bytes en stelt d de 1-byte verplaatsing voor in 2-complement notatie. Denk eraan dat alle 2-byte getallen worden gecodeerd door de junior byte te laten voorafgaan aan de senior byte.

Voorbeelden:

LD BC, nn heeft opcode Ø1 nn, dus wordt LD BC, 732F: Ø1 2F 73.

LD A, (IY + d) ,, ,, FD 7E d, ,, ,, LD A (IY + Ø7): FD 7E Ø7.

De tabel van de opcodes steunt op een tabel die door Zilog Inc. werd gepubliceert. In het Sinclair *Handboek* vindt u in appendix A een numerieke listing per opcode: merk op dat we daar voor de mnemonics kleine letters hebben gebruikt.

ADC A, (HL)	8E
ADC A, (IX + d)	DD8Ed
ADC A, (IY + d)	FD8Ed
ADC A, A	8F
ADC A, B	88
ADC A, C	89
ADC A, D	8A
ADC A, E	8B
ADC A, H	8C
ADC A, L	8D
ADC A, n	CEn
ADC HL, BC	ED4A
ADC HL, DE	ED5A
ADC HL, HL	ED6A
ADC HL, SP	ED7A
ADD A, (HL)	86
ADD A, (IX + d)	DD86d
ADD A, (IY + d)	FD86d
ADD A, A	87
ADD A, B	80
ADD A, C	81
ADD A,D	82
ADD A, E	83
ADD A, H	84
ADD A, L	85
ADD A, n	C6n
ADD HL, BC	09
ADD HL, DE	19
ADD HL, HL	29
ADD HL, SP	39
ADD IX, BC	DD09
ADD IX, DE	DD19
ADD IX, IX	DD29
ADD IX, SP	DD39
ADD IY, BC	FD09
ADD IY, DE	FD19
ADD IY, IY	FD29
ADD IY, SP	FD39
AND (HL)	A6
AND (IX + d)	DDA6d
AND (IY + d)	FDA6d
AND A	A7
AND B	A0
AND C	A1
AND D	A2
AND E	A3
AND H	A4
AND L	A5
AND n	E6n
BIT 0, (HL)	CB46
BIT 0, (IX + d)	DDCBd46
BIT 0, (IY + d)	FDCBd46
BIT 0, A	CB47
BIT 0, B	CB40
BIT 0, C	CB41
BIT 0, D	CB42
BIT 0, E	CB43
BIT 0, H	CB44
BIT 0, L	CB45
BIT 1, (HL)	CB4E
BIT 1, (IX + d)	DDCBd4E
BIT 1, (IY + d)	FDCBd4E
BIT 1, A	CB4F
BIT 1, B	CB48
BIT 1, C	CB49
BIT 1, D	CB4A
BIT 1, E	CB4B
BIT 1, H	CB4C
BIT 1, L	CB4D
BIT 2, (HL)	CB56
BIT 2, (IX + d)	DDCBd56
BIT 2, (IY + d)	FDCBd56
BIT 2, A	CB57
BIT 2, B	CB50

BIT 2, C	CB51
BIT 2, D	CB52
BIT 2, E	CB53
BIT 2, H	CB54
BIT 2, L	CB55
BIT 3, (HL)	CB5E
BIT 3, (IX + d)	DDCBd5E
BIT 3, (IY + d)	FDCBd5E
BIT 3, A	CB5F
BIT 3, B	CB58
BIT 3, C	CB59
BIT 3, D	CB5A
BIT 3, E	CB5B
BIT 3, H	CB5C
BIT 3, L	CB5D
BIT 4, (HL)	CB66
BIT 4, (IX + d)	DDCBd66
BIT 4, (IY + d)	FDCBd66
BIT 4, A	CB67
BIT 4, B	CB60
BIT 4, C	CB61
BIT 4, D	CB62
BIT 4, E	CB63
BIT 4, H	CB64
BIT 4, L	CB65
BIT 5, (HL)	CB6E
BIT 5, (IX + d)	DDCBd6E
BIT 5, (IY + d)	FDCBd6E
BIT 5, A	CB6F
BIT 5, B	CB68
BIT 5, C	CB69
BIT 5, D	CB6A
BIT 5, E	CB6B
BIT 5, H	CB6C
BIT 5, L	CB6D
BIT 6, (HL)	CB76
BIT 6, (IX + d)	DDCBd76
BIT 6, (IY + d)	FDCBd76
BIT 6, A	CB77
BIT 6, B	CB70
BIT 6, C	CB71
BIT 6, D	CB72
BIT 6, E	CB73
BIT 6, H	CB74
BIT 6, L	CB75
BIT 7, (HL)	CB7E
BIT 7, (IX + d)	DDCBd7E
BIT 7, (IY + d)	FDCBd7E
BIT 7, A	CB7F
BIT 7, B	CB78
BIT 7, C	CB79
BIT 7, D	CB7A
BIT 7, E	CB7B
BIT 7, H	CB7C
BIT 7, L	CB7D
CALL C, nn	DCnn
CALL M, nn	FCnn
CALL NC, nn	D4nn
CALL nn	CDnn
CALL NZ, nn	C4nn
CALL P, nn	F4nn
CALL PE, nn	ECnn
CALL PO, nn	E4nn
CALL Z, nn	CCnn
CCF	3F
CP (HL)	BE
CP (IX + d)	DDBEd
CP (IY + d)	FDBEd
CPA	BF
CP B	B8
CP C	B9
CP D	BA
CP E	BB
CP H	BC
CP L	BD
CP n	FEn
CPD	EDA9
CPDR	EDB9
CPI	EDA1
CPIR	EDB1
CPL	2F
DAA	27
DEC (HL)	35
DEC (IX + d)	DD35d
DEC (IY + d)	FD35d
DEC A	3D
DEC B	05
DEC BC	0B
DEC C	0D
DEC D	15
DEC DE	1B
DEC E	1D
DEC H	25
DEC HL	2B
DEC IX	DD2B
DEC IY	FD2B
DEC L	2D
DEC SP	3B
DI	F3
DJNZ, d	10d
EI	FB
EX (SP), HL	E3
EX (SP), IX	DDE3
EX (SP), IY	FDE3
EX AF, AF'	08
EX DE, HL	EB
EXX	D9
HALT	76
IM 0	ED46
IM 1	ED56
IM 2	ED5E
IN A, (C)	ED78
IN A, (n)	DBn
IN B, (C)	ED40
IN C, (C)	ED48
IN D, (C)	ED50
IN E, (C)	ED58
IN H, (C)	ED60
IN L, (C)	ED68
INC (HL)	34
INC (IX + d)	DD34d
INC (IY + d)	FD34d
INC A	3C
INC B	04
INC BC	03
INC C	0C
INC D	14
INC DE	13
INC E	1C
INC H	24
INC HL	23
INC IX	DD23
INC IY	FD23
INC L	2C
INC SP	33
IND	EDAA
INDR	EDBA
INI	EDA2
INIR	EDB2
JP (HL)	E9

JP (IX)	DDE9
JP (IY)	FDE9
JP C, nn	DAnn
JP M, nn	FAnn
JP NC, nn	D2nn
JP nn	C3nn
JP NZ, nn	C2nn
JP P, nn	F2nn
JP PE, nn	EAnn
JP PO, nn	E2nn
JP Z, nn	CAnn
JR C, d	38d
JR, d	18d
JR NC, d	30d
JR NZ, d	20d
JR Z, d	28d
LD (BC), A	02
LD (DE), A	12
LD (HL), A	77
LD (HL), B	70
LD (HL), C	71
LD (HL), D	72
LD (HL), E	73
LD (HL), H	74
LD (HL), L	75
LD (HL), n	36n
LD (IX + d), A	DD77d
LD (IX + d), B	DD70d
LD (IX + d), C	DD71d
LD (IX + d), D	DD72d
LD (IX + d), E	DD73d
LD (IX + d), H	DD74d
LD (IX + d), L	DD75d
LD (IX + d), n	DD36dn
LD (IY + d), A	FD77d
LD (IY + d), B	FD70d
LD (IY + d), C	FD71d
LD (IY + d), D	FD72d
LD (IY + d), E	FD73d
LD (IY + d), H	FD74d
LD (IY + d), L	FD75d
LD (IY + d), n	FD36dn
LD (nn), A	32nn
LD (nn), BC	ED43nn
LD (nn), DE	ED53nn
LD (nn), HL	22nn
LD (nn), IX	DD22nn
LD (nn), IY	FD22nn
LD (nn), SP	ED73nn
LD A, (BC)	0A
LD A, (DE)	1A
LD A, (HL)	7E
LD A, (IX + d)	DD7Ed
LD A, (IY + d)	FD7Ed
LD A, (nn)	3Ann
LD A, A	7F
LD A, B	78
LD A, C	79
LD A, D	7A
LD A, E	7B
LD A, H	7C
LD A, I	ED57
LD A, L	7D
LD A, n	3En
LD B, (HL)	46
LD B, (IX + d)	DD46d
LD B, (IY + d)	FD46d
LD B, A	47
LD B, B	40
LD B, C	41
LD B, D	42
LD B, E	43
LD B, H	44
LD B, L	45
LD B, n	06n
LD BC, (nn)	ED4Bnn
LD BC, nn	01nn
LD C, (HL)	4E
LD C, (IX + d)	DD4Ed
LD C, (IY + d)	FD4Ed
LD C, A	4F
LD C, B	48
LD C, C	49
LD C, D	4A
LD C, E	4B
LD C, H	4C
LD C, L	4D
LD C, n	0En
LD D, (HL)	56
LD D, (IX + d)	DD56d
LD D, (IY + d)	FD56d
LD D, A	57
LD D, B	50
LD D, C	51
LD D, D	52
LD D, E	53
LD D, H	54
LD D, L	55
LD D, n	16n
LD DE, (nn)	ED5Bnn
LD DE, nn	11nn
LD E, (HL)	5E
LD E, (IX + d)	DD5Ed
LD E, (IY + d)	FD5Ed
LD E, A	5F
LD E, B	58
LD E, C	59
LD E, D	5A
LD E, E	5B
LD E, H	5C
LD E, L	5D
LD E, n	1En
LD H, (HL)	66
LD H, (IX + d)	DD66d
LD H, (IY + d)	FD66d
LD H, A	67
LD H, B	60
LD H, C	61
LD H, D	62
LD H, E	63
LD H, H	64
LD H, L	65
LD H, n	26n
LD HL, (nn)	2Ann
LD HL, nn	21nn
LD I, A	ED47
LD IX, (nn)	DD2Ann
LD IX, nn	DD21nn
LD IY, (nn)	FD2Ann
LD IY, nn	FD21nn
LD L, (HL)	6E
LD L, (IX + d)	DD6Ed
LD L, (IY + d)	FD6Ed
LD L, A	6F
LD L, B	68
LD L, C	69
LD L, D	6A
LD L, E	6B
LD L, H	6C
LD L, L	6D

LD L, n	2En
LD SP, (nn)	ED7Bnn
LD SP, HL	F9
LD SP, IX	DDF9
LD SP, IY	FDF9
LD SP, nn	31nn
LDD	EDA8
LDDR	EDB8
LDI	EDA0
LDIR	EDB0
NEG	ED44
NOP	00
OR (HL)	B6
OR (IX + d)	DDB6d
OR (IY + d)	FDB6d
OR A	B7
OR B	B0
OR C	B1
OR D	B2
OR E	B3
OR H	B4
OR L	B5
OR n	F6n
OTDR	EDBB
OTIR	EDB3
OUT (C), A	ED79
OUT (C), B	ED41
OUT (C), C	ED49
OUT (C), D	ED51
OUT (C), E	ED59
OUT (C), H	ED61
OUT (C), L	ED69
OUT (n), A	D3n
OUTD	EDAB
OUTI	EDA3
POPAF	F1
POP BC	C1
POP DE	D1
POP HL	E1
POP IX	DDE1
POP IY	FDE1
PUSH AF	F5
PUSH BC	C5
PUSH DE	D5
PUSH HL	E5
PUSH IX	DDE5
PUSH IY	FDE5
RES 0, (HL)	CB86
RES 0, (IX + d)	DDCBd86
RES 0, (IY + d)	FDCBd86
RES 0, A	CB87
RES 0, B	CB80
RES 0, C	CB81
RES 0, D	CB82
RES 0, E	CB83
RES 0, H	CB84
RES 0, L	CB85
RES 1, (HL)	CB8E
RES 1, (IX + d)	DDCBd8E
RES 1, (IY + d)	FDCBd8E
RES 1, A	CB8F
RES 1, B	CB88
RES 1, C	CB89
RES 1, D	CB8A
RES 1, E	CB8B
RES 1, H	CB8C
RES 1, L	CB8D
RES 2, (HL)	CB96
RES 2, (IX + d)	DDCBd96
RES 2, (IY + d)	FDCBd96
RES 2, A	CB97
RES 2, B	CB90
RES 2, C	CB91
RES 2, D	CB92
RES 2, E	CB93
RES 2, H	CB94
RES 2, L	CB95
RES 3, (HL)	CB9E
RES 3, (IX + d)	DDCBd9E
RES 3, (IY + d)	FDCBd9E
RES 3, A	CB9F
RES 3, B	CB98
RES 3, C	CB99
RES 3, D	CB9A
RES 3, E	CB9B
RES 3, H	CB9C
RES 3, L	CB9D
RES 4, (HL)	CBA6
RES 4, (IX + d)	DDCBdA6
RES 4, (IY + d)	FDCBdA6
RES 4, A	CBA7
RES 4, B	CBA0
RES 4, C	CBA1
RES 4, D	CBA2
RES 4, E	CBA3
RES 4, H	CBA4
RES 4, L	CBA5
RES 5, (HL)	CBAE
RES 5, (IX + d)	DDCBdAE
RES 5, (IY + d)	FDCBdAE
RES 5, A	CBAF
RES 5, B	CBA8
RES 5, C	CBA9
RES 5, D	CBAA
RES 5, E	CBAB
RES 5, H	CBAC
RES 5, L	CBAD
RES 6, (HL)	CBB6
RES 6, (IX + d)	DDCBdB6
RES 6, (IY + d)	FDCBdB6
RES 6, A	CBB7
RES 6, B	CBB0
RES 6, C	CBB1
RES 6, D	CBB2
RES 6, E	CBB3
RES 6, H	CBB4
RES 6, L	CBB5
RES 7, (HL)	CBBE
RES 7, (IX + d)	DDCBdBE
RES 7, (IY + d)	FDCBdBE
RES 7, A	CBBF
RES 7, B	CBB8
RES 7, C	CBB9
RES 7, D	CBBA
RES 7, E	CBBB
RES 7, H	CBBC
RES 7, L	CBBD
RET	C9
RET C	D8
RET M	F8
RET NC	D0
RET NZ	C0
RET P	F0
RET PE	E8
RET PO	E0
RET Z	C8
RETI	ED4D
RETN	ED45
RL (HL)	CB16
RL (IX + d)	DDCBd16

RL (IY + d)	FDCBd16
RL A	CB17
RL B	CB10
RL C	CB11
RL D	CB12
RL E	CB13
RL H	CB14
RL L	CB15
RLA	17
RLC (HL)	CB06
RLC (IX + d)	DDCBd06
RLC (IY + d)	FDCBd06
RLC A	CB07
RLC B	CB00
RLC C	CB01
RLC D	CB02
RLC E	CB03
RLC H	CB04
RLC L	CB05
RLCA	07
RLD	ED6F
RR (HL)	CB1E
RR (IX + d)	DDCBd1E
RR (IY + d)	FDCBd1E
RR A	CB1F
RR B	CB18
RR C	CB19
RR D	CB1A
RR E	CB1B
RR H	CB1C
RR L	CB1D
RRA	1F
RRC (HL)	CB0E
RRC (IX + d)	DDCBd0E
RRC (IY + d)	FDCBd0E
RRC A	CB0F
RRC B	CB08
RRC C	CB09
RRC D	CB0A
RRC E	CB0B
RRC H	CB0C
RRC L	CB0D
RRC A	0F
RRD	ED67
RST 0	C7
RST 10H	D7
RST 18H	DF
RST 20H	E7
RST 28H	EF
RST 30H	F7
RST 38H	FF
RST 8	CF
SBC A, (HL)	9E
SBC A, (IX + d)	DD9Ed
SBC A, (IY + d)	FD9Ed
SBC A, A	9F
SBC A, B	98
SBC A, C	99
SBC A, D	9A
SBC A, E	9B
SBC A, H	9C
SBC A, L	9D
SBC A, n	DEn
SBC HL, BC	ED42
SBC HL, DE	ED52
SBC HL, HL	ED62
SBC HL, SP	ED72
SCF	37
SET 0, (HL)	CBC6
SET 0, (IX + d)	DDCBdC6
SET 0, (IY + d)	FDCBdC6
SET 0, A	CBC7
SET 0, B	CBC0
SET 0, C	CBC1
SET 0, D	CBC2
SET 0, E	CBC3
SET 0, H	CBC4
SET 0, L	CBC5
SET 1, (HL)	CBCE
SET 1, (IX + d)	DDCBdCE
SET 1, (IY + d)	FDCBdCE
SET 1, A	CBCF
SET 1, B	CBC8
SET 1, C	CBC9
SET 1, D	CBCA
SET 1, E	CBCB
SET 1, H	CBCC
SET 1, L	CBCD
SET 2, (HL)	CBD6
SET 2, (IX + d)	DDCBdD6
SET 2, (IY + d)	FDCBdD6
SET 2, A	CBD7
SET 2, B	CBD0
SET 2, C	CBD1
SET 2, D	CBD2
SET 2, E	CBD3
SET 2, H	CBD4
SET 2, L	CBD5
SET 3, (HL)	CBDE
SET 3, (IX + d)	DDCBdDE
SET 3, (IY + d)	FDCBdDE
SET 3, A	CBDF
SET 3, B	CBD8
SET 3, C	CBD9
SET 3, D	CBDA
SET 3, E	CBDB
SET 3, H	CBDC
SET 3, L	CBDD
SET 4, (HL)	CBE6
SET 4,(IX + d)	DDCBdE6
SET 4, (IY + d)	FDCBdE6
SET 4, A	CBE7
SET 4, B	CBE0
SET 4, C	CBE1
SET 4, D	CBE2
SET 4, E	CBE3
SET 4, H	CBE4
SET 4, L	CBE5
SET 5, (HL)	CBEE
SET 5, (IX + d)	DDCBdEE
SET 5, (IY + d)	FDCBdEE
SET 5, A	CBEF
SET 5, B	CBE8
SET 5, C	CBE9
SET 5, D	CBEA
SET 5, E	CBEB
SET 5, H	CBEC
SET 5, L	CBED
SET 6, (HL)	CBF6
SET 6, (IX + d)	DDCBdF6
SET 6, (IY + d)	FDCBdF6
SET 6, A	CBF7
SET 6, B	CBF0
SET 6, C	CBF1
SET 6, D	CBF2
SET 6, E	CBF3
SET 6, H	CBF4
SET 6, L	CBF5
SET 7, (HL)	CBFE
SET 7, (IX + d)	DDCBdFE

SET 7, (IY + d)	FDCBdFE
SET 7, A	CBFF
SET 7, B	CBF8
SET 7, C	CBF9
SET 7, D	CBFA
SET 7, E	CBFB
SET 7, H	CBFC
SET 7, L	CBFD
SLA (HL)	CB26
SLA (IX + d)	DDCBd26
SLA (IY + d)	FDCBd26
SLA A	CB27
SLA B	CB20
SLA C	CB21
SLA D	CB22
SLA E	CB23
SLA H	CB24
SLA L	CB25
SRA (HL)	CB2E
SRA (IX + d)	DDCBd2E
SRA (IY + d)	FDCBd2E
SRA A	CB2F
SRA B	CB28
SRA C	CB29
SRA D	CB2A
SRA E	CB2B
SRA H	CB2C
SRA L	CB2D
SRL (HL)	CB3E
SRL (IX + d)	DDCBd3E
SRL (IY + d)	FDCBd3E
SRL A	CB3F
SRL B	CB38
SRL C	CB39
SRL D	CB3A
SRL E	CB3B
SRL H	CB3C
SRL L	CB3D
SUB (HL)	96
SUB (IX + d)	DD96d
SUB (IY + d)	FD96d
SUB A	97
SUB B	90
SUB C	91
SUB D	92
SUB E	93
SUB H	94
SUB L	95
SUB n	D6n
XOR (HL)	AE
XOR (IX + d)	DDAEd
XOR (IY + d)	FDAEd
XOR A	AF
XOR B	A8
XOR C	A9
XOR D	AA
XOR E	AB
XOR H	AC
XOR L	AD
XOR n	EEn

7 HELPA

Helpa is een veelzijdig gebruiksprogramma dat in BASIC werd geschreven zodat het gemakkelijk kan worden aangepast. Het werd geschreven om u te helpen bij het editten, het loaden en het runnen van machinecode. Helpa is het acroniem van Hex Editor, Loader en Partial Assembler.

1. Typ het programma in en SAVE het als volgt:
   SAVE "helpa" LINE 5

2. RUN. Het programma vraagt hoeveel geheugenruimte moet worden gereserveerd en het stelt de vraag: "Wenst u een niet-standaard geheugen?", waarna u een input moet geven. Alles wat u invoert (behalve ENTER) wordt beschouwd als vragen naar het startadres van de machinecode dat dan verschillend is van de standaard 32ØØØ die in heel dit boek werd gebruikt. Daarna wordt het geheugen geCLEARd, de RAMTOP wordt gereset en de nieuwe RAMTOP-waarde en het startadres van de machinecode worden voor controle uitgeprint.

3. Dan informeert het programma naar het aantal databytes die aan het programmagebied voorafgaan. Deze moeten achteraf gePOKEt worden (misschien geeft u er de voorkeur aan om een routine toe te voegen die dit doet).

4. Nu worden deze waarde en het actuele startadres dat voor de machinecode moet worden gebruikt, uitgeprint.

5. Vervolgens vraagt het programma naar de HEX CODE en geeft het een rode cursor, een flitsend [>] -symbool.

6. Nu kunt u de machinecode in hex intypen (en manipuleren indien u de stuurcommando's gebruikt die hieronder worden opgesomd.) Zodra een codegroep is ingetypt, verwijdert het programma alle blanco spaties (dus u kunt spaties gebruiken indien u dit wenst), splitst deze groep in twee-karakter codes en print ze uit in een display met 1Ø kolommen. Aan het eind van elke groep wordt tenslotte een ** delimiter toegevoegd; de cursor wordt verplaatst naar het einde van de net ingevoerde code.

De input "□a1□e23□□4" wordt bijvoorbeeld als volgt uitgeprint:

a1 e2 34 ** [>]

(en op dezelfde manier gebeurt dit voor "a1e234" of "a1□e2□34").

7. Nu mag de volgende groep codes worden ingevoerd, deze groep zal automatisch onmiddellijk achter de cursorpositie worden aangehecht. De **-delimiters werden slechts voor het gemak van de gebruiker toegevoegd, ze worden trouwens genegeerd wanneer de code boven RAMTOP wordt geladen. De groepen *hoeven* dus niet overeen te komen met Z8Ø-instructiegroepen, hoewel dit het nakijken wel vergemakkelijkt.

8. Daarenboven kan men "stuurinstructies" invoeren. Deze moeten het *eerste* symbool in een groep zijn; achtereenvolgende karakters worden gewoonlijk genegeerd, behalve in enkele gevallen die we dadelijk zullen beschrijven.

De stuurcommando's zijn:

g	(Go)	Run de machinecode routine.
l	(Load)	Laad de machinecode routine boven RAMTOP.
m	(Move)	Verplaats de cursor voorwaarts.
n	(Negative)	Verplaats de cursor achterwaarts.
p	(Print)	Print de lopende hex listing uit.
r	(Relative)	Wordt gebruikt om relatieve sprongen automatisch te berekenen.
s	(Save)	Save het programma.
x	(eXcise)	Schrap uit de listing.

We geven een meer gedetailleerde beschrijving van deze commando's in een volgorde die hiervoor geschikter is:

m, n Het commando ma, waarin a een getal voorstelt, verplaatst de cursor a spaties voorwaarts, na verplaats het a spaties achterwaarts. Er werd een bescherming tegen het van de listing "aflopen" ingebouwd. Indien a wordt weggelaten dan werkt het commando alsof a op 1 werd geset.

x Het commando xa, waarin a een getal $\geqslant 0$ zal de volgende a paren karakters (de **-delimiters inbegrepen) na de cursor schrappen. De display wordt niet beïnvloed zolang p niet wordt ingeduwd. Indien a wordt weggelaten dan wordt verondersteld dat a op 1 werd geset.

Om tekst *tussen te voegen* hoeft u slechts de cursor te verplaatsen tot vlak voor de gewenste positie en dan in te voeren. Alles wat erna voorkomt wordt opgeschoven zodat er plaats vrijkomt, maar de nieuwe tekst verschijnt pas op de display nadat p wordt ingedrukt.

p Print de lopende tekst uit en verplaatst de cursor naar boven.

s SAVEt het programma, de in omloop zijnde hex listing inbegrepen. Er bestaat een optie die het mogelijk maakt dat u het programma noemt zoals u wenst. Om een "gesaved" programma te runnen, SAVEt u het *alvorens* u "l" en "g" intypt. Daarna LOAD in BASIC, typ GO TO 200 (*niet* RUN!!) en vervolgens "l" en "g".

l Laadt de machinecode, die werd ontdaan van overbodige **-delimiters, boven RAMTOP te beginnen bij de RAMTOP-waarde die u hebt ingesteld en hangt het eindcommando RET er aan vast.

g Dit commando laat de routine draaien. De sturing keert terug

naar HELPA (behalve bij een "crash"!).

r Dit is erg nuttig om *relatieve* sprongen veel eenvoudiger te maken. Relatieve sprongen met de hand berekenen is vervelend omdat de verplaatsingen in 2-complement hex moeten worden uitgewerkt. Maar in een programma zijn ze zeer praktisch omdat ze "portable" zijn, d.w.z. de programma's kunnen om het even waar in het RAM-geheugen worden geplaatst. Het werkt als volgt:

(a) Voer de machinecode routine in en zet alle relatieve spronggroottes op ØØ.

(b) Om de correcte waarde voor een gegeven sprong te verkrijgen, plaatst u de cursor vlak voor de ØØ en schrapt u deze ØØ d.m.v. "x". Nu voert u rn in, waarbij n de (decimale) positie is van de bestemmingsbyte van de sprong. Dit getal wordt als volgt van het scherm afgelezen: nummer de rijen en de kolommen van de hex code array te beginnen bij Ø:

```
  Ø 1 2 3 4 5 6 7 8 9
Ø
1
2
3
.
.
.
```

en maak n gelijk aan xy voor rij x, kolom y. (Dus, voor de byte in rij 17, kolom 5 wordt dat r175). Het aflezen van de getallen is erg gemakkelijk omdat er 1Ø kolommen zijn. (U kunt het programma zo wijzigen dat de cursor naar de bestemming wordt verplaatst en dat hieruit de sprong wordt gevonden. In de praktijk is dit trager omdat u dan veel schuifwerk met de cursor moet uitvoeren.)

(c) Nu drukt u "p" in voor een nieuwe display waarin de juiste spronggrootte voorkomt.

(d) Plaats de cursor vóór de volgende relatieve spronggrootte ØØ en herhaal de werkwijze.

(e) Bemerk dat het programma alle **-delimiters automatisch bijstelt en dat het de benodigde 2-complement code geeft: bovendien zal het programma meedelen dat een eventuele sprong het toelaatbare bereik overschrijdt.

(f) Dit klinkt allemaal misschien nogal ingewikkeld, daarom geven we een voorbeeld waarbij we de kolomscroll routine uit hoofdstuk 14 gebruiken. Voer achtereenvolgens de groepen hex codes in; het resultaat zal zijn:

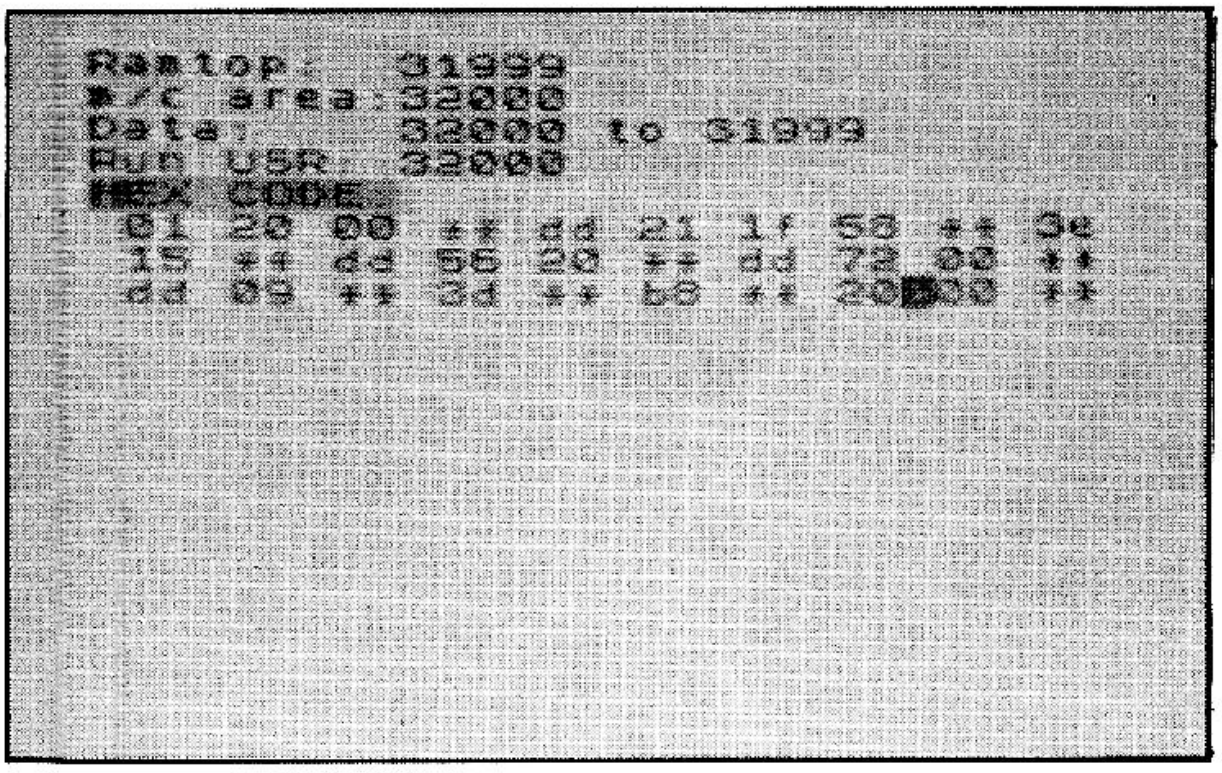
```
Ramtop:        31999
M/C area:      32000
Data:          32000 to 31999
Run USR        32000
HEX CODE:

01 20 00 ** dd 21 1f 58 ** 3e
15 ** dd 56 20 ** dd 72 00 **
dd 09 ** 3d ** b8 ** 20 00 **
>
```

De onderstreepte ØØ is de plaats van de te berekenen relatieve sprong. (Het datagebied maakt een idiote indruk – u kunt het programma aanpassen zodat het "Data: geen" uitprint indien het aantal databytes Ø is.)

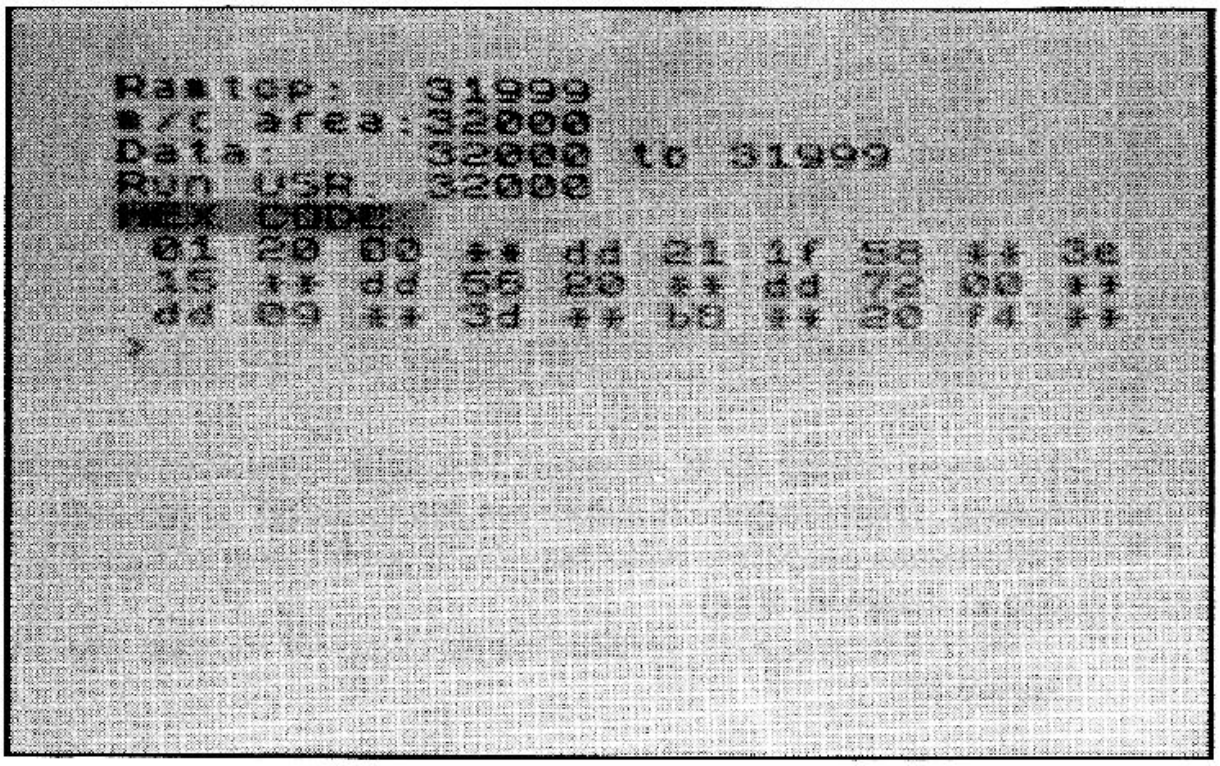
Door n te gebruiken verplaatsen we de cursor tot vlak voor de laatste ØØ:

```
dd 09 ** 3d ** b8 ** 20 > 00 **
```

en dan schrappen we ØØ d.m.v. "x". De instructie is hier JRNZ *lus*, en de lus begint bij code dd op de tweede hex regel. Dit is op rij 1, kolom 2 (denk eraan dat beide met Ø beginnen) dus moet u "r12" invoeren. Doe dit maar eens.



Figuur A7.1 HELPA is klaar om een relatieve sprong te berekenen.



Figuur A7.2 Het sprongadres, f4, werd ingeschreven.

Na een tijdje wordt het scherm blanco en wordt een nieuwe listing uitgeprint waarin de ØØ vervangen is door f4, de juiste relatieve spronggrootte.

Hier volgt de listing van HELPA:

```
  5  REM helpa © 1982 Ian Stewart & Robin Jones
  7  POKE 23609, 50
 10  INPUT "wenst u een niet-standaard geheugen?"; a$
 20  IF a$ < > " " THEN INPUT "START van M/C zone?"; rt
 30  IF a$ = " " THEN LET rt = 32000
 40  CLEAR rt - 1
 50  LET rt = PEEK 23730 + 256 * PEEK 23731 + 1
 60  PRINT "Ramtop: □ □"; rt - 1
 70  PRINT "m/c zone: □"; rt
 80  INPUT "Aantal databytes?"; d
 90  PRINT "Data: □ □ □ □"; rt; "□ tot □"; rt + d - 1
100  PRINT "Run USR: □"; rt + d
110  PRINT PAPER 6; "HEX CODE:"
120  LET ci = 0
130  LET h$ = " "
140  GO SUB 400
150  DIM f(20)
151  LET f(1) = 700
152  LET f(6) = 800
153  LET f(7) = 900
154  LET f(8) = 1000
155  LET f(10) = 1100
156  LET f(11) = 1200
157  LET f(12) = 1300
158  LET f(13) = 1400
159  LET f(14) = 1500
160  LET f(18) = 1600
200  INPUT i$
210  LET a = 0
220  LET a = a + 1
230  IF a > LEN i$ THEN GO TO 300
```

```
240  IF i$(a) < > "□" THEN GO TO 220
250  LET i$ = i$(TO a − 1) + i$(a + 1 TO)
260  GO TO 230

300  IF CODE i$(1) > = 103 THEN GO TO 500
310  LET i$ = i$ + "**"
320  LET h$ = h$(TO 2 * ci) + i$ + h$(2 * ci + 1 TO)
330  GO SUB 450
340  GO SUB 600
350  GO SUB 400
360  GO TO 200

400  REM print cursor
410  PRINT AT 5 + INT(ci/10), 3 * (ci − 10 * INT(ci/10) );
     FLASH 1; INK 2; ">";
420  RETURN

450  REM unprint cursor
460  PRINT AT 5 + INT(ci/10), 3 * (ci − 10 * INT(ci/10) ); "□";
470  RETURN

500  REM toetsenbord routines
510  GO SUB f(CODE i$(1) − 102) )
520  GO TO 200

600  REM print toegevoegde tekst
610  FOR j = 1 TO LEN i$/2
620  PRINT i$(2 * j − 1 TO 2 * j) + "□";
630  LET ci = ci + 1
640  IF ci = 10 * INT(ci/10) THEN PRINT "□ □";
650  NEXT j
660  RETURN

700  REM go
```

```
710  CLS
720  LET y = USR(rt + d)
730  RETURN

800  REM laad boven ramtop
810  LET h$ = h$ + "c9"
820  LET j = rt + d - 1
830  LET i = -1
840  LET j = j + 1
850  LET i = i + 2
860  IF i > LEN h$ THEN RETURN
870  IF h$(i) = "*" THEN GO TO 850
880  POKE j, 16 * (CODE h$(i) - 48 - 39 * (h$(i) > "9") )
     + CODE h$(i + 1) - 48 - 39 * (h$(i + 1) > "9")
890  GO TO 840
900  REM verplaats cursor naar rechts
910  GO SUB 450
920  IF LEN i$ = 1 THEN LET cm = 1
930  IF LEN i$ > 1 THEN LET cm = VAL i$(2 TO)
940  LET ci = ci + cm
950  IF ci > LEN h$/2 THEN LET ci = LEN h$/2
960  GO SUB 400
970  RETURN

1000 REM verplaats cursor naar links
1010 GO SUB 450
1020 IF LEN i$ = 1 THEN LET cm = 1
1030 IF LEN i$ > 1 THEN LET cm = VAL i$(2 TO)
1040 LET ci = ci - cm
1050 IF ci < 0 THEN LET ci = 0
1060 GO SUB 400
1070 RETURN

1100 REM print
```

```
1110  PRINT AT 5, 0;
1120  FOR r = 1 TO 17: PRINT " [32 spaties]; : NEXT r
1130  PRINT AT 5, 0; "□";
1140  LET ci = 0
1150  FOR j = 1 TO LEN h$/2
1160  PRINT h$(2 * j - 1 TO 2 * j) + "□";
1170  LET ci = ci + 1
1180  IF ci = 10 * INT(ci/10) THEN PRINT "□ □";
1185  NEXT j
1190  GO SUB 400
1195  RETURN

1300  REM relatieve sprongen
1310  LET jci = VAL i$(2 TO)
1320  LET js = jci - ci - 1
1325  GO SUB 2000
1330  IF js > = -128 AND js < = 127 THEN GO TO 1355
1340  INPUT "Ongeldige grootte; ENTER om verder te gaan"; a$
1350  RETURN
1355  IF js < 0 THEN LET js = js + 256
1360  LET x1 = INT(js/16)
1365  LET x0 = js - 16 * x1
1367  LET x$ = CHR$(x1 + 48 + 39 * (x1 > 9) ) +
                                    CHR$(x0 + 48 + 39 * (x0 > 9) )
1370  LET h$ = h$(TO 2 * ci) + x$ + h$(2 * ci + 1 TO)
1375  LET ci = ci + 1
1380  GO SUB 1100
1390  RETURN

1400  REM save
1410  INPUT "SAVE naam? Bij verstek " "helpa" " "; n$
1420  IF n$ = " " THEN LET n$ = "helpa"
1430  POKE rt + d + LEN h$, 201
```

```
1440  SAVE n$ CODE rt, d + LEN h$ + 1
1450  PRINT "Om te herladen" ' ' "LOAD" " "; n$; " " "
                                                  □ CODE"
1460  RETURN

1600  REM schrap
1610  IF LEN i$ = 1 THEN LET k = 1
1620  IF LEN i$ > 1 THEN LET k = VAL i$(2 TO)
1630  LET h$ = h$(TO 2 * ci) + h$(2 * ci + 2 * k + 1 TO)
1640  RETURN

2000  REM plaats van sterretje
2010  IF js < 0 THEN LET w$ = h$(2 * jci + 1 TO 2 * ci)
2020  IF js > = 0 THEN LET w$ = h$(2 * ci + 1 TO 2 * jci)
2030  LET sc = 0
2040  FOR t = 1 TO LEN w$
2050  IF w$(t) = "*" THEN LET sc = sc + 1
2060  NEXT t
2070  IF js < 0 THEN LET js = js + sc/2
2080  IF js > 0 THEN LET js = js − sc/2
2090  RETURN
```

ISBN 90 6215 089 6
D 1984/1997/2
Stewart, MCode Spectrum

Stewart/Jones MACHINE CODE met de ZX SPECTRUM

Machine Code

met de

ZX SPECTRUM

MAARTEN KLUWER'S
INTERNATIONALE UITGEVERSONDERNEMING
Antwerpen - Apeldoorn