Z80 Assembly Language Programming for Students

Roger Hutty

8

CURSUS Z-80
ASSEMBLEERTAAL

CURSUS Z-80 ASSEMBLEERTAAL

Roger Hutty

ACADEMIC SERVICE

Published in the United Kingdom by the Macmillan Press
under the title: *Z80 Assembly Language Programming for Students*

1e druk januari 1983
2e druk december 1983
3e druk juni 1985

Vertaling en bewerking: drs. A.J.J. van Veen

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Hutty, Roger

Cursus Z-80 assembleertaal / Roger Hutty ; bew. en vert. [uit het Engels door] Nok van Veen ; [ill. Eddy Both]. - Den Haag : Academic Service. - Ill., tek.
Vert. van: Z80 assembly language programming for students. - Londen : Macmillan Press, 1981. - Met index.
ISBN 90-6233-090-8
SISO 365.3 SVS 7.12.3 UDC 681.3.06:800.92
Trefw. : programmeren / programmeertalen.

Uitgegeven door: Academic Service
Postbus 96996
2509 JJ Den Haag
Zetwerk: multitASk, Blaricum
Omslagontwerp: Eddy Both en JAM Gauw
Illustraties: Eddy Both
Druk: Krips Repro Meppel
Bindwerk: Meeuwis, Amsterdam
ISBN 90 6233 090 8

Voorwoord

Dit leerboek, dat de assembleertaal van de Z80 microprocessor behandelt, stelt de lezer in staat een professionele aanpak van het programmeren in assembleertaal te ontwikkelen. Voorwaarde is dan wel dat de in de tekst opgenomen vragen en de programmeeropdrachten consequent worden beantwoord respectievelijk uitgewerkt worden. Om goede programma's te kunnen schrijven is het niet voldoende de regels van een programmeertaal te kennen (dat is een vereiste). De keuze van de juiste programmeertechniek, de leesbaarheid en de logica van een programma zijn minstens even belangrijk, zo niet belangrijker, kortom: "There is an art to programming".

Dit boek is niet zomaar een studieboek, het is een leerboek. Het kan gebruikt worden als leerboek bij een programmeercursus, waarbij de nadruk zowel op 'klassikaal onderwijs' als op 'begeleide zelfstudie' kan liggen. Ook is het geschreven voor mensen die zich de kunst van het programmeren in assembleertaal geheel door zelfstudie willen eigen maken. Deze lezers zullen dan wel enige affiniteit met het onderwerp moeten bezitten, bijvoorbeeld doordat zij reeds iets afweten van de Z80 assembleertaal of van de Z80 microprocessor of van programmeren in het algemeen. Tenslotte zal dit boek ook voor de meer ervaren Z80 assembleertaalprogrammeur, bijvoorbeeld als naslagwerk, zijn nut kunnen hebben.

De leerstof is in zestien hoofdstukken verdeeld. Elk hoofdstuk vormt een min of meer op zichzelf staand stukje leerstof, zodat elk hoofdstuk in een betrekkelijk kort tijdsbestek onderwezen dan wel zelf bestudeerd kan worden. De opzet van de leerstof is zo, dat elk volgend hoofdstuk het bestuderen van de vorige hoofdstukken vereist.

Elk hoofdstuk is verdeeld in een aantal paragrafen, die elk één of twee onderdelen uit de Z80 assembleertaal behandelen. In elke paragraaf zijn een of meer opgaven opgenomen. De paragrafen zijn vrij kort gehouden, terwijl het beantwoorden van de vragen aanleiding is tot het doen van 'nieuwe' ontdekkingen. Hierdoor worden al vrij snel een aantal vaardigheden ontwikkeld. Er wordt niets

nagelaten om praktische problemen ter hand te nemen en reeds vanaf het begin 'goede' programmeergewoonten aan te kweken.

De indeling van de leerstof, de opgaven, de antwoorden van die opgaven en de programmeeropdrachten aan het einde van elk hoofdstuk zijn alle even belangrijk in het hele leerproces. De opgaven zijn zo gekozen dat een objectieve meting van kennis en inzicht mogelijk is. Bij sommige antwoorden op vragen worden nog enkele opmerkingen geplaatst, waardoor een positieve bijdrage aan de begripsvorming wordt geleverd. Aan het einde van elk hoofdstuk treft de lezer een programmeeropdracht aan. De bedoeling daarvan is een programma te coderen (te schrijven) en zo mogelijk het programma te testen en te draaien op een microcomputer. In zo'n programma wordt getracht zoveel mogelijk onderwerpen uit het desbetreffende hoofdstuk op te nemen.

Aangeraden wordt in ieder geval te proberen alle opgaven uit een hoofdstuk te beantwoorden en het programma uit de programmeeropdracht te coderen alvorens verder te gaan met het bestuderen van het volgende hoofdstuk. In een onderwijssituatie zullen veel studenten min of meer zelf het boek kunnen doorwerken. De aandacht van de docent kan zich dan richten op diegenen die wat meer moeite hebben.

De filosofie achter de opzet van dit boek is die van 'al doende leert men'. Veel oefenen is een goede manier om een programmeertaal te leren, hetgeen trouwens geldt voor het leren van elke 'vreemde taal'! Vaardigheid in het programmeren krijg je door veel programma's te schrijven.

Dit is een van de redenen waarom de opgaven en programmeeropdrachten zijn opgenomen. Vele vragen vereisen dat de antwoorden worden opgeschreven. Het is belangrijk dat ook daadwerkelijk te doen. Ongeduldige lezers, die snel verder willen met het volgende onderwerp, zullen de verleiding om deze vragen 'zo uit het hoofd' te beantwoorden moeten weerstaan.

Het modulair programmeren wordt van begin tot eind gestimuleerd; het is dan ook niet verwonderlijk dat het begrip subroutine reeds in hoofdstuk 3 aan de orde komt.
Reeds in een vroeg stadium wordt de lezer in staat gesteld programma's te schrijven die ook gedraaid kunnen worden.

Het valt aan te bevelen de programmeeropdracht, die aan het einde van elk hoofdstuk is opgenomen, zo uit te voeren dat het programma niet alleen geschreven wordt maar dat het ook getest en gedraaid wordt vóórdat aan het volgende hoofdstuk begonnen wordt. Is er niet direct een computer voorhanden, codeer dan in ieder

geval de programma-instructies. Het programma moet eigenlijk een goed draaiend programma zijn, dat tevens aan alle vooraf gestelde eisen behoort te voldoen. Niemand heeft iets aan een programma dat niet precies dat doet wat er van verlangd wordt. Elk hoofdstuk bevat slechts één programmeeropdracht. De lezer kan zich dus volledig op deze opdracht concentreren.

Van de programma's gaat een stimulerende werking uit en de programmeur wordt altijd 'visueel' beloond voor het schrijven van een goed werkend programma; alle programma's drukken namelijk iets op een beeldscherm af.

Natuurlijk is een ieder vrij andere programmeeropdrachten te geven of te kiezen; wel is het zo dat de opdrachten in dit boek zorgvuldig zijn samengesteld, opdat zoveel mogelijk aspecten van de Z80-assembleertaal aan bod komen.

De meeste onderwerpen worden niet in één hoofdstuk uitputtend behandeld maar komen in verschillende hoofdstukken aan de orde. Hierdoor kunnen in een vroeger stadium gerichtere programma's geschreven worden en wordt de lezer gestimuleerd geconcentreerd bezig te zijn. Door dit 'gespreid' behandelen van de onderwerpen krijgt de lezer meer gelegenheid detail voor detail in zich op te nemen en te verwerken.

Een probleem dat zich voordoet bij het geven van onderwijs in een programmeertaal aan een groep studenten is, dat de snelheid waarmee de leerstof wordt opgenomen van student tot student sterk verschilt. Een veel gehanteerde onderwijsvorm waarin dit soort cursussen gegeven worden is dan ook een vorm waarbij van de studenten een grote mate van zelfwerkzaamheid wordt verlangd. De docent ondersteunt dan de studenten die wat meer moeite met de stof hebben. Dit boek is met name voor deze wijze van lesgeven en studeren uitermate geschikt.

Het boek is niet voor een bepaald type Z80 microcomputer geschreven. Het kan gebruikt worden bij elk microcomputersysteem dat gebaseerd is op de Z80 microprocessor. De minimaal vereiste configuratie is een Z80 processor, een toetsenbord en een beeldscherm.

Mijn dank gaat uit naar de Leicester Polytechnic, alwaar ik in de gelegenheid werd gesteld bij het ontwikkelen van de programma's en de produktie van dit boek van de computer gebruik te maken. Ook wil ik Bob Reeve van de Wolverhampton Polytechnic bedanken voor het zorgvuldig doorlezen van het manuscript en voor de vele door hem voorgestelde verbeteringen. Ook bedank ik mijn vrouw, Susan, die mij op vele manieren heeft bijgestaan, in het bijzonder door het uittikken van de tekst met behulp van een tekstverwerker.

Tenslotte nog een woordje van dank aan Nicolas en Elisabeth voor het geduld dat zij moesten opbrengen op die momenten waarin de ouderlijke zorg en aandacht even op zich lieten wachten.

Roger Hutty

Nog een woord(je) vooraf

De Nederlandstalige uitgave van het boek *Z80 Assembly Language Programming for Students* van Roger Hutty, zoals dat nu voor u ligt, bevat ten opzichte van de originele Engelstalige uitgave een aantal toevoegingen.

Ter verduidelijking van de tekst heb ik een aantal tekeningen opgenomen. U zult deze 'visuele uitleg' vooral aantreffen bij de behandeling van de diverse adresseermethoden maar ook op een aantal andere plaatsen.

Naast deze 'technische' tekeningen treft u een aantal speelsere tekeningen, beter gezegd illustraties, aan. Deze illustraties zijn uitgewerkt en getekend door Eddy Both. Mochten deze illustraties bijdragen tot het besef van de 'relativiteit' van datgene waar je op een bepaald moment druk mee bezig bent, dan denk ik dat ze aan hun doel beantwoorden. In deze illustraties zult u de onderstaande personages nogal eens tegenkomen.

Bij dit boek hoort ook een inlegkaart. Op deze kaart staat een aantal gegevens, zoals een ASCII tabel, diverse conversietabellen, BCD-code en nog meer; handig bij het programmeren in assembleertaal en bij het beantwoorden van de, in dit boek opgenomen, opgaven en opdrachten.

Om in assembleertaal te kunnen werken dient u te beschikken over een assembler programma. Hiermee krijgt u dan de mogelijkheid de mnemonics (Z80-instructies) in te voeren. Het assembler programma zorgt voor de vertaling in machinetaal. Ook een disassembler, die machinetaal terugvertaalt naar mnemonics en een debugger voor het opsporen van fouten in machinetaalprogramma's zijn handige hulpmiddelen. Mocht u dit soort programmatuur willen aanschaffen, dan kunt u daarvoor onder andere bij computerwinkels terecht. Lulu, Frans en Bob van het Computercollectief (Amstel 312a, 1017 AP Amsterdam, tel. 020-223573) hebben voor ons een overzicht samengesteld van Z80 assembler programmatuur voor de TRS-80 model I, II en III, voor de Sinclair ZX81 en ZX Spectrum, voor de Apple met Z80 softcard, voor de Osborne, Newbrain, Kaypro, Xerox 820, DEC Rainbow 100, VT 180 en 8" CP/M systemen en de Exidy Sorcerer. U vindt dit overzicht na de antwoorden van de opgaven (p.179). Voor de aanschaf van of informatie over deze software kunt u onder andere bij bovengenoemd computercollectief terecht.

Ik hoop dat dit boek zijn weg zal vinden naar diegenen die op een of andere manier betrokken raken (of zijn) bij het programmeren in assembleertaal in het algemeen en in Z80 assembleertaal in het bijzonder. Ik denk hierbij aan studenten in het technisch onderwijs, aan cursisten van informaticaopleidingen, maar zeker ook aan de vele personal computer bezitters en hobbyisten die in het bezit zijn van een op de Z80 microprocessor gebaseerd systeem; wellicht willen zij ook wel eens iets anders dan BASIC of snellere programma's schrijven of gewoon 'meer uit hun machine halen'.

Een woord van dank aan Inge Geerling voor het verzorgen van het tik- en zetwerk is dacht ik zonder meer op zijn plaats.

Nok van Veen

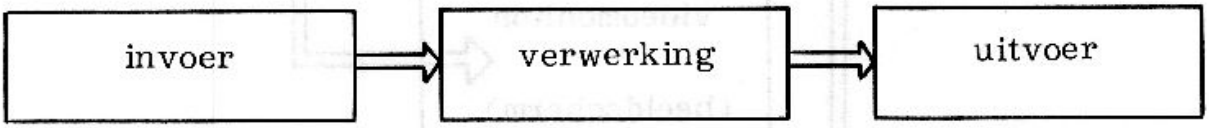
Voorburg, januari 1983

Inhoud

1 De Architectuur van de Z80

1.1 De microprocessor

Gegevensverwerking op een microcomputer is een proces dat, evenals vele andere processen, in drie stappen verloopt:



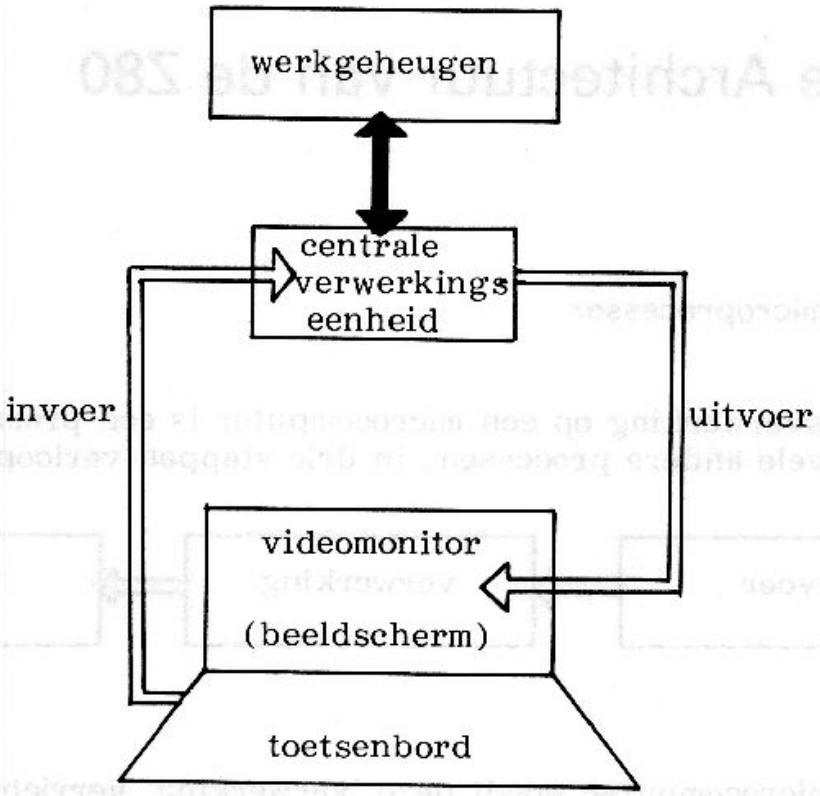
Bij een microcomputer wordt deze 'verwerking' verricht door een microprocessor. In dit boek behandelen we de Z80 microprocessor van Zilog Inc.

Een microcomputersysteem bestaat uit drie hoofdbestanddelen, te weten:
- een centrale verwerkingseenheid
- een werkgeheugen (intern geheugen)
- in- en uitvoerapparatuur.

De centrale verwerkingseenheid, kortweg CPU (Central Processing Unit) genoemd, vormt het hart van het systeem. Als we het hebben over de Z80 microprocessor bedoelen we de CPU. De CPU haalt programma-instructies uit het geheugen, decodeert ze en voert ze uit. De CPU bestuurt ook de in- en uitvoerapparatuur.

Bij een microcomputer bestaat deze in- en uitvoerapparatuur in eerste instantie uit een videomonitor (uitvoer) en een toetsenbord (invoer). Daarnaast kunnen ook een cassetterecorder of een disk-drive (schijfeenheid) als in- en uitvoerapparatuur optreden. Tot slot kan de regeldrukker (printer) als uitvoerapparaat genoemd worden.

In dit boek gaan we uit van een microcomputersysteem, opgebouwd uit de volgende apparatuur (de hardware):



Via het toetsenbord voeren we programma's en gegevens in. Ook geven we op deze wijze bepaalde besturingsopdrachten aan de microprocessor.

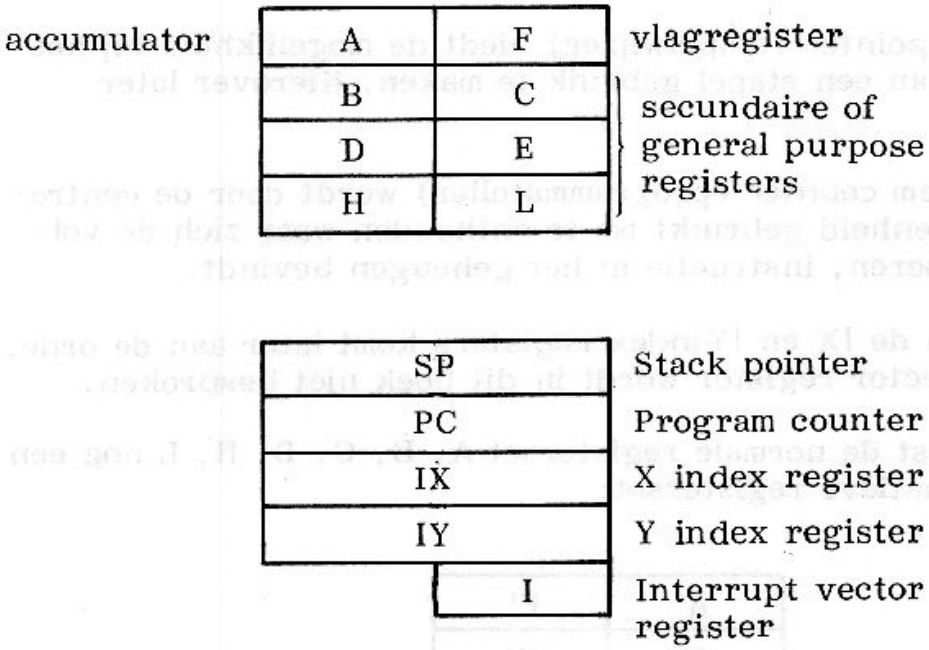
Een programma is een reeks instructies (gecodeerde opdrachten) die de computer vertelt wat hij moet doen. Zo'n programma (met eventuele invoergegevens voor dat programma) wordt opgeslagen in het werkgeheugen van de microprocessor. De CPU zorgt ervoor dat deze instructies één voor één op de juiste wijze worden geïnterpreteerd en uitgevoerd.

Opgave 1.1
Kunnen cassettebandjes en diskettes (floppy disks) ook als in- en uitvoerapparatuur worden opgevat?

Voelt u zich nog niet geheel thuis in het omgaan met het tweetallig en zestientallig getalstelsel en met de manier waarop negatieve getallen in computergeheugens worden weergegeven (bijvoorbeeld de twee-complement methode) bestudeer dan, voordat u verder gaat, eerst Appendix A.

1.2 De Z80 CPU-registers

Kennis over de opbouw van de hardware van de Z80 microprocessor is voor de programmeur van assembleertaalprogramma's niet vereist. De enige onderdelen van de CPU die van groot belang zijn voor de programmeur zijn de zogeheten geheugenregisters. In onderstaande tekening ziet u deze geheugenregisters.



Een geheugenregister is een onderdeel van de CPU waarin informatie (nullen en énen) kan worden opgeslagen. Een geheugenregister maakt géén deel uit van het werkgeheugen van de microprocessor!

De **accumulator** is een **8-bit register**, waarin het resultaat van een rekenkundige of logische bewerking wordt bewaard. Als wij, bijvoorbeeld, met de Z80 twee getallen bij elkaar optellen, moet het eerste getal zich in de accumulator bevinden; daarna wordt het tweede getal hierbij opgeteld en bevindt zich aldus de som van de twee getallen in de accumulator. Hoe we getallen in de accumulator zetten en hoe we erin kunnen optellen wordt in het volgende hoofdstuk behandeld.

Het **Flag register** (vlagregister) wordt gebruikt om informatie over bepaalde bewerkingen in op te slaan. Zo bevat dit vlagregister bijvoorbeeld informatie over het feit of de inhoud van de accumulator, na een optelling, positief, negatief dan wel nul is.

De B, C, D, E, H en L registers vormen de secundaire registers (ook wel algemene of general-purpose registers genoemd) en zij dienen hoofdzakelijk voor het tijdelijk opslaan van gegevens. Zij kunnen gebruikt worden als 8-bit registers maar, door te refereren naar BC, DE of HL, ook als 16-bit registers.

Er is een afspraak om deze secundaire registers op een bepaalde manier te gebruiken, u zult dat in de loop van dit boek ontdekken. Zo wordt doorgaans het HL register gebruikt als verwijzing naar een bepaald gegeven (data) in het geheugen.

De 16-bit stack pointer (stapelwijzer) biedt de mogelijkheid bij het programmeren van een stapel gebruik te maken. Hierover later meer!

De 16-bit program counter (programmateller) wordt door de centrale verwerkingseenheid gebruikt om te onthouden waar zich de volgende, uit te voeren, instructie in het geheugen bevindt.

Het gebruik van de IX en IY index registers komt later aan de orde. Het interrupt vector register wordt in dit boek niet besproken.

De Z80 kent naast de normale registerset A, B, C, D, H, L nog een zogeheten alternatieve registerset:

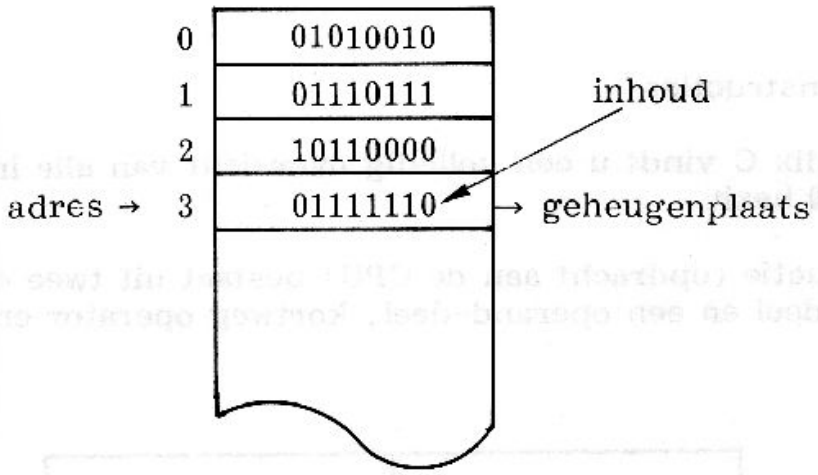
A'	F'
B'	C'
D'	E'
H'	L'

Normaal gesproken worden de 'gewone' registers gebruikt. De programmeur kan echter op elk gewenst moment met slechts twee opdrachten de inhoud van de normale registers met die van de alternatieve registers omwisselen. Dit is vooral handig wanneer de inhoud van alle registers 'gered' moet worden omdat de normale registers tijdelijk voor andere doeleinden gebruikt moeten worden.

Opgave 1.2
Welke registers zouden er gebruikt worden bij het aftrekken van twee getallen en wat is de functie van die registers?

1.3 Het geheugen

Het geheugen van een Z80 microprocessorsysteem bestaat uit een aantal geheugenplaatsen, gewoonlijk aangeduid als bytes. Een byte is 8 bits. In onderstaande figuur ziet u de eerste vier geheugenplaatsen in het geheugen van de Z80.



Het aantal bytes in het geheugen (de grootte van het geheugen) varieert van systeem tot systeem. Bijna altijd zien we 4K, 8K, 16K, 32K, 48K of 64K geheugens. Allemaal 'mooie' binaire getallen (zoals 10, 100, 1000 bij decimale getallen). 1K staat voor 1024 decimaal of 10000000000 binair.

Opgave 1.3
Hoeveel bytes bevat een 64K geheugen, opgegeven als decimaal getal en als hexadecimaal getal?

Geheugenplaatsen worden opvolgend genummerd, beginnend bij nul. Het nummer van een geheugenplaats (byte) wordt het adres genoemd. Elke geheugenplaats bevat een combinatie van 8 bits, de inhoud van die geheugenplaats. Het patroon van 8 bits, de inhoud van een geheugenplaats, kan een instructie voorstellen of een getal of een teken.

Opgave 1.4
Wat is in de tekening de inhoud (hexadecimaal) van de geheugenplaats met adres 2?

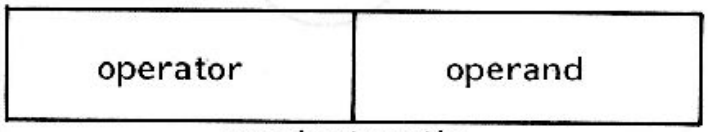
Als we willen aangeven dat we de 'inhoud van de geheugenplaats met een bepaald adres' bedoelen, doen we dit door het adres van die geheugenplaats tussen haakjes te zetten. Zo bedoelen we met (3) de inhoud van de geheugenplaats met adres 3, hetgeen in de tekening hexadecimaal 7E is.

In het vervolg zullen we binaire getallen aangeven met een B erachter en hexadecimale getallen met een H. Getallen zonder enige aanduiding zijn 'gewoon' decimaal.

1.4 Z80 instructies

In Appendix C vindt u een volledig overzicht van alle instructies die de Z80 bezit.

Een instructie (opdracht aan de CPU) bestaat uit twee delen, een operator-deel en een operand-deel, kortweg operator en operand genoemd.



een instructie

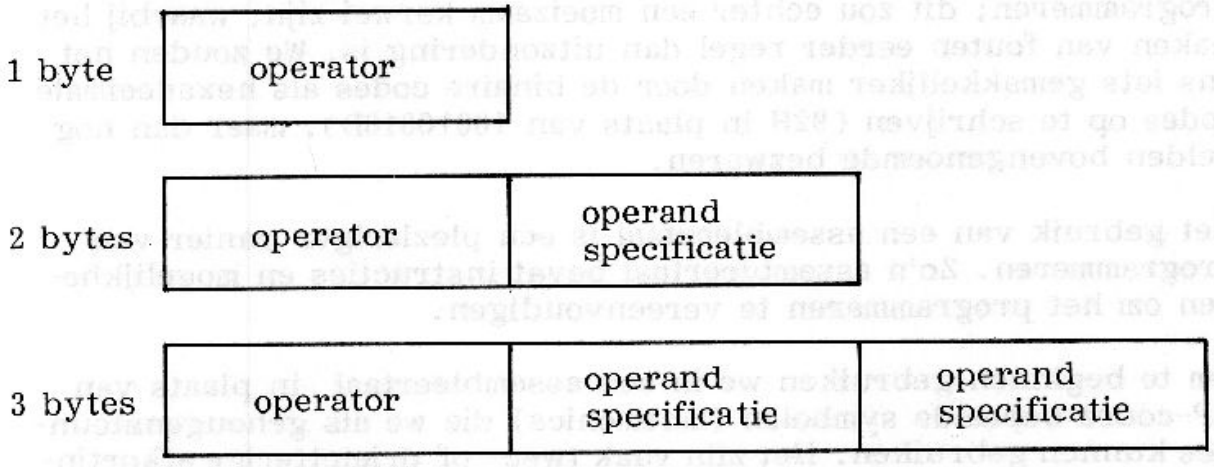
De operator bevat informatie over de soort bewerking die de CPU moet uitvoeren, bijvoorbeeld optellen, aftrekken, vergelijken of kopiëren. Deze informatie is doorgaans niet voldoende om de CPU de opdracht eenduidig te kunnen laten uitvoeren. Wat moet hij optellen, aftrekken, vergelijken of kopiëren? Het antwoord op deze vraag wordt gegeven door de operand. De operand is datgene (of een verwijzing naar datgene) wat opgeteld, afgetrokken, vergeleken of gekopieerd moet worden. Soms is de operand gewoon een constante (een hexadecimaal getal) maar meestal is de operand een adres van (of een verwijzing naar) een geheugenplaats waarvan de inhoud moet worden opgehaald en waarop vervolgens de bewerking plaatsvindt (waarop de operator werkt!).

De Z80 kent een aantal verschillende methoden waarop het adres van de eigenlijke operand gespecificeerd kan worden. Dit heeft tot gevolg dat de Z80 een aantal verschillende 'adresseermethoden' kent. Dit betekent dat in een bepaald soort instructie verschillende adresseermethoden gebruikt kunnen worden. Zo kan met de LOAD (LD)

instructie een bepaald CPU-register gekopieerd worden in een ander CPU-register. Maar we kunnen ook met een LOAD instructie de inhoud van een bepaalde geheugenplaats kopiëren in een CPU register. De wijze van adressering (waar haal je de operand vandaan) is een onderdeel van de operator. Bovendien is het zo dat een bepaald soort instructie voor verschillende CPU registers gebruikt kan worden. Zo zijn er kopieer (LD) instructies voor het kopiëren van een operand in het B register, in het E register, in de Accumulator, enzovoorts. De registers doen dan ook dienst als operand en wel als 'vaste' operand bij een bepaalde instructie. Zo'n vaste operand wordt onderdeel van de operator. De operator zegt dus iets over:
a. de soort instructie;
b. het register, vastgebakken aan de instructie;
c. de wijze van adressering (waar bevindt zich de operand).

Deze informatie ligt vast in een OP (OPeration) code van 8 bits (één byte). De operator (in de vorm van een OP code) neemt één byte in een instructie in beslag. Er zijn slechts een paar instructies, waarin geïndexeerde adressering gebruikt wordt, die twee bytes voor de operator claimen.

Er zijn instructies die geen operand hebben. Deze instructies zijn dus 1 byte lang. Instructies die wel een operand bevatten zijn naar gelang de specificatie van die operand (de adressering) 2 of 3 bytes lang. We kunnen dit als volgt weergeven:



Lengte van een Z80-instructie

In Appendix C kunt u vinden hoe 'lang' een bepaalde instructie is.

De OP code is voor een bepaalde instructie een vast 8-bit patroon (zie appendix C). Zo heeft de 1-byte-instructie 'trek de inhoud van register D af van de inhoud van de accumulator' als OP code

10010010B

Het deel 10010 is voor de CPU de aanwijzing dat er iets van de accumulator moet worden afgetrokken. Het deel 010 geeft aan dat het gaat om de inhoud van register D. Ziet u dat hier register D als 'operand' dienst doet en dat deze operand als het ware een deel is van de operator! (Zowel de accumulator als register D zijn 'vaste' operanden!)

Opgave 1.5
Geef de OP code van de instructie
"Tel inhoud van register C op bij de inhoud van de accumulator".
(zie appendix C)

De Z80 staat een operand van twee bytes toe. Als deze operand het adres is van een geheugenbyte, is het grootste adres dat gespecificeerd kan worden het adres 1111111111111111B. Dit is het adres 65535. De Z80 instructies kunnen dus een (RAM) geheugen van 65536 (0 t/m 65535) geheugenplaatsen bestrijken. Dit is een geheugen van 64K bytes.

1.5 Assembleertaal

De computer slaat instructies in binaire code op. Wij, als programmeur, zouden in principe onze programma's in binaire code kunnen programmeren; dit zou echter een moeizaam karwei zijn, waarbij het maken van fouten eerder regel dan uitzondering is. We zouden het ons iets gemakkelijker maken door de binaire codes als hexadecimale codes op te schrijven (92H in plaats van 10010010B), maar dan nog gelden bovengenoemde bezwaren.

Het gebruik van een assembleertaal is een plezieriger manier van programmeren. Zo'n assembleertaal bevat instructies en mogelijkheden om het programmeren te vereenvoudigen.

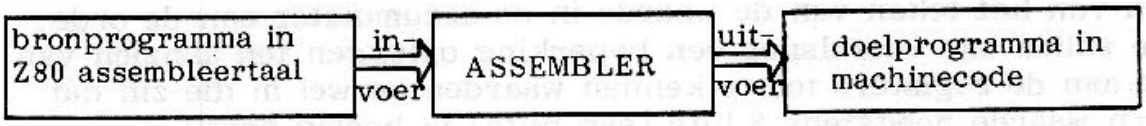
Om te beginnen gebruiken we in een assembleertaal in plaats van OP-codes bepaalde symbolen (mnemonics) die we als geheugensteuntjes kunnen gebruiken. Het zijn vaak twee- of drieletterige afkortingen van woorden, die aangeven wat de bewerking inhoudt. De eerder genoemde instructie 'trek register D af van de accumulator' kan in assembleertaal worden geschreven als:

```
SUB D                    (SUB is een mnemonic voor
                          SUBTRACT)
```

hetgeen even eenvoudiger is om te onthouden dan 10010010B of 92H. Ook ziet u dat de 'operand' uit de instructie als de letter D geschreven mag worden in plaats van als 010B!

Opgave 1.6
Wat is de assembleertaalinstructie om "register B met één op te hogen"? (ophogen = increment). Gebruik tabel C.5, appendix C.

Assembleertaalprogramma's kunnen niet zonder meer door een computer worden uitgevoerd. Zulke programma's moeten eerst 'vertaald' worden in een binaire code, de machinecode genoemd. Dit vertalen wordt verzorgd door een programma, dat een **assembler** heet. Invoer voor de assembler is een assembleertaalprogramma dat we het bronprogramma noemen. Uitvoer van de assembler is een programma in machinecode, het doelprogramma. Dit programma kan door de microprocessor worden uitgevoerd.



Het assembleren van een bronprogramma.

2 Instructies voor de Accumulator en andere Registers

In dit hoofdstuk komen instructies voor het laden van registers, voor het optellen en aftrekken in de accumulator, voor het ophogen en verlagen van (de inhoud vàn de) registers en voor het verwisselen van het teken van de waarde in de accumulator aan de orde. We zullen ons vooralsnog een beperking opleggen ten aanzien van de aan de registers toe te kennen waarden en wel in die zin dat zo'n waarde hoogstens 8 bits (één byte) in beslag neemt.

2.1 Het laden van een register

We kunnen aan elk register afzonderlijk een waarde toekennen. We zeggen dan 'we laden het register met .....'. Dat kan met de twee-bytes instructie:

```
LD r,n
```

LD is een mnemonic voor LOAD. r staat voor één van de registers A, B, C, D, E, H of L en n is een getal zonder teken tussen 0 en 255 of een getal met teken in het bereik van -128 tot +127. Zo zal de instructie

```
LD B,99
```

de waarde 99 (het getal 99) toekennen aan register B.

Opgave 2.1
Waarom deze beperking voor de waarden van n?

2.2 Optelllen en aftrekken in de accumulator

Optellen en aftrekken in de accumulator gaat als volgt:

`ADD A,n` en `SUB n`

De linker instructie telt de waarde van n bij de Accumulator (A) op, de rechter instructie trekt de waarde van n ervanaf. Beide resultaten blijven in de accumulator achter! Bekijk de volgende drie instructies:

```
LD   A,15
ADD  A,46
SUB  22
```

Na de LD-instructie bevat de accumulator het getal 15, na de ADD-instructie 61 en na de SUB-instructie 39.

Merk op dat in de ADD-instructie de Accumulator (register A) expliciet genoemd wordt, terwijl de SUB-instructie veronderstelt dat met de accumulator gewerkt wordt! (vaste operand)

Opgave 2.2
Schrijf een reeks instructies om in de accumulator het volgende sommetje uit te rekenen: 73 + 55 - 21.

2.3 Optellen en aftrekken met registers

De inhoud van elk willekeurig register kunnen we bij die van de accumulator optellen of er van aftrekken, waarbij het resultaat van de bewerking in de accumulator achterblijft.

`ADD A,r` en `SUB r`

r is één van de registers A t/m L. Hiermee kunnen tussenresultaten die tijdelijk in één van de registers zijn opgeslagen, bij de inhoud van de accumulator worden opgeteld, dan wel er van af worden getrokken.

2.4 Het ene register kopiëren in het andere

Als we de inhoud van register r2 willen kopiëren in register r1 gaat dat zo:

```
LD r1,r2
```

Deze instructie wordt dikwijls gebruikt voor het 'redden' van de inhoud van de accumulator. Dit komt voor als de accumulator tijdelijk voor iets anders gebruikt moet worden. De LD r1,r2 instructie laat de inhoud van register r2 intact. Alle LD-instructies hebben dus een kopieerfunctie en geen 'laad en veeguit'- of 'verplaats'-functie.

Opgave 2.3
Schrijf een reeks opdrachten voor het berekenen van 3 × (56 - 22). Doe dit door twee keer 56-22 bij zichzelf op te tellen.

2.5 Ophogen en verlagen van een register

De één-byte instructies

```
INC r
```

en

```
DEC r
```

verhoogt (INCrement), respectievelijk verlaagt (DECrement) de inhoud van register r met één. U zult deze instructies straks tegenkomen als we programmalussen gaan programmeren. Ook zijn deze instructies handig en efficiënt als we tijdens een rekenproces bij de inhoud van een bepaald register één willen optellen of van de inhoud één willen aftrekken. Bovendien is het zo dat de instructie INC A (verhoog accumulator met 1) 'sneller' is dan de instructie ADD A,1, die ook één bij de accumulator optelt. INC A is een één-byte instructie en ADD A,1 neemt twee bytes in het geheugen in beslag. Hetzelfde geldt voor DEC A en ADD A,-1 of SUB 1. Hoe korter een instructie des te sneller wordt hij uitgevoerd.

2.6 Accumulator van teken verwisselen

De volgende één-byte instructie

```
NEG
```

draait het teken van de waarde in de accumulator om. Als de accumulator bijvoorbeeld 78 bevat, is deze waarde na een NEG-opdracht -78 (NEG is kort voor NEGate).

Opgave 2.4
Wat is de inhoud van de accumulator, zowel in decimale vorm als in hexadecimale vorm, na elk van de volgende instructies:

```
LD  A,27
NEG
INC A
```

2.7 'Onmiddellijke' en 'uitgebreide' adressering

De instructies LD r,n; ADD A,n en SUB n impliceren een zogenaamde onmiddellijke adressering (immediate adressing) en wel omdat de waarde van de operand (n) in de instructie is opgenomen (als tweede byte van deze instructies). Voordat we een voorbeeld van onmiddellijke adressering geven eerst even het volgende.

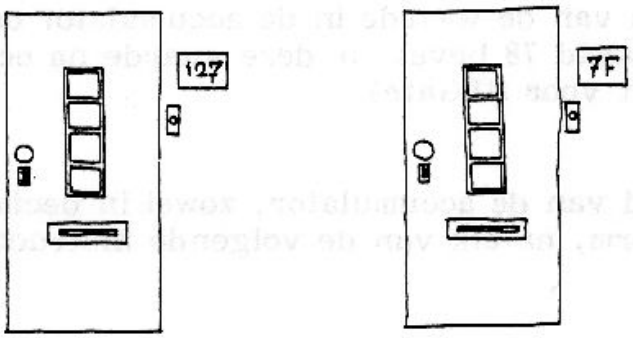
Instructies (programma's) die de CPU moet uitvoeren liggen opgeslagen in het interne geheugen (ook wel werkgeheugen genoemd) van de microprocessor. Het interne geheugen is voor te stellen als een aaneenrijging van geheugenplaatsen. Elke geheugenplaats kan (bij de Z80 althans) één byte (8 bits) aan informatie bevatten.

Instructies worden dus in het geheugen als bytes opgeslagen. Nu kan een Z80-instructie uit 1, 2, 3 of 4 bytes bestaan. Is een instructie langer dan één byte dan zullen de bytes uit zo'n instructie in opeenvolgende geheugenplaatsen worden opgeslagen.

De CPU 'weet' hoe lang elke instructie is (dat kan hij zien aan de inhoud van de eerste byte). Als de CPU nu ook nog weet waar de uit te voeren instructie in het geheugen ligt opgeslagen, dat wil zeggen: kent hij het adres van de eerste byte uit die instructie, dan kan hij de instructie dus inlezen (1, 2, 3 of 4 bytes) en uitvoeren.

Hoe de CPU weet van welk adres hij een instructie moet ophalen zal u (hopenlijk) in de loop van dit boek duidelijk worden.

In de voorbeelden die we bij de diverse adresseermogelijkheden geven gaan we uit van een 64K geheugen. Dit is een geheugen met 64 × 1024 = 65536 geheugenplaatsen (bytes), genummerd van 0 tot en met 65535. We zullen in het vervolg de adressen van geheugenplaatsen als hexadecimale getallen opgeven.



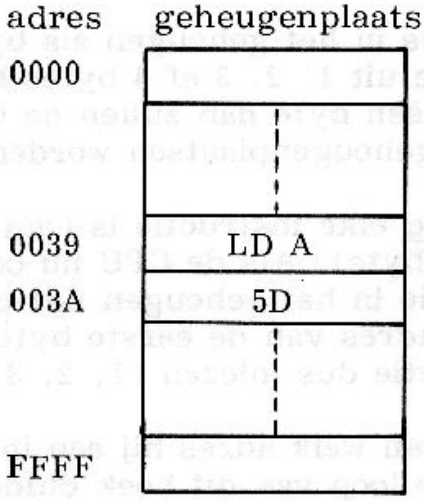
In een 64K geheugen gebruiken we dus adressen van 0000H tot en met FFFFH (FFFFH = $15\times16^3 + 15\times16^2 + 15\times16 + 15 = 65535$).

Nu gaan we terug naar de onmiddellijke adressering.

Een voorbeeld van onmiddellijke adressering is de volgende opdracht:

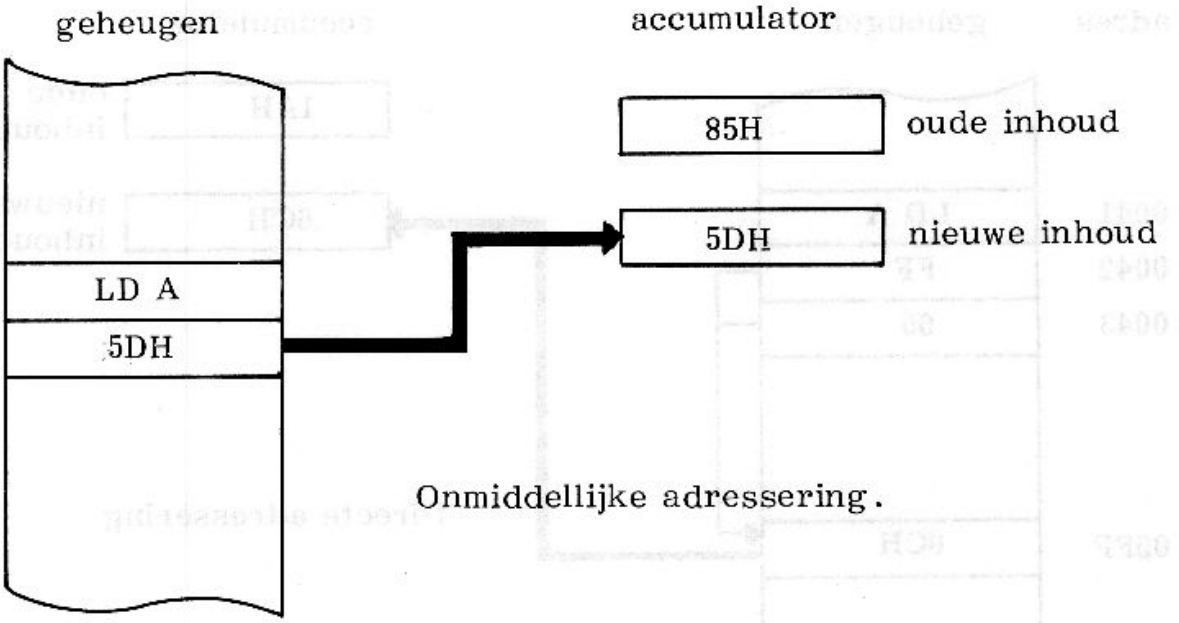
LD A,5DH

We laden de accumulator met 5DH (met 93 dus).
De operand 5D is een stukje data en geen adres van een geheugenplaats. Als LD A,5DH een instructie is uit een assembleertaalprogramma, dan staat deze instructie ergens in het interne geheugen. Wellicht op de adressen 0039 en 003A (denk erom: hexadecimaal!):



Het effect van deze instructie met onmiddellijke adressering is dat de inhoud van de tweede byte van de instructie gekopieerd wordt in de accumulator.

Op de volgende bladzijde ziet u dat als de instructie LD A,5DH wordt uitgevoerd, de oude inhoud van de accumulator (85H) wordt vervangen door de nieuwe waarde (5DH) uit de LD-instructie.



Een andere adresseermethode is de **uitgebreide adressering** (extended adressing) of **directe adressering**. In plaats van de operand zelf geven we het adres op waar de operand (de inhoud van die geheugenplaats) gezocht moet worden. Een instructie met directe adressering is de volgende:

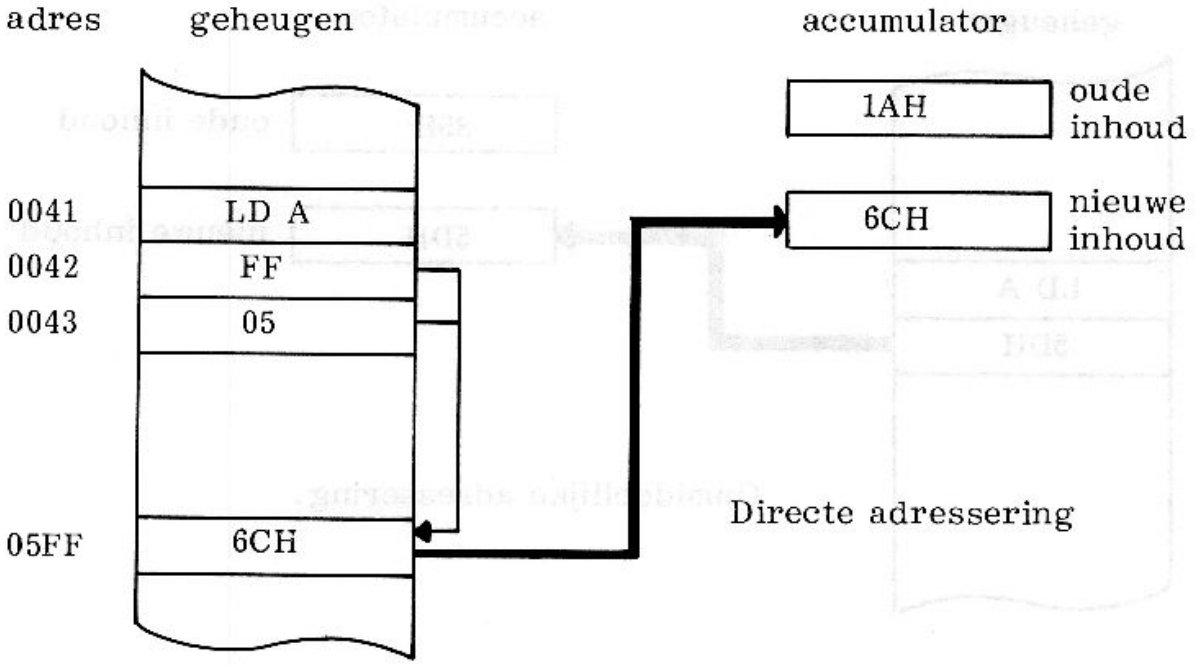
`LD A,(nn)`

laadt de accumulator met de inhoud van de geheugenplaats met adres nn. Deze instructie is drie bytes lang. Een byte voor de OP-code en twee bytes om het adres te specificeren (we moeten 64K kunnen bestrijken).

Een voorbeeld: LD A,(05FFH)

De adresspecificatie 05FF kost twee bytes, een zogenaamd lage-orde byte FF en een hoge-orde byte 05. In de tekening op p.16 zien we de situatie waarbij deze instructie zelf ligt opgeslagen op de adressen 0041, 0042 en 0043 en ook zien we het effect van deze directe adressering als de instructie zou worden uitgevoerd. Aangenomen is dat de inhoud van adres 05FF de waarde 6CH is.

In de tekening staat op adres 0041 "LD A". In werkelijkheid staat daar natuurlijk niet LD A maar de OP-code van de instructie LD A,(nn). Deze OP-code is 3AH of 00111010B. Strikt genomen



staat op adres 0041 een byte van 8 bits, dus 00111010!
Merk ook op dat de lage-orde byte FF direct volgt op de OP-code.

Zo kunnen we ook de inhoud van de accumulator op een bepaalde geheugenplaats 'stallen':

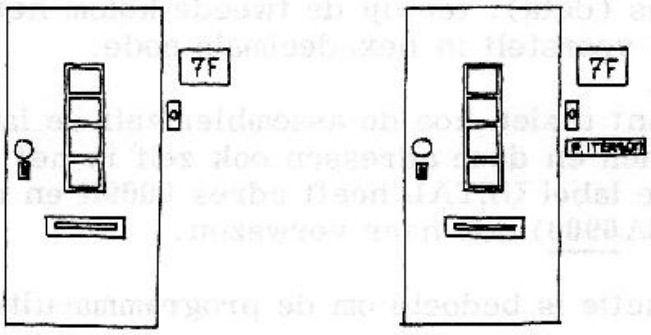
```
LD (nn),A
```

Nu wordt de inhoud van de accumulator gekopieerd op de geheugenplaats met adres nn.
Ook deze instructie neemt drie bytes in beslag omdat de adresspecificatie van de operand twee bytes kost.

Opgave 2.5
Stel geheugenplaats 35H bevat 79. Wat is de inhoud van de accumulator na elk van de volgende instructies:
a. LD A,35H en b. LD A,(35H)

Als u naar tabel C2 in Appendix C kijkt, zult u zien dat van de enkelvoudige (8-bit) registers alleen de accumulator in directe adressering gebruikt mag worden. In de loop van het boek bespreken we nog een aantal andere adresseringsmogelijkheden.

2.8 Labels



adres adres met label

Bij het schrijven van programma's komt het dikwijls voor dat we in instructies willen verwijzen naar een bepaalde geheugenplaats. Nu kan dit, dat hebben we in de vorige paragraaf gezien, door het opgeven van het adres van de desbetreffende geheugenplaats. Om diverse redenen is het vaak een moeilijke zaak deze adressen bij te houden. De assembler komt ons hier te hulp door de mogelijkheid te bieden labels te gebruiken. Een label is een naam die we aan een bepaalde geheugenplaats geven. In de instructies geven we dan 'namen' op in plaats van adressen. We geven natuurlijk niet elke geheugenplaats een naam maar alleen die waarnaar in één (of meer) instructies verwezen wordt. Een voorbeeld zien we in onderstaand programma, waarin bij een bepaald GETAL 10 wordt opgeteld:

```
                    ;Programma 2.1  telt 10 op bij een getal
                    ;
0000  3A0900            LD   A,(GETAL)
0003  C60A              ADD  A,10
0005  320A00            LD   (SOM),A   ; som = getal + 10
0008  76                HALT
0009  4A          GETAL: DEFB 74
000A  00          SOM  : DEFB 0
```

De programmeur kiest zelf de labels. Een label mag ten hoogste uit zes cijfers en/of letters bestaan, waarbij het eerste teken een letter moet zijn. Na een label komt altijd een dubbele punt. Een verstandig (en vooral spaarzaam) gebruik van labels komt de begrijpelijkheid van uw programma's ten goede; labelt u er maar op los, dan leidt dit bijna altijd tot 'slechte' programma's.

Het bovenstaande programma vormt de uitvoer van de assembler na het assembleren van het bronprogramma. De eerste kolom bevat de adressen van de geheugenplaatsen van de diverse instructies en van de gegevens (data), terwijl de tweede kolom het doelprogramma (de 'vertaling') voorstelt in hexadecimale code.

In de listing kunt u zien hoe de assembler zelf de labels van adressen heeft voorzien en deze adressen ook zelf in het doelprogramma gebruikt. De label GETAL heeft adres 0009H en daar wordt in regel 1 (0000 3A0900) ook naar verwezen.

De HALT instructie is bedoeld om de programma-uitvoering te stoppen.

De DEFB pseudo-instructie vraagt aan de assembler een byte te reserveren en aan deze byte (geheugenplaats) een bepaalde waarde toe te kennen. Pseudo-instructies komen nog uitvoerig ter sprake.

Om assembleertaalprogramma's 'leesbaar' te maken is het beslist noodzakelijk commentaar in de programma's op te nemen. Zo begint bovenstaand programma ook met een commentaarregel. Een commentaarregel begint met een puntkomma. De assembler negeert alles wat na een puntkomma op een regel staat. Hij drukt het alleen op de listing af. Commentaar kunt u op elke gewenste positie in een regel opnemen, ook aan het begin. Hoeveel commentaar in een programma moet worden opgenomen valt moeilijk te zeggen. U kunt enigszins een idee krijgen door te kijken naar de in dit boek gegeven voorbeelden. Te weinig is niet goed, maar teveel commentaar ook zeker niet.

2.9 Een programmeeropdracht

Schrijf een programma voor het berekenen van

RESULT = N1 - 3(N2 + 1) - 1

Definieer RESULT, N1 en N2 als labels (zie 2.8) van gegevenselementen (DEFB) aan het einde van uw programma.

Als u waarden gaat toekennen aan deze 'gegevensbytes', zorg er dan voor dat alle tussenresultaten en het eindRESULTaat liggen tussen -128 en +127.

Stop het programma in de computer en laat het assembleren (vertalen in machinecode). Controleer de berekeningen door de inhoud van de geheugenplaats (de byte) "RESULT" op te vragen. Vraag aan uw

docent of kijk in een gebruikershandleiding hoe u de inhoud van een bepaalde byte (geheugenplaats) moet opvragen.

Om te kunnen zien wat er tijdens de berekening allemaal gebeurt moet u stap voor stap het programma volgen. Bekijk na elke uitgevoerde instructie de inhoud van de relevante registers en geheugenplaatsen (RESULT,N1,N2). Vraag alweer aan uw docent of aan uw gebruikershandleiding hoe u de instructies uit uw programma stap voor stap kunt laten uitvoeren.

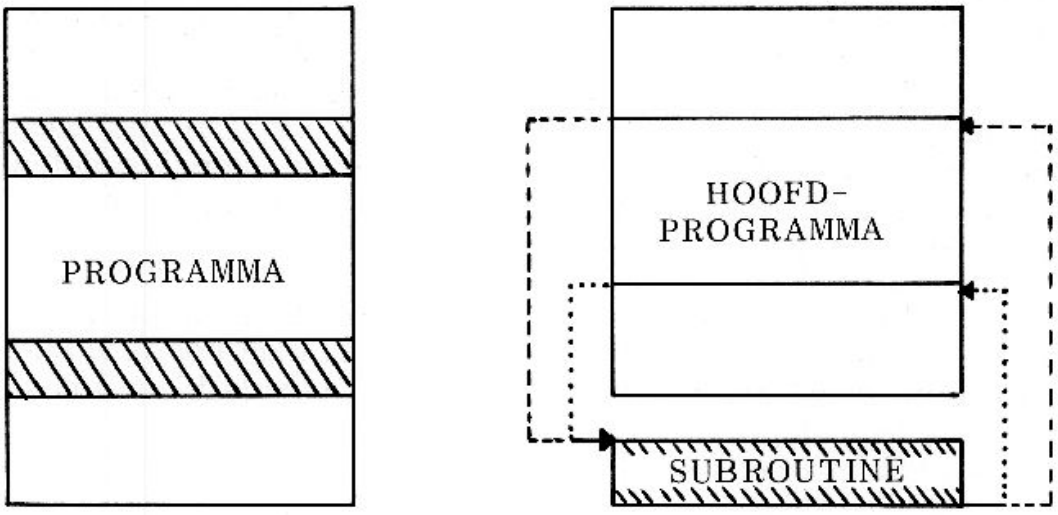
Kies andere waarden voor N1 en N2 en herhaal het hele proces van assembleren, eindcontrole en 'stap-voor-stap'-controle.

3 Subroutines en Uitvoer op het Beeldscherm

3.1 Het begrip SUBROUTINE

Subroutines spelen bij het programmeren een belangrijke rol. Jammergenoeg vindt de introductie ervan veelal pas aan het einde van een programmeercursus plaats. U leert echter nu al waarom subroutines gebruikt worden en hoe u ze moet gebruiken. Wat u nu nog niet leert is hoe het mechanisme precies werkt; dat komt in hoofdstuk 7 aan de orde.

Laten we eerst eens kijken waarom subroutines gebruikt worden. Stel een programma bevat twee groepen instructies die in feite precies hetzelfde doen. Deze twee instructiegroepen zijn in de linker tekening als gearceerde stroken weergegeven.



Het opnemen van twee of meer keren van dezelfde groep instructies kost tijd, en geheugenruimte.
Het idee is nu de groep instructies als 'subroutine' te definiëren en deze subroutine slechts één maal in het programma op te nemen.

meestal aan het begin of het einde van een programma (zie tekening). Het programma bestaat dan uit twee delen. Het eerste (of het laatste) noemen we het hoofdprogramma (ook wel hoofdmodule) en het tweede deel is de subroutine (of submodule).

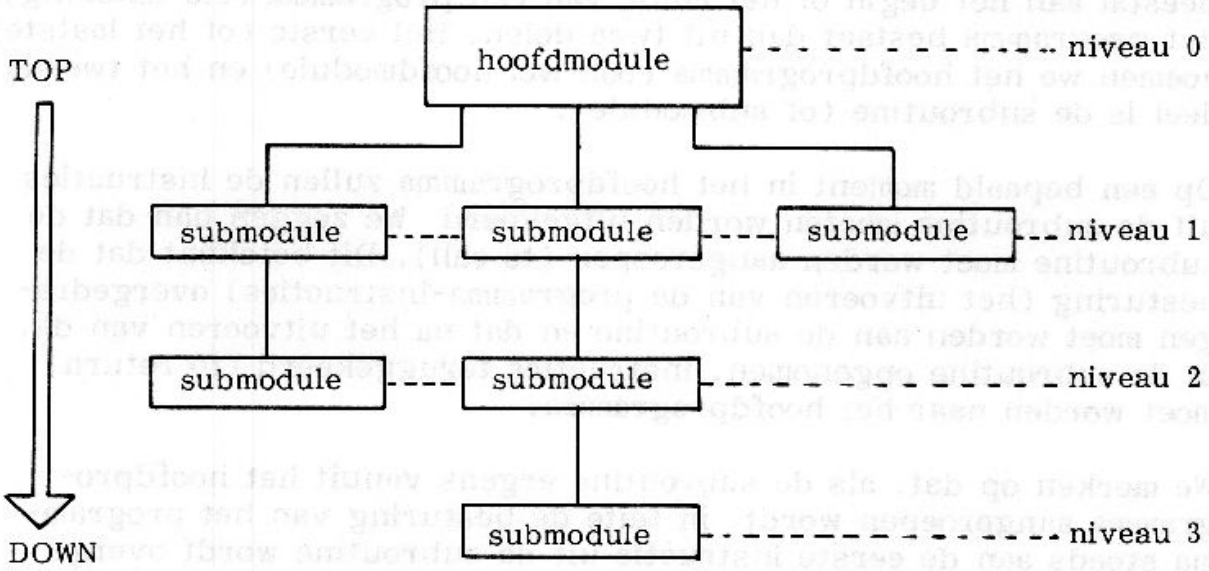
Op een bepaald moment in het hoofdprogramma zullen de instructies uit de subroutine moeten worden uitgevoerd. We zeggen dan dat de subroutine moet worden aangeroepen (to call). Dit betekent dat de besturing (het uitvoeren van de programma-instructies) overgedragen moet worden aan de subroutine en dat na het uitvoeren van de, in de subroutine opgenomen, instructies teruggekeerd (to return) moet worden naar het hoofdprogramma.

We merken op dat, als de subroutine ergens vanuit het hoofdprogramma aangeroepen wordt, in feite de besturing van het programma steeds aan de eerste instructie uit de subroutine wordt overgedragen, maar dat bij terugkeer uit de subroutine de uitvoering van het hoofdprogramma steeds op een ander punt moet worden hervat, namelijk met de instructie volgend op die waarin de subroutine werd aangeroepen.

Hoe de computer precies weet naar welk punt in het hoofdprogramma teruggekeerd moet worden zult u later in dit boek aan de weet komen. In de volgende paragraaf worden de instructies voor het 'aanroepen van' en 'terugkeren uit' subroutines behandeld.

Subroutines worden niet alleen gebruikt in situaties waarin verschillende keren eenzelfde groep instructies moet worden uitgevoerd. Ook indien een aantal instructies in een programma slechts één maal wordt uitgevoerd is het mogelijk zo'n groep instructies in een subroutine op te nemen. Dit doet zich voor bij 'gestructureerd programmeren', waarbij een programma in een hoofdmodule en een aantal submodules wordt onderverdeeld, zodanig dat de hoofdmodule een soort besturende taak heeft en de submodules een uitvoerende taak.

Bij het ontwikkelen van submodules (subroutines) kunnen we de TOP-DOWN methode hanteren, waarbij we met de hoofdmodule beginnen en vandaaruit submodules op een steeds lager niveau ontwikkelen.



Gestructureerd TOP-DOWN programmeren.

3.2 De CALL en RET instructies

In de Z80 microprocessor worden subroutines aangeroepen met de CALL instructie, gevolgd door de label van de eerste instructie uit die subroutine. Programma 3.1 is een voorbeeld van een programma waarin vanuit het hoofdprogramma tweemaal dezelfde subroutine wordt aangeroepen.

```
; Programma 3.1 vermenigvuldigt twee getallen met 4
;
        LD   A,(N1)     ; begin hoofdprogramma
        CALL KWAD       ; berekent N1 x 4
        LD   (R1),A     ; R1 = N1 x 4
;
        LD   A,(N2)
        CALL KWAD
        LD   (R2),A     ; R2 = N2 x 4
        HALT
N1:     DEFB 31
N2:     DEFB 25
R1:     DEFB 0
R2:     DEFB 0
;
; Subroutine - vermenigvuldigt de accumulator met 4
;
KWAD:   ADD A,A         ; A x 2
        ADD A,A         ; (A x 2) x 2 = A x 4
        RET
```

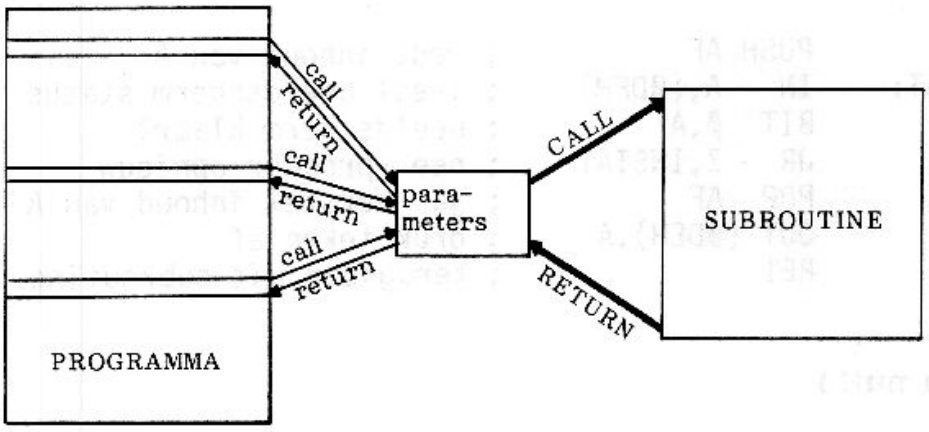
De subroutine, die door middel van de label KWAD wordt aangeroepen, vermenigvuldigt het getal in de accumulator met vier.

Als we dit programma tijdens de uitvoering volgen, zien we dat, voordat de subroutine met CALL KWAD wordt aangeroepen, de accumulator eerst met N1 geladen wordt. Door het aanroepen van de subroutine wordt eerst de opdracht ADD A,A uitgevoerd, gevolgd door nog een opdracht ADD A,A. Door de RET instructie wordt teruggekeerd naar de instructie LD (R1),A in het hoofdprogramma. Dit is de instructie direct volgend op de CALL KWAD instructie waarmee de subroutine werd aangeroepen.

Na het uitvoeren van de instructie LD (R1),A wordt de instructie LD A,(N2) uitgevoerd, waardoor de accumulator geladen wordt met N2 alvorens in de daaropvolgende instructie voor de tweede maal de subroutine KWAD wordt aangeroepen, natuurlijk weer met CALL KWAD.

Na het uitvoeren van de tweede ADD A,A en de RET instructie keert de programmabesturing ditmaal terug bij de LD (R2),A instructie die volgt op de tweede CALL instructie. Tenslotte eindigt het programma doordat de computer de HALT instructie uitvoert.

De accumulator wordt in dit programma gebruikt om het getal dat vervierVoudigd moet worden van het hoofdprogramma over te brengen naar de subroutine en ook om het vervierVoudigde getal (dit gebeurt immers in de subroutine) van de subroutine over te brengen naar het hoofdprogramma. We zeggen in dit geval dat de accumulator gebruikt wordt voor het doorgeven van parameters tussen het hoofdprogramma en de subroutine. De parameters zijn het te vervierVoudigen getal en het vervierVoudigde getal. Voor een dergelijke parameteroverdracht kunnen ook andere registers, geheugenplaatsen of de stapel gebruikt worden.



Parameteroverdracht.

Opgave 3.1
Schrijf een subroutine, waarin de inhoud van de registers B, C, D, E, H en L bij elkaar wordt opgeteld en waarbij de som in de accumulator achterblijft; schrijf een passend hoofdprogramma, vanwaaruit deze subroutine wordt aangeroepen.

Doorgaans worden er vanuit een hoofdprogramma verschillende subroutines aangeroepen. Al deze subroutines nemen we na elkaar op aan het begin of aan het einde van een programma.

3.3 Uitvoer op het beeldscherm

Een voordeel van subroutines is ook dat je als programmeur een subroutine van een collega kan gebruiken zonder te hoeven weten hoe de subroutine precies werkt.

Zo zijn er weinig boeken over assembleertalen, waarin je al in hoofdstuk 3 leert hoe je iets op een beeldscherm kunt laten afdrukken. Als u echter een subroutine krijgt die deze uitvoer voor u regelt en als u wordt verteld hoe u deze subroutine moet gebruiken, dan kunt u tekens op het beeldscherm laten afdrukken, zonder echter te weten hoe de subroutine precies in elkaar zit. Welnu, de volgende subroutine regelt het afdrukken van één teken op de regeldrukker.

```
; programma 3.2
;
; subroutine voor het afdrukken van een teken op het beeldscherm
;  -bij aanroep bevat de accumulator de code van het af te
;   drukken teken
;
COUT:     PUSH AF          ; redt inhoud van A
INSTAT:   IN   A,(ØDFH)    ; leest beeldscherm status
          BIT  Ø,A         ; beeldscherm klaar?
          JR   Z,INSTAT    ; nee -probeer opnieuw
          POP  AF          ; ja -herstel inhoud van A
          OUT (ØDEH),A     ; druk teken af
          RET              ; terugkeer uit subroutine
```

(Ø is een nul!)

Deze routine wordt aangeroepen met CALL COUT. Wel moet vlak voor de aanroep de accumulator de code van het af te drukken teken

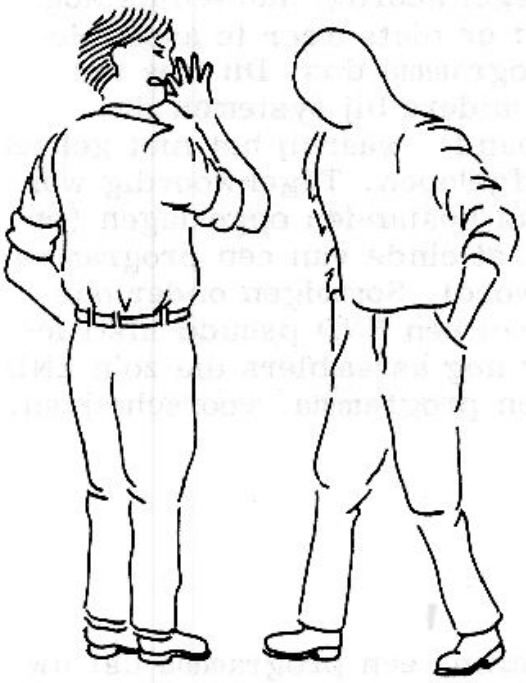
bevatten! Merk op dat na terugkeer uit de subroutine de accumulator nog steeds de desbetreffende code bevat.

De ASCII codes voor de diverse tekens vindt u in Appendix D. De ASCII code is het binaire bitpatroon dat gebruikt wordt bij het transport van een teken tussen de CPU en de randapparatuur.

Opgave 3.2
Wat is de ASCII code voor het teken G en welk teken heeft een ASCII code van 2BH (2B hexadecimaal)?

In assembleertaalinstructies kunnen we de code van een teken opgeven als een numerieke (bijvoorbeeld hexadecimale) code of door het teken zelf tussen '-tekens te plaatsen. Zo kunnen we zowel de instructie LD A,2AH als de instructie LD A,'*' gebruiken voor het laden van de accumulator met de ASCII code van het teken *. De tweede instructie valt wellicht te prefereren omdat die wat meer zegt dan de eerste. De assembler vertaalt '*' voor ons in een numerieke code.

3.4 Pseudo-instructies



pseudo-instructie tegen assembler:
Pssst ..... ik moet je wat vertellen.

Pseudo-instructies zijn instructies die niet echt uitgevoerd worden door de computer, ja zelfs niet geassembleerd worden, maar die slechts dienen als informatie voor de assembler. We hebben al zo'n pseudo-instructie gezien, namelijk de DEFB instructie. Hiermee geven we de assembler alleen te kennen dat we een bepaalde geheugenplaats met inhoud willen definiëren.

In het boek zult u nog meer van dergelijke pseudo-instructies of pseudo-operatoren (ook wel directives genoemd) tegenkomen. We zullen nu even stilstaan bij twee van deze, overigens weinig gebruikte, pseudo-instructies.

De ORG nn pseudo-instructie vertelt de assembler dat de eerstvolgende instructie zodanig geassembleerd moet worden dat de instructie begint op adres nn. De mnemonic ORG (kort voor ORiGin) is gekozen omdat deze pseudo-instructie hoofdzakelijk gebruikt wordt om aan te geven waar in het geheugen (op welk adres) een programma begint, alhoewel de ORG pseudo-instructie overal in een programma kan voorkomen. Met de ORG instructie kan de programmeur precies aangeven waar een programma in het geheugen moet worden opgeslagen. De meeste microprocessors worden echter bestuurd door een besturingssysteem, zoals bijvoorbeeld CP/M, dat ofwel zelf de toewijzing van geheugenruimte aan programma's regelt, ofwel toestaat dat een bepaald absoluut adres opgegeven wordt waarop een programma na het assembleren moet worden opgeslagen, hetgeen meestal gebeurt in de link fase. De algemene gang van zaken is tegenwoordig echter dat een assembler een programma relatief ten opzichte van byte nul assembleert, zodat de pseudo-instructie ORG (vrijwel) niet meer gebruikt wordt.

Opgave 3.3
Kijk nog eens naar programma 2.1 op blz.17. Stel dat de eerste instructie voorafgegaan wordt door de pseudo-instructie ORG 1000 H, wat zou dan het adres van de byte, waar de HALT instructie in is opgeslagen, geweest zijn?

Ook de END pseudo-instructie wordt tegenwoordig nauwelijks nog gebruikt. END vertelt de assembler dat er niets meer te assembleren valt, gewoon het einde van een programma dus. Dit was wel noodzakelijk bij oude systemen; onder andere bij systemen die gebruik maakten van papertape (ponsband), waarbij het niet geheel duidelijk was dat een programma was afgelopen. Tegenwoordig worden programma's meestal in de vorm van bestanden opgeslagen (en bewaard) en de assembler constateert het einde van een programma dan als een end-of-file teken gelezen wordt. Sommigen onder ons vinden nog steeds dat elk programma met een END pseudo-instructie zou moeten eindigen, en ook zijn er nog assemblers die zo'n END instructie, als laatste instructie van een programma, voorschrijven.

3.5 Een programmeeropdracht

Schrijf met behulp van de COUT subroutine een programma dat uw voorletters op het beeldscherm afdrukt, gevolgd door een carriage-return en line-feed (naar het begin van een nieuwe regel). De ASCII codes voor Carriage-Return en Line Feed kunt u in Appendix D vinden onder CR en LF.
Lees de opmerkingen op de volgende bladzijde!

Opmerking-1.
Controleer of de subroutine COUT uit 3.3 ook in elk opzicht werkt op uw Z80 microprocessor systeem. Informeer desnoods bij uw computerleverancier of alle instructies uit COUT ook voor uw systeem geldig zijn. Wellicht zijn de adressen DFH en DEH anders en als u pech hebt is de hele routine anders.
Ga tevens na of de stapel aan het begin van een programma geïnitialiseerd moet worden; normaal gesproken doet het besturingssysteem (operating system) dat voor u.

Opmerking-2.
De ASCII-codes 00H tot en met 1FH zijn niet de codes voor bepaalde tekens (zoals letters, cijfers of leestekens), maar codes voor bepaalde besturingsopdrachten zoals bijvoorbeeld CR (ga naar het begin van de regel), BS (zet de cursor één positie terug) of BEL (luidt de bel). Het 'afdrukken' van een dergelijke besturingscode (met de OUT-opdracht) zal tot gevolg hebben dat de processor de opdracht met die code uitvoert. De betekenis van deze besturingsopdrachten (ASCII-codes 00H t/m 1FH) vindt u beschreven onder de ASCII-tabel op de bij dit boek behorende inlegkaart.

4 Onvoorwaardelijke Sprongopdracht en Invoer via het Toetsenbord

4.1 Onvoorwaardelijke sprongopdracht

In de tot nu toe gegeven Z80 programma's worden de opdrachten sequentieel (de één na de ander) vanaf de eerste tot en met de laatste uitgevoerd. Het is echter mogelijk, om niet te zeggen gebruikelijk, dat om bepaalde redenen van deze sequentiële uitvoering afgeweken moet worden. We kunnen de volgorde van het uitvoeren van programma-instructies wijzigen door het opnemen van sprongopdrachten (jumps).

We kennen onvoorwaardelijke en voorwaardelijke sprongopdrachten. (unconditional en conditional jumps). De voorwaardelijke sprongopdrachten, waarbij alleen gesprongen wordt indien aan een bepaalde voorwaarde voldaan is, komen in de volgende paragraaf aan de orde.

De onvoorwaardelijke sprongopdracht heeft de volgende vorm:

```
JP  nn
```

Hierin is JP (JumP) de operatie en nn is het adres van de geheugenplaats waarnaar gesprongen moet worden. Dit sprongadres is gewoonlijk de **label** van een bepaalde instructie uit het programma en niet het **absolute adres** van die instructie.

Met zo'n onvoorwaardelijke sprongopdracht kunnen we, als het tenminste een sprong 'terug' in het programma is, alleen zogeheten eindloze lussen (indefinite loops) programmeren. Kijk maar naar het volgende programmavoorbeeld:

```
; Programma 4.1 eindloze lus
        -           ; beginopdrachten
        -
        -
LUS:    -           ; *
        -           ; * deze instructies worden steeds herhaald
        -           ; *
        JP LUS      ; spring naar LUS
```

Eerst worden de beginopdrachten (aangegeven door drie streepjes) uitgevoerd, daarna de instructies met een *. Deze vormen de lusopdrachten.

De lus begint met de instructie met het label LUS en eindigt met de onvoorwaardelijke sprongopdracht JP LUS. Er is geen enkele manier om het uitvoeren van de met een * gemerkte opdrachten te stoppen anders dan het indrukken van de RESET toets of erger nog door het geheel uitzetten van de computer. U zult straks zien dat een 'eindloze' lus in een 'tijdelijke' lus verandert als we in plaats van een onvoorwaardelijke sprongopdracht een voorwaardelijke sprongopdracht gebruiken.

Opgave 4.1
Schrijf een aantal instructies die een 'eindloze' reeks sterretjes op het beeldscherm afdrukt.

Met de JP instructie kunnen we in principe naar elke byte in een 64K geheugen springen. Afgezien van het feit dat dit erg ongewenst, dan wel levensgevaarlijk kan zijn, is het zo dat de meeste sprongen gemaakt worden naar geheugenposities niet ver van de plaats vanwaaruit de sprong gemaakt wordt; immers de sprong zal altijd binnen één programma gemaakt worden en vaak nog binnen één programmamodule! Het is dan ook niet verwonderlijk dat er een JR instructie bestaat, de zogenaamde relatieve sprongopdracht (Jump Relative). Het adresgedeelte van deze instructie bevat nu niet een adres (of label) waarnaar gesprongen moet worden, maar een aantal bytes waar 'overheen' gesprongen moet worden. Dit aantal is relatief ten opzichte van de JR instructie zelf: het is het aantal bytes tussen de JR instructie en de instructie waar in feite naar toe gesprongen wordt.

Opgave 4.2
Ga met behulp van tabel C.10 uit Appendix C na hoeveel bytes de JP en JR instructies zelf in beslag nemen.

4.2 Invoer via toetsenbord

In het vorige hoofdstuk hebben we u een subroutine COUT gegeven voor het afdrukken van een teken op het beeldscherm, waarbij de ASCII code van het desbetreffende teken zich in de accumulator moest bevinden. Nu krijgt u een subroutine CIN, die een teken,

ingetoetst via het toetsenbord, accepteert en de code ervan overbrengt naar de accumulator.

```
; Programma 4.2
;
; subroutine voor het invoeren van een teken via het toetsenbord
;     - na terugkeer uit routine bevat de accumulator de code
;       van het ingetoetste teken.
;
CIN:    IN A,(ØDFH)     ; is toets
        BIT 1,A         ; ingedrukt?
        JR Z,CIN        ; nee -probeer opnieuw
        IN A,(ØDEH)     ; ja -voer teken in
        RET             ; keer terug naar aanroepend programma
```

Als de programmabesturing weer uit de subroutine terugkeert naar de plaats van waar de routine werd aangeroepen, zal de accumulator de ASCII code van het ingetoetste teken bevatten.
U kunt de routine aanroepen met CALL CIN.

Het is niet altijd zo dat een teken dat we intoetsen automatisch op het beeldscherm wordt ingedrukt. Gebeurt dit zogeheten 'echoën' niet automatisch, dan zullen we een ingevoerd teken zelf direct op het beeldscherm moeten laten afdrukken.

Opgave 4.3
Schrijf een subroutine CINEKO, die een ingetoetst teken accepteert en direct op het beeldscherm afdrukt. Bedenk hierbij dat u vanuit een subroutine een andere subroutine mag aanroepen!
Ga ook voor de routine CIN na of deze voor uw computersysteem juist geformuleerd is. Probeer er tevens achter te komen of een ingetoetst teken automatisch 'geëchood' wordt.

Als een aantal opdrachten in één programma verschillende keren als een groep opdrachten voorkomt of als een aantal programma's dezelfde groep opdrachten bevat, is het gewenst dat zo'n groep opdrachten in een subroutine wordt opgenomen. In het programma, dat in hoofdstuk 3 gemaakt moest worden, moet een 'terug-wagen-teken' (Carriage Return, CR) en een 'nieuwe-regel-teken' (Line Feed, LF) op het scherm afgedrukt worden. Het zal duidelijk zijn dat dit 'naar het begin van een nieuwe regel gaan' veel vaker nodig zal zijn. Het is wellicht raadzaam nu reeds een subroutine CRLF te maken die deze handelingen voor u verricht. Als u in een programma dan naar het begin van een nieuwe regel moet, is de instructie CALL CRLF voldoende om dit te bewerkstelligen.

Deze programmeeraanpak noemt men modulair programmeren

of modulair ontwerp. Dit is iets wat meer weggelegd is voor nog hogere programmeertalen (COBOL, Pascal), maar het kan toch ook bij assembleertaalprogramma's van belang zijn voor het leesbaar maken van de programma's, voor de overzichtelijkheid ervan en vooral ook voor het onderhoud ervan.

4.3 De 'waarde' en de 'code' van een teken

Als we in instructies met cijfers werken, moeten we ervoor waken niet de 'waarde' van een cijfer te verwarren met een cijfer als 'teken' (de code van een cijfer). Dit zijn twee totaal verschillende zaken!

De waarden van de cijfers 0 tot en met 9 worden in de registers en geheugenplaatsen opgeslagen als 00H tot en met 09H.

De codes van de tekens 0 tot en met 9 worden weergegeven door de hexadecimale getallen 30H tot en met 39H.

De cijfers 0 t/m 9 worden 'echt' als cijfer gebruikt bij het uitvoeren van berekeningen, terwijl de cijfers 0 t/m 9 als tekens behandeld worden bij het in- en uitvoeren (inlezen en afdrukken) van cijfers. Zo moet de code van een cijfer, dat als teken wordt ingevoerd, eerst worden omgezet (geconverteerd) in de waarde van het cijfer, voordat ermee gerekend kan worden. Dit geldt andersom als we bijvoorbeeld uitkomsten van berekeningen willen afdrukken.

Opgave 4.4
Geef de inhoud van de accumulator na het uitvoeren van elk van de volgende twee instructies:

```
LD  A,'7'        en        LD  A,8
```

4.4 De EQU pseudo-instructie

De EQU pseudo-instructie is een ten onrechte weinig gebruikte pseudo-instructie. We kunnen met EQU een constante een naam geven door de waarde van die constante aan een label toe te kennen. Hierdoor wordt een programma leesbaarder en is het gemakkelijker een programma aan bepaalde situaties aan te passen.

We geven een voorbeeld. Als we de accumulator willen laden met een carriage return (CR), is de instructie LD A,CR begrijpelijker dan de instructie LD A,ØDH. Welnu, dit kan met behulp van EQU. Kijk maar:

```
CR:  EQU ØDH
     -
     -
     LD A,CR
```

CR is nu de naam van een label met als waarde ØDH (de hexadecimale ASCII-code voor een Carriage Return). EQU is een afkorting van EQUate (gelijk maken aan) en deze RQU's worden doorgaans aan het begin van een programma opgenomen. We mogen maar één keer een waarde aan hetzelfde label toekennen, maar we mogen zo vaak als we willen naar de label verwijzen.

Een ander handig gebruik van de EQU pseudo-instructie is het geven van namen en adressen van de beeldscherm- en toetsenbord-status-bytes (display and keybord status) en van gewone geheugenplaatsen waar gegevens zijn opgeslagen.

Op p.33 zien we een programma met in de IN- en OUT-opdrachten niet de adressen van de bytes, die de status van toetsenbord en beeldscherm bevatten, maar de namen van labels. Deze labels worden met EQU instructies toegekend aan de absolute adressen van de status-bytes. Voor de goede orde nog dit: ØDFH (zie eerste regel van het programma) is het adres ØDF Hexadecimaal (nul D F hexadecimaal) van de byte waarin zich de status van het toetsenbord bevindt.

Mocht het nu nodig zijn de adressen van de status-byte (ØDFH) en van de gegevensbyte (label DATA, adres ØDEH) te wijzigen, dan hoeven alleen de EQU instructies veranderd te worden, terwijl we zonder EQU's dit in al die instructies waarin we naar deze adressen verwijzen zouden moeten doen. Na het eventueel wijzigen van de adressen in de EQU instructies hoeven we het programma alleen nog maar opnieuw te assembleren en klaar is kees.

```
DATA:   EQU  ØDEH
STATUS: EQU  ØDFH
        -
        -
CIN:    IN   A,(STATUS)
        -
        IN   A,(DATA)
        -
COUT:   PUSH AF
INSTAT: IN   A,(STATUS)
        -
        OUT  (DATA),A
        -
```

Regel zou moeten zijn dat alle getallen die als code of als adres in een programma gebruikt worden, via een EQU opdracht aan een label worden toegekend. Dit geldt niet voor getallen die echt als 'getal' gebruikt worden.

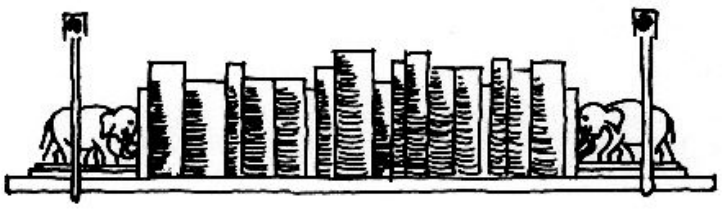
4.5 Een programmeeropdracht

Schrijf een programma dat steeds twee decimale cijfers via het toetsenbord inleest en de som van de cijfers afdrukt. U mag er van uitgaan dat deze som nooit groter is dan 9. Als het programma draait moet de uitvoer er bijvoorbeeld zo uitzien:

```
2 + 2 = 4
5 + 3 = 8
4 + 1 = 5
```

U moet zelf zorg dragen voor het afdrukken van + en = tekens.

Als u in verschillende programma's van dezelfde subroutine gebruik zoudt willen maken, is het niet nodig zo'n subroutine telkens opnieuw in een programma op te nemen, dat wil zeggen opnieuw te programmeren. Elke subroutine kan als een bestand op



een achtergrondgeheugen opgeslagen worden. Door middel van een opdracht als *ATTACH subroutine-bestandsnaam of iets dergelijks kunt u dan als het ware de subroutine aan uw programma koppelen. Hoe dat precies moet verschilt van computer tot computer. Zorg dat u er achter komt hoe dat in zijn werk gaat bij het systeem dat u gebruikt, dat bespaart u namelijk veel moeite en tijd! U kunt zo uw eigen programma- (en subroutine-) bibliotheek opbouwen.

5 Vlaggen, Voorwaardelijke Sprongopdracht en de CP Instructie

5.1 Het vlag-register

De Z80 heeft een 8-bit vlag-register, ook wel status-register genoemd, dat gebruikt wordt voor het opslaan van informatie over de laatst uitgevoerde instructie. In feite worden slechts zes van de acht statusbits hiervoor gebruikt.

S	Z	X	H	X	P/V	N	C

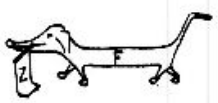
S - Sign flag (teken-vlag)
Z - Zero flag (nul-vlag)
H - Half-carry flag (halve-overdracht-vlag)
P/V - Parity/Overflow flag (pariteit/overflow-vlag)
N - Add/Subtract flag (optel/aftrek-vlag)
C - Carry flag (overdracht-vlag)
X - Ongebruikte bits

We zullen de Engelse benamingen voor de vlaggen hanteren, daar de eerste letter van zo'n vlag nogal eens in een instructie voorkomt.

De SIGN vlag (teken statusbit) wordt op 1 gezet als de uitkomst van een instructie negatief is, anders is deze vlag 0. Zo zal na het uitvoeren van

```
LD   A,23
SUB  56
```

de accumulator -33 bevatten en zal de sign vlag op 1 gezet zijn.



De ZERO vlag (nul statusbit) wordt op 1 gezet als de uitkomst van een instructie nul is; zo niet, dan is dit bit gelijk aan nul. Zo zal na het uitvoeren van de twee bovenstaande instructies de zero vlag weer op nul staan.

Voorlopig hebben we aan de S en Z vlaggen voldoende. De H, P/V, N en C vlaggen komen later aan bod.

Het is niet zo dat elke instructie invloed heeft op elk statusbit in het vlagregister. Zo heeft de LD instructie op geen van de statusbits enige invloed. U kunt zelf nagaan wat de invloed van een bepaalde instructie op een bepaalde vlag is door in de derde kolom van de tabellen in Appendix C te kijken. U kunt dan bijvoorbeeld zien dat de S (sign) en Z (zero) vlaggen hoofdzakelijk beïnvloed worden als er gerekend wordt en door de schuif- (shift), roteer- en bit-instructies.

Opgave 5.1
Geef de inhoud van de accumulator en de 0 of 1 status van de S en Z vlaggen na het uitvoeren van elk van de volgende instructies

```
LD   A,120
SUB  122
LD   B,A
SUB  B
ADD  A,70
NEG
```

We zullen nu overgaan naar de voorwaardelijke spronginstructies die gebruik maken van de statusbits uit het vlagregister.

5.2 Voorwaardelijke spronginstructies

Een voorwaardelijke spronginstructie heeft alleen een sprong naar een instructie elders in het programma tot gevolg als aan een bepaalde voorwaarde wordt voldaan. Is aan de voorwaarde niet voldaan, dan wordt gewoon de instructie volgend op de spronginstructie uitgevoerd.

Of er gesprongen wordt (de voorwaarde) hangt af van de status (0 of 1) van één bit uit het vlagregister (in onderstaand programma is dit de Z-vlag, het Z-bit). Met een voorwaardelijke spronginstructie kunnen we dus de volgorde van het uitvoeren van instructies beïnvloeden. Hier volgt een programmavoorbeeld.

```
; Programma 5.1 voorwaardelijke spronginstructie
;
        LD  A,(X)
        SUB 10          ; bereken X-10
        JP  Z,GELIJK
        LD  A,1         ; X ≠ 10
        JP  GADOOR
GELIJK: LD  A,0         ; X = 10
GADOOR: -
        -
X:      DEFB 25
```

Dit programma zet de accumulator op 0 als X gelijk is aan 10, anders wordt de inhoud van de accumulator 1 gemaakt. De SUB instructie berekent de waarde van X-10 (in de accumulator) en zet de sign en zero vlaggen afhankelijk van de uitkomst van de berekening. Bij X = 25 worden zowel de S-vlag als de Z-vlag op nul gezet.

De voorwaardelijke spronginstructie JP Z,GELIJK dwingt de computer te springen naar de met 'GELIJK' gelabelde instructie als de Z-vlag 1 is. Is de Z-vlag nul dan wordt gewoon de instructie LD A,1 uitgevoerd. Als X = 25, dan is X-10 gelijk aan 15, hetgeen een positieve uitkomst in de accumulator oplevert; dus wordt de zero vlag op nul gezet; dus wordt er als gevolg van het uitvoeren van de JP Z,GELIJK instructie niet gesprongen (dit gebeurt alleen als Z = 1). Na het uitvoeren van LD A,1 springt het programma, als gevolg van de onvoorwaardelijke spronginstructie JP GADOOR naar de instructie met de label GADOOR.

Opgave 5.2
Geef in programma 5.1 de volgorde aan van de uitgevoerde instructies indien de laatste instructie er zo had uitgezien:
X: DEFB 10

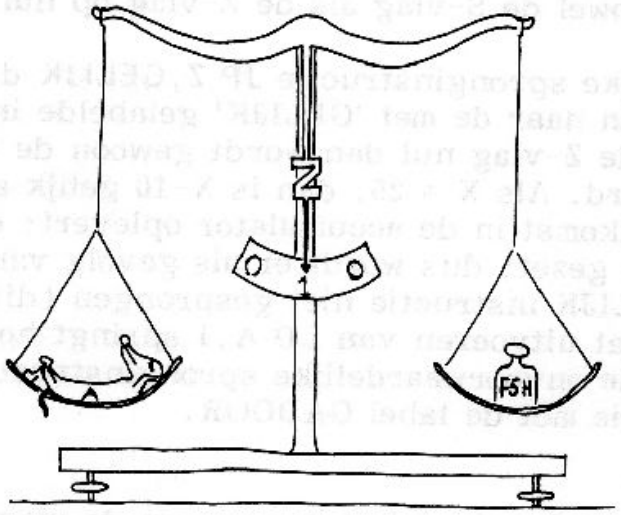
Met behulp van de status van de S en Z bits kunnen vier verschillende voorwaardelijke spronginstructies gebruikt worden. Deze zijn:

uitkomst	Nul	⇒ Z=1 ; spring als Z-bit is 1	- JP Z,GELIJK
	niet Nul	⇒ Z=0 ; spring als Z-bit is 0	- JP NZ,ONGEL
uitkomst	Negatief (Min)	⇒ S=1 ; spring als S-bit is 1	- JP M,NEG
	Positief (Plus)	⇒ S=1 ; spring als S-bit is 0	- JP P,POS

Opgave 5.3
Schrijf een stukje programma dat de letter N op het display afdrukt indien de som van twee getallen, het éne zit in het B register en het andere in het C register, negatief is. Is de som nul dan moet de letter Z afgedrukt worden en bij een positief resultaat de letter P.

De JP Z en JP NZ instructies hebben 'broertjes' (of 'zusjes') in de vorm van JR Z en JR NZ, de relatieve voorwaardelijke sprongin-structies. Dit geldt echter niet voor de JP M en JP P instructies.

5.3 De COMPARE (vergelijk) instructie



de ComPare-balans

De Z80 compare instructie is een erg handige instructie. We kunnen met deze instructie namelijk de inhoud van de accumulator vergelijken met een andere waarde zonder dat dit invloed heeft op de inhoud van de accumulator zelf. (U weet nu dat bijvoorbeeld de ADD en SUB instructies wel de inhoud van de accumulator kunnen veranderen.)

De ComPare (CP) instructie werkt als volgt. De operand, een register of een 8-bit waarde, die we in de CP instructie opgeven, wordt van de inhoud van de accumulator afgetrokken. Afhankelijk van het resultaat worden de 'statusbits' al of niet 'gezet'. De inhoud van de

accumulator blijft onaangetast en de uitkomst van de aftrekking gaat gewoon verloren. De CP instructie heeft dus alleen het al of niet zetten van statusbits uit het vlagregister tot gevolg.

De algemene vorm van de ComPare instructie is

```
CP  s
```

waarin s de operand is die met de inhoud van de accumulator vergeleken wordt; s kan data zijn, een enkelvoudig register (A t/m H) of een registerpaar (HL) en nog meer.

Het meest directe gebruik van de CP-instructie is het bepalen of de accumulator al dan niet een bepaalde waarde bevat. Zo zullen de instructies

```
CP  60
JR  Z,ZESTIG
```

tot gevolg hebben dat (na de CP instructie) de Z vlag op 1 gezet wordt als de accumulator 60 bevat en op 0 als dit niet het geval is, terwijl (na de JR instructie) naar de instructie met label ZESTIG gesprongen wordt als de inhoud van de accumulator inderdaad 60 is (Z = 1).

Opgave 5.4
Schrijf een stukje programma, dat naar een met MINDER gelabelde instructie springt, indien de waarde van de variabele TELLER minder dan 100 is, en dat naar een instructie met label GELIJK springt indien de waarde van TELLER nul is, en dat tenslotte naar een met GROTER gelabelde instructie springt als TELLER groter is dan 100.

Zoals u weet zijn er instructies die geen invloed hebben op de bits in het vlagregister. Met name de LD instructie is er zo een. Willen we nu na zo'n LD instructie toch iets over de status van de accumulator (is de inhoud nul, ongelijk nul, positief, negatief?) te weten komen, dan kunnen we direct na de LD instructie, waardoor de accumulator geladen wordt, een CP 0 instructie opnemen. Een voorbeeld:

```
LD  A,(TEMP)
CP  0
```

De opdracht LD A,(TEMP) laadt de accumulator met de inhoud van de geheugenplaats die gelabeld is met TEMP. De LD instructie verandert niets ('not a bit') in het vlagregister. Het vlagregister zegt dus niets over de 'nieuwe' inhoud van de accumulator. De CP 0 instructie trekt 0 (niets) van de inhoud van de accumulator af (resultaat is dus de inhoud van de accumulator), maar zet wel de S

en Z-vlaggen afhankelijk van het resultaat van de aftrekking, met andere woorden afhankelijk van wat we zojuist (met de LD opdracht) in de accumulator geladen hebben. De S en Z vlaggen vertellen nu wel iets over de nieuwe inhoud van de accumulator!

5.4 Het beëindigen van een voorwaardelijke lus

We gaan nu bekijken met welke instructies we het uitvoeren van een programmalus kunnen afbreken. We zullen in de komende hoofdstukken verschillende instructies leren kennen waarmee we dit kunnen bewerkstelligen. In dit hoofdstuk zullen we zien hoe we voor dit doel de CP instructie kunnen gebruiken.

In het onderstaande programma 5.2 ziet u een lus waarmee we tekens kunnen invoeren. We kunnen het invoeren (de lus) beëindigen door een spatie (dat is ook een teken!) in te tikken.

```
; Programma 5.2
;
NOGEEN: CALL CINEKO
        CP   ' '
        JP   NZ,NOGEEN
VERDER: -
        -
```

De drie instructies CALL, CP en JP vormen de lus en worden steeds opnieuw uitgevoerd tot het moment waarop een spatie (' ') ingevoerd wordt. Als dit gebeurt zet de CP instructie de Z-vlag op 1. Dan is NZ nul en zal de JP instructie niet naar NOGEEN springen maar zal het programma verder gaan met de instructie, die het label VERDER heeft meegekregen.

5.5 Een programmeeropdracht

Schrijf een subroutine waarmee u een bepaald teken kunt karakteriseren (is het een cijfer, een kleine letter of hoofdletter uit het alfabet of is het een ander teken?).

Als de subroutine aangeroepen wordt bevindt zich de code van het te onderzoeken teken in de accumulator. De subroutine laat de accumulator ongemoeid, maar geeft in register B het volgende resultaat:

-1 als het een cijfer is,
 0 als het een kleine letter of een hoofdletter is èn
 1 als het een ander teken betreft.

U hebt hierbij de tabel met ASCII tekens uit Appendix D nodig!

Neem vervolgens deze subroutine op in een programma waarmee u herhaaldelijk een teken via het toetsenbord kunt intikken en dat bij elk teken eerst een spatie afdrukt gevolgd door een C, een L of een A voor respectievelijk een Cijfer, een Letter of een Ander teken. Na het afdrukken van een van deze letters moet er naar het begin van een nieuwe regel gegaan worden.

Het programma moet beëindigd worden als u op de RETURN-toets drukt!

6 Programmalussen en de Stapel

Een lus (loop) in een programma kunnen we op vele manieren construeren. We hebben al kennis gemaakt met een 'eindloze' lus (niet aanbevolen) en een lus die beëindigd wordt als aan een bepaalde voorwaarde voldaan is (conditional terminated loop).
In de loop van dit boek zullen we nog een aantal lusconstructies tegenkomen. In dit hoofdstuk bijten we de spits af met een, wat we zouden kunnen noemen, 'aftellende' lus (counting loop).

6.1 Aftellende lus

Een van de eenvoudigste programmalussen is die waarin een groep instructies een bepaald vast aantal malen wordt uitgevoerd. Onderstaand programma is hiervan een voorbeeld. De opdrachten tussen WEER en GADOOR worden precies tien keer uitgevoerd. Bij elke doorloop van de lus wordt een cijfer (digit) ingelezen en opgeteld bij de som van de reeds ingelezen cijfers.

```
;  Programma 6.1 invoeren en optellen van 10 cijfers
;
        LD   C,Ø          ; lopende som zit in C
        LD   B,1Ø         ; initialiseren van de lusteller
WEER:   CALL CINEKO       ; lees cijfer in
        SUB  3ØH          ; converteer teken in waarde
        ADD  A,C          ; tel cijfer op bij lopende som
        DJNZ WEER
GADOOR: LD   C,A
```

De instructies CALL CINEKO tot en met DJNZ WEER worden tien keer uitgevoerd. Register B, dat dienst doet als lusteller, wordt eerst geladen met het aantal malen dat de lus doorlopen moet worden (LD B,1Ø). De DJNZ (Decrement and Jump on Non Zero) instructie zorgt ervoor dat de inhoud van register B met één verlaagd wordt (het aftellen dus) en dat, indien de inhoud van B ongelijk nul is,

gesprongen wordt naar de instructie met label WEER. Mocht B nul worden (na 10 'doorlopen') dan zal het programma verder gaan met de instructie die met GADOOR gelabeld is; dus met de instructie volgend op de DJNZ instructie.

Opgave 6.1
Wat is de grootste waarde, die we in programma 6.1 aan de teller B kunnen toekennen?

Het getuigt doorgaans niet van een goede programmeerstijl om steeds voor elk speciaal geval, zoals programma 6.1, een programma te schrijven. Een programma moet toepasbaar zijn in meer dan één situatie. Voor programma 6.1 betekent dit dat we het aantal malen dat de lus doorlopen wordt niet vastprikken op tien maar variabel maken. Dit zou gerealiseerd kunnen worden door de lus in een subroutine op te nemen en de lusteller als parameter op te nemen. De waarde van de parameter (de inhoud van B) zouden we via het toetsenbord kunnen invoeren. Ook is het denkbaar dat deze lusteller in het programma zelf berekend wordt, maar dan wel voor het aanroepen van de 'lussubroutine'.

Opgave 6.2
Schrijf een programma dat een cijfer n inleest van het toetsenbord en dat vervolgens n sterretjes afdrukt.

Behalve register B kunnen ook andere registers of geheugenplaatsen als teller dienst doen. Dat de keuze op B gevallen is komt door de DJNZ instructie. Deze instructie werkt alleen op het B register. DJNZ impliceert domweg het met één verlagen van het B register en het al dan niet springen naar een opgegeven label.

Zouden we bijvoorbeeld in programma 6.1 register C als teller willen gebruiken, dan krijgen we de volgende situatie:

```
        -
        LD  C,beginwaarde
WEER:   -
        -
        DEC C
        JR  NZ,WEER
GADOOR: -
```

Voor het verlagen en het testen van de inhoud van register C zijn nu twee instructies nodig, hetgeen betekent dat de uitvoering van het programma meer tijd zal kosten. De lezer zal inzien dat DEC B gevolgd door JR NZ,WEER identiek is met DNJZ WEER!

Het komt dikwijls voor dat de lusteller zelf binnen in de lus gebruikt wordt. Een voorbeeld hiervan is een programma, waarin we de getallen 1 tot en met N bij elkaar optellen.

```
; Programma 6.2 optellen van de getallen 1 t/m N
;
N:      EQU  11
        LD   A,Ø          ; som wordt nul
        LD   B,N
TELOP:  ADD  A,B          ; tel n bij som op
        DJNZ TELOP
```

De teller B wordt gebruikt als 'afteller', maar tevens wordt de waarde van B telkens bij de som (A) opgeteld. De waarde van de lusteller wordt dus in een opdracht (ADD A,B) in de lus zelf gebruikt. Na afloop bevat de accumulator de gewenste optelling. Denk erom dat de accumulator eerst op nul gezet moet worden.

In de hierboven geschetste situatie is het niet altijd wenselijk (lees 'is het ongewenst') dat tot nul wordt afgeteld. We kunnen dan niet van de DJNZ of JR instructie gebruik maken, maar we zullen dan de CP instructie (ComPare) moeten gebruiken. Aangezien de CP instructie alleen op de accumulator werkt moet in zo'n geval de accumulator als teller gebruikt worden.

Opgave 6.3
Schrijf een programma dat de cijfers 9 tot en met 0 in aflopende volgorde afdrukt.

6.2 Invoeren van getallen

Tot nu toe hebben we alleen ééncijferige getallen (één teken dus) via het toetsenbord ingevoerd. Het invoeren van getallen bestaande uit meer dan één cijfer is lastiger dan je zou denken.

Stel we willen het getal 123 invoeren. We moeten dit doen door het invoeren van drie tekens (het teken 1, het teken 2 en het teken 3). Dit kan door het driemaal aanroepen van de subroutine CINEKO, waarna het programma de beschikking heeft over de codes van de drie cijfers. De vraag is nu hoe deze drie codes geconverteerd moeten worden tot de 8-bit code, waarmee de waarde van het getal 123 in een register of in een geheugenplaats wordt opgeslagen.

Zeker is het dat de code van elk cijfer geconverteerd moet worden in de waarde van dat cijfer. Vervolgens moet het eerst ingevoerde

cijfer met 100 vermenigvuldigd worden en het tweede met 10. Tot slot moeten deze twee waarden worden opgeteld bij het laatst ingevoerde cijfer om de waarde van het getal te krijgen. Voor het getal 123 moet dus de volgende berekening worden uitgevoerd:

1 × 100 + 2 × 10 + 3.

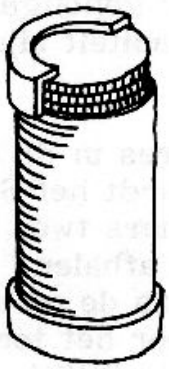
Een efficiëntere methode (algoritme) om hetzelfde te bereiken is het volgende:

((eerste cijfer × 10) + tweede cijfer) × 10 + laatste cijfer.

Dit algoritme (rekenvoorschrift) kan gemakkelijk uitgebreid worden voor het invoeren van een getal van n cijfers.

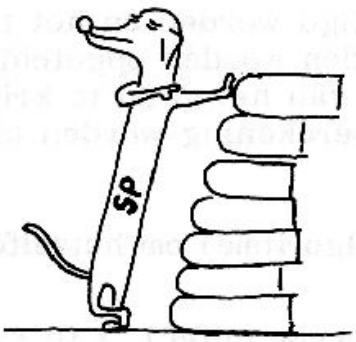
6.3 De stapel

De (Z80) stapel is een stukje geheugen dat we kunnen gebruiken voor tijdelijke opslag van gegevens. De stapel (een aantal opvolgende geheugenplaatsen) werkt volgens het LIFO principe. LIFO betekent Last-In-First-Out. We kunnen dus alleen gegevens boven op de stapel leggen en alleen het bovenste gegeven van de stapel afhalen. Volgens dit principe werken ook die, bij u wellicht bekende, munthoudertjes.



Last In First Out

De Z80 stapel werkt met elementen van twee bytes. Als we iets op de stapel kwijt willen, kost ons dat dus twee geheugenplaatsen. Omdat het laatst aan de stapel toegevoegde element (de top van de stapel) eigenlijk het enige is (LIFO-principe) dat van belang is, bevat de CPU een speciaal register, waarvan de inhoud verwijst naar het bovenste element van de stapel.

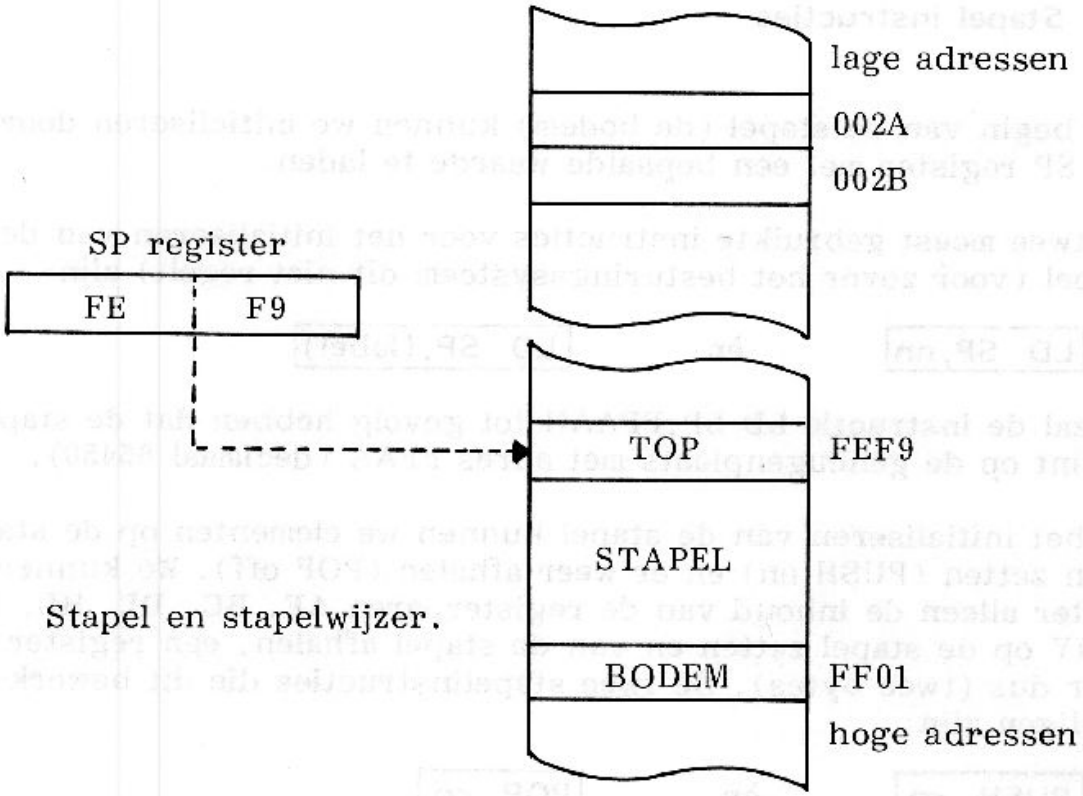


Dit register heet de Stack Pointer (stapelwijzer), kortweg SP register genoemd. Behalve een top heeft een stapel ook een 'bodem'. Het adres van de bodem (het eerste element van de stapel) kan met bepaalde instructies gedefinieerd worden. Er kan zelfs meer dan één stapel zijn. Doorgaans is er echter één stapel in een systeem en wordt het beginadres van de stapel door het operating system bepaald.

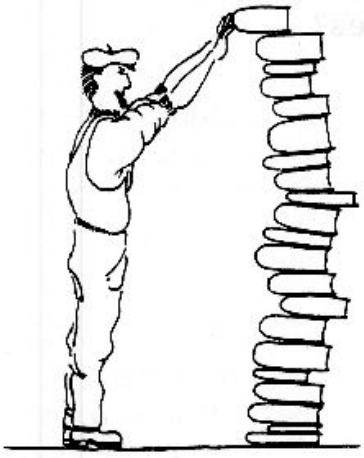
In de tekening op p.47 is te zien dat de top van de stapel een 'lager' adres heeft dan de bodem. Dikwijls is het beginadres van de stapel één van de hoogste adressen uit het geheugen. Het zal duidelijk zijn dat het SP register twee bytes (16 bits) groot is. De SP wijst immers naar de geheugenplaats waar zich de top van de stapel bevindt. Dit betekent dat de SP een 'hoog' geheugenadres bevat.. Als we even uitgaan van 64K geheugencapaciteit zijn er twee bytes nodig om een adres te specificeren.

Hoe 'hoger' de stapel des te 'lager' het adres in de stack pointer. Als we een element op de stapel zetten, wordt het SP register met 2 verminderd. Elk stapelelement neemt immers twee bytes in beslag. Als we het bovenste element van de stapel afhalen, wordt het SP register met twee opgehoogd en geeft het zo de nieuwe top van de stapel aan. De Engelstalige benamingen voor het toevoegen aan en afhalen van de stapel zijn respectievelijk 'to PUSH on' en 'to POP off'.

De stapel wordt hoofdzakelijk gebruikt voor tijdelijke opslag van gegevens en adressen en bij het werken met subroutines. Het subroutinemechanisme komt in het volgende hoofdstuk aan de orde. In de volgende paragrafen bekijken we de eerstgenoemde toepassing van de stapel.



Stapel en stapelwijzer.



PUSH

POP

6.4 Stapel instructies

Het begin van de stapel (de bodem) kunnen we initialiseren door het SP register met een bepaalde waarde te laden.

De twee meest gebruikte instructies voor het initialiseren van de stapel (voor zover het besturingssysteem dit niet regelt) zijn

`LD SP,nn` èn `LD SP,(label)`

Zo zal de instructie LD SP,FFAAH tot gevolg hebben dat de stapel begint op de geheugenplaats met adres FFAA (decimaal 65450).

Na het initialiseren van de stapel kunnen we elementen op de stapel gaan zetten (PUSH on) en er weer afhalen (POP off). We kunnen echter alleen de inhoud van de registerparen AF, BC, DE, HL, IX en IY op de stapel zetten en van de stapel afhalen, een registerpaar dus (twee bytes). De twee stapelinstructies die dit bewerkstelligen zijn:

`PUSH rp` èn `POP rp`

rp is één van bovengenoemde register-paren. Na een PUSH zal het SP register met twee verminderd zijn; na een POP zal het SP register met twee verhoogd zijn. SP wijst altijd naar de top van de stapel. Dus:

`PUSH ⇒ SP → SP - 2` èn `POP ⇒ SP → SP + 2`

Opgave 6.4
Wat is de inhoud van de registers A, B, C, D, E en SP na het uitvoeren van de volgende reeks instructies?

```
LD    A,0AH
LD    B,0BH
LD    C,0CH
LD    D,0DH
LD    E,0EH
LD    SP,16383
PUSH  AF
PUSH  BC
PUSH  DE
POP   BC
POP   DE
```

Een handige stapelinstructie is EX (SP),HL waarmee we de inhoud van de top van de stapel kunnen verwisselen met de inhoud van het

HL registerpaar. Dit verwisselen kunnen we ook doen met de IX en IY indexregisters. De opdrachten hiervoor zijn respectievelijk EX (SP),IX en EX (SP),IY.

6.5 Het redden en herstellen van registers

Wanneer een subroutine van één of meer CPU registers gebruik wil maken moet zo'n subroutine deze registers achterlaten in de toestand waarin de subroutine ze heeft aangetroffen, met andere woorden:

een subroutine mag de inhoud van registers, anders dan die welke bedoeld zijn voor de parameteroverdracht tussen subroutine en aanroepend programma, niet veranderen.

Wordt een subroutine aangeroepen, dan moet dus òf het aanroepende programma òf de subroutine zelf ervoor zorgen dat de inhoud van bepaalde registers gered wordt. Evenzo moet bij terugkeer uit de subroutine naar het aanroepende programma ervoor gezorgd worden dat de oude inhoud van die registers hersteld wordt.

De eenvoudigste manier om registers te 'redden' en te 'herstellen' is gebruik te maken van de stapel. Stel dat een subroutine de registers B, C en D wil gebruiken. Voordat de eigenlijke instructies in die subroutine worden uitgevoerd, moeten deze registers eerst gered worden. In onderstaand voorbeeld gebeurt dit in de eerste twee instructies van de subroutine (denk erom: alleen registerparen kunnen op de stapel gezet worden!).

```
SUBIN:  PUSH BC
        PUSH DE
        -
        -
        POP  DE
        POP  BC
        RET
```

Vlak voor de RETurn uit de subroutine worden de registers B, C en D weer geladen met hun 'oude' inhoud. U ziet in bovenstaand voorbeeld dat de registerparen precies in omgekeerde volgorde gePOPt worden dan waarin ze gePUSHt zijn. Dit is een gevolg van de LIFO-structuur van de stapel.
Bovendien moeten we er goed voor zorgen dat alle registers hersteld zijn voor uitvoering van de RET opdracht, de terugkeer naar het aanroepende programma.

Mocht de subroutine zelf de door hem gebruikte registers niet redden, dan zal via commentaarregels aan het begin van de subroutine duidelijk gemaakt moeten worden welke registers de routine gebruikt.

6.6 Een programmeeropdracht

Herhaald optellen is een nogal omslachtige manier om een vermenigvuldiging uit te voeren. Zo is 3×4 hetzelfde als 3+3+3+3. In het algemeen is p×q hetzelfde als q-1 maal p bij zichzelf optellen.

Schrijf een subroutine VERM die de inhoud van register B vermenigvuldigt met de inhoud van register C en die het resultaat van deze vermenigvuldiging in de accumulator achterlaat. De subroutine mag er van uit gaan dat dit produkt klein genoeg is om in de accumulator te worden opgeslagen.

Schrijf ook een subroutine GETIN die een decimaal getal, zonder teken, bestaande uit een willekeurig aantal cijfers, inleest. Het getal wordt afgesloten met een ander teken dan een cijfer en er mogen spaties voor staan, die door de subroutine genegeerd moeten worden. Bij terugkeer uit de subroutine moet de accumulator het ingelezen getal bevatten en het register B moet het aantal cijfers in dit getal bevatten.

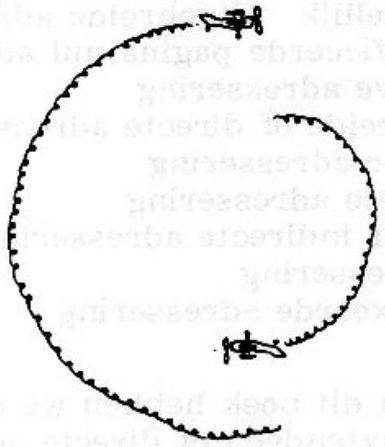
Gebruik deze twee subroutines en subroutines uit eerder gemaakte programma's om een (hoofd)programma te schrijven dat herhaald (met echo!) een decimaal getal inleest. De getallen worden afgesloten met een komma. Uitgevoerd moet worden het woord OK als het getal twee cijfers (met eventueel een nul aan het begin!) bevat anders dan de combinatie 99. Wordt hieraan niet voldaan dan moet het aantal cijfers in dat getal uitgevoerd worden.

Een voordeel van modulair programmeren is dat een module, bijvoorbeeld een subroutine in een assembleertaalprogramma, vervangen kan worden door een andere module zonder dat de rest van het programma hierdoor beïnvloed wordt. Voorwaarde is dan natuurlijk wel dat de nieuwe module precies dezelfde functie vervult als de oude en dat de parameteroverdracht op dezelfde wijze uitgevoerd wordt. Als u bijvoorbeeld straks een slimmere subroutine VERM maakt, kunt u deze als vervanging gebruiken van de, wellicht niet al te efficiente, subroutine VERM die u zojuist hebt gemaakt. Het programma wordt hierdoor op geen enkele wijze beïnvloed! De naam van de nieuwe module moet dan wel identiek zijn aan de naam van de oude routine.

7 Geneste Lussen en Adresseermethoden

7.1 Geneste lussen

We kunnen binnen een programmalus een tweede lus opnemen. We spreken dan van geneste lussen. Een lus binnen een lus dus. Dit nesten hoeft niet tot twee lussen beperkt te blijven. We kunnen binnen de nieuwe lus weer een lus opnemen, enzovoorts. Als we een situatie met twee geneste lussen hebben, spreken we ook wel over de buitenste en de binnenste lus. Het hiernavolgende programma bevat twee geneste lussen.



geneste loops

```
; Programma 7.1 drukt vier regels met 6 *'s af
;
        LD   C,4            ; initialiseer lijnenafteller
LUSC:   CALL CRLF           ; begin 'lijnenlus C'
        LD   A,'*'
        LD   B,6            ; initialiseer sterrenafteller
LUSB:   CALL COUT           ; begin 'sterrenlus B'
        DJNZ LUSB           ; eind 'lus B'
        DEC  C
        JP   NZ,LUSC        ; eind 'lus C'
        HALT
```

Dit programma levert als uitvoer vier regels met zes sterretjes. De buitenste lus gebruikt register C om het aantal lijnen (af) te tellen en de binnenste lus gebruikt register B voor het bijhouden van het aantal sterretjes.

Opgave 7.1
Hoeveel maal worden in programma 7.1 de instructies LD A,'*', CALL COUT en DEC C uitgevoerd als we dit programma zouden draaien?

7.2 Onmiddellijk-uitgebreide en Register-indirecte adressering

De Z80 kent tien verschillende adresseermethoden. Anders gezegd: we kunnen op tien verschillende manieren de Z80 microprocessor duidelijk maken hoe en waar de operand voor een instructie opgezocht moet worden. De tien adresseertechnieken zijn:

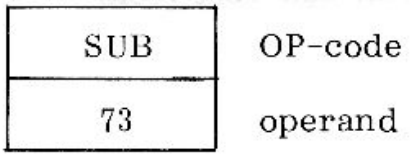
- onmiddellijke adressering
- onmiddellijk uitgebreide adressering
- gemodificeerde pagina nul adressering
- relatieve adressering
- uitgebreide of directe adressering
- register adressering
- impliciete adressering
- register indirecte adressering
- bit adressering
- geïndexeerde adressering

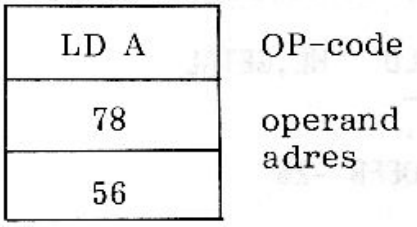
Eerder in dit boek hebben we de onmiddellijke (immediate) en uitgebreide (extended) of directe adressering behandeld. Ook hebben we onopgemerkt een drietal andere adresseermethoden gebruikt en wel de register, de geïmpliceerde en de relatieve adressering.

- Onmiddellijke adressering kenmerkt zich doordat de waarde van de operand als tweede byte van een instructie is opgenomen.
  Voorbeeld: SUB 73 (zie p.53).
- Uitgebreide of directe adressering gebruikt niet de waarde van de operand maar het adres van de operand als tweede en derde byte van een instructie.
  Voorbeeld: LD A,(5678H) (zie p.53).
- Bij Register adressering is de operand een register.
  Voorbeeld: ADD A,C
- Impliciete adressering houdt in dat de operand deel uitmaakt van de OP code (soms is er zelfs geen operand) en niet expliciet in de instructie behoeft te worden gespecificeerd.
  Voorbeeld: HALT (zonder operand) of JP (HL)
- Bij Relatieve adressering is de operand een 'verplaatsing' relatief ten opzichte van de instructie zelf. Deze verplaatsing is opgeno-

men als tweede byte van de instructie.
Voorbeeld: JR 8C. Door gebruik te maken van de twee-complementmethode kunnen zowel positieve als negatieve verplaatsingen opgegeven worden. Effectief bereik is dan van -126 tot +129 (de instructie zelf wordt niet bij de verplaatsing gerekend).

Opgave 7.2
Welke vorm van adressering wordt gebruikt in elk van de volgende instructies: (i) NEG; (ii) INC D; (iii) CP 50; (iv) LD A,(6352H)?





SUB 73 LD A,(5678H)

Onmiddellijke adressering Uitgebreide adressering

- Bit adressering wil zeggen dat we in een CPU-register of in een geheugenplaats één bit kunnen adresseren. Bit adressering komt altijd samen voor met andere adresseervormen.
  Voorbeeld: BIT 3,E. Deze instructie test bit 3 uit register E (zie § 9.1).

- Gemodificeerde pagina-nul adressering komt voor in een speciale CALL instructie, één byte groot, waarmee een sprong naar bepaalde geheugenplaatsen gemaakt kan worden. Die geheugenplaatsen liggen tussen 0000 en 00FF. Al deze adressen beginnen met 00, vandaar pagina-nul! We zullen deze adresseermethode in dit boek niet bespreken.

In § 2.7 is in twee tekeningen de werking van onmiddellijke en directe adressering weergegeven. Zoiets gaan we nu doen met de register-indirecte en de onmiddellijk-uitgebreide adressering.

De onmiddellijk-uitgebreide adressering is, precies wat u al dacht, een uitbreiding van de onmiddellijke adresseervorm, waarbij de waarde van de operand zelf in de instructie gespecificeerd wordt. Zijn dit bij onmiddellijke adressering 8-bit waarden, bij onmiddellijk-uitgebreide adressering worden 16-bit waarden opgegeven.

De 16-bit waarde bij onmiddellijk-uitgebreide adressering wordt in de tweede en derde byte van de instructie gespecificeerd. Zo zal de instructie

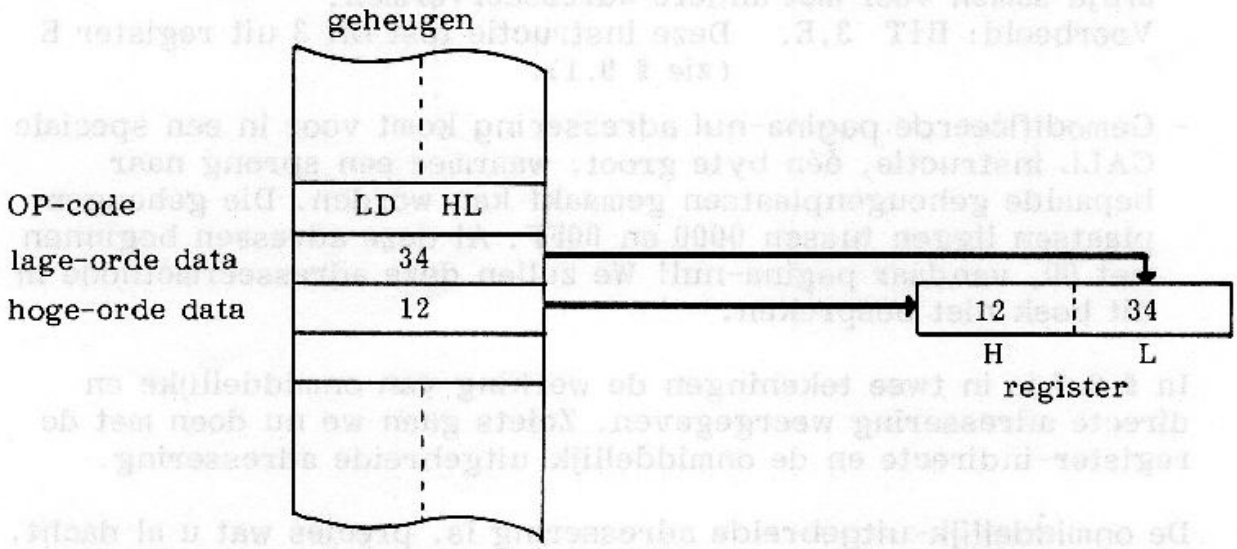
```
LD  HL,1234H
```

de hexadecimale waarde 1234 in het registerpaar HL kopiëren. In plaats van HL kunnen we ook de registerparen BC en DE gebruiken. Het registerpaar HL wordt echter in dit soort opdrachten het meest gebruikt, omdat dit registerpaar gebruikt wordt als verwijzing naar gegevens (data) die in een programma gebruikt worden. Om een gegeven in het programma te kunnen gebruiken moet HL het adres van dit gegeven bevatten. We geven hiervan een voorbeeld:

```
        LD   HL,GETAL
        -
        -
GETAL:  DEFB -25
```

De instructie LD HL,GETAL zorgt ervoor dat het adres van de geheugenplaats (data byte) met label GETAL in het HL register wordt geladen.

Hieronder is schematisch de werking van de onmiddellijk-uitgebreide adressering weergegeven aan de hand van het voorbeeld LD HL,1234H.



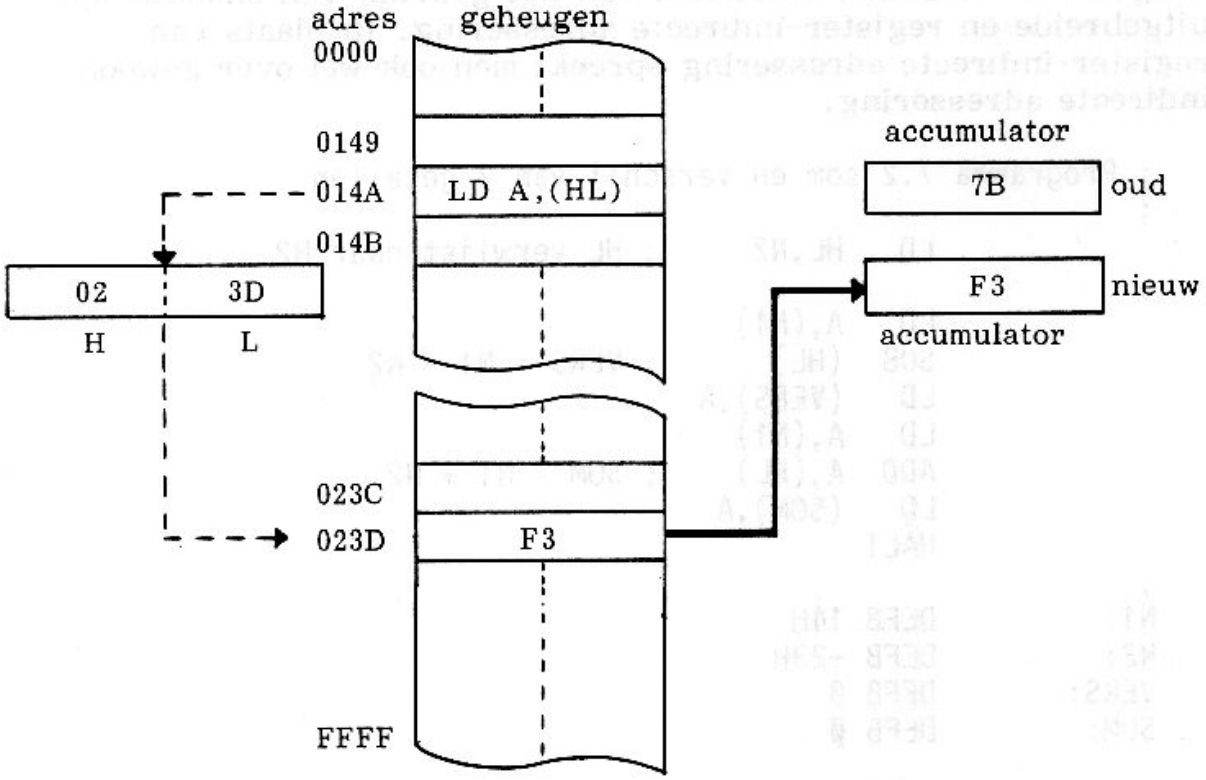
Onmiddellijk-uitgebreide adressering.

We zien dat het lage-orde deel van de data (34) direct na de OP-code staat. Dit lage-orde deel wordt geladen in het L-deel van het HL register. Het hoge-orde deel (12) wordt geladen in het H-deel van het registerpaar HL.

Bij register-indirecte adressering bevindt het adres van de operand zich in een registerpaar. Een voorbeeld hiervan is de instructie

```
LD A,(HL)
```

Door deze instructie wordt de inhoud van de geheugenplaats, waarvan het adres in het registerpaar HL staat, in de accumulator gekopieerd. Hieronder is schematisch de werking van register-indirecte adressering weergegeven aan de hand van bovenstaand voorbeeld.



Register-indirecte adressering.

Uit bovenstaande tekening blijkt dat het uiteindelijke effect van de één-byte instructie LD A,(HL), die als adres 014A heeft, is dat de inhoud van een geheugenplaats (de eigenlijke operand) in de accumulator geladen wordt. De dikgetrokken pijl geeft dus de data-overdracht naar de accumulator aan. Om achter het adres van deze

operand te komen, in dit geval adres 023D, moet de CPU eerst in het HL register kijken. Daar staat namelijk het adres van de operand. De gestippelde pijlen in de tekening geven dus aan hoe de CPU het adres van de operand kan vinden. Omdat deze adressering niet direct maar via het HL register loopt, noemen we dit indirecte adressering.

Uiteindelijk zal de inhoud van adres 023D, de waarde F3H, in de accumulator gekopieerd worden. Hierdoor wordt dus de oude inhoud van de accumulator, de waarde 7BH, overschreven!.

Register-indirecte adressering kan gebruikt worden bij optellen, zoals in ADD A,(HL) of bij aftrekken, zoals SUB(HL). Ook kan met deze adresseringsvorm de inhoud van de accumulator gekopieerd worden in een geheugenplaats, waarvan zich het adres in het registerpaar HL bevindt, zoals in LD (HL),A.

Programma 7.2 is een voorbeeld van het gebruik van onmiddellijk-uitgebreide en register-indirecte adressering. In plaats van register-indirecte adressering spreekt men ook wel over gewoon indirecte adressering.

```
; Programma 7.2 som en verschil van 2 getallen
;
            LD   HL,N2      ; HL verwijst naar N2
;
            LD   A,(N1)
            SUB  (HL)       ; VERS = N1 - N2
            LD   (VERS),A
            LD   A,(N1)
            ADD  A,(HL)     ; SOM = N1 + N2
            LD   (SOM),A
            HALT
;
N1:         DEFB 14H
N2:         DEFB -23H
VERS:       DEFB Ø
SOM:        DEFB Ø
```

Dit programma berekent de som en het verschil van twee getallen N1 en N2, die aan het einde van het programma in een zogeheten 'datagebied' zijn opgenomen.

De eerste instructie (LD HL,N2) zorgt ervoor dat het HL register verwijst naar de 'data-byte' N2. Na het laden van de accumulator (LD A,(N1)) met de inhoud van het 'data-byte' N1 wordt de inhoud van de geheugenplaats waarnaar in het HL register verwezen wordt (N2 dus) afgetrokken van de accumulator (SUB (HL)). Op dezelfde

wijze wordt door de instructie ADD A,(HL) de inhoud van die geheugenplaats bij de accumulator opgeteld.

Opgave 7.3
Stel het adres van de geheugenplaats met label N1 is 1760H. Wat is na het uitvoeren van programma 7.2 de inhoud van de accumulator en van het HL register?

Geïndexeerde adressering komt in § 9.4 aan de orde.

7.3 De DEFM pseudo-instructie

In het 'datagebied' van een programma kunnen we een tekenstring (character string) specificeren met een aantal DEFB pseudo-instructies. Een voorbeeld:

```
TEKST:  DEFB  'B'
        DEFB  'E'
        DEFB  'R'
        DEFB  'I'
        DEFB  'C'
        DEFB  'H'
        DEFB  'T'
```

Dit is echter een nogal omslachtige manier. Het kan korter!

```
TEKST:  DEFM  'BERICHT'
```

De betekenis van bovenstaande DEFM instructie is exact dezelfde als die van de zeven DEFB instructies.

De instructie DEFM is echter veel korter en duidelijker. Als de assembler (het assembleertaal-vertaalprogramma) zo'n DEFM pseudo-instructie in het source programma tegenkomt, wordt het eerste teken van de string, in dit geval de B, opgeslagen op het adres met het label TEKST. Elk volgend teken uit de string komt in een daaropvolgende geheugenplaats terecht. In bovenstaand voorbeeld neemt de string dus zeven opvolgende geheugenplaatsen (bytes) in beslag.

DEFM is een mnemonic voor DEFine Message.

Opgave 7.4
Schrijf een tekenstring met label REGEL die afgedrukt op het scherm voor de volgende tekst zorgt:

```
EERSTE REGEL
TWEEDE REGEL
```

7.4 Uitvoer van tekst

Als we weinig tekens (een, twee of drie) op het beeldscherm willen afdrukken, doen we dat doorgaans door de accumulator verschillende keren te laden met de code van het gewenste uitvoerteken en de COUT routine aan te roepen. Moeten we echter meer tekens afdrukken, dan is deze wijze van uitvoer niet al te efficient.

In zo'n situatie is het efficienter om van een programma gebruik te maken. In programma 7.3 zien we hiervan een voorbeeld. De tekst (5 tekens) wordt in een DEFM instructie aan het einde van het programma gespecificeerd.

```
; Programma 7.3 tekstuitvoer op het beeldscherm
;
        LD    HL,TEKST      ; HL wijst naar het eerste teken
        LD    B,5
;
WEER:   LD    A,(HL)        ; haal volgend teken
        CALL  COUT
        INC   HL            ; HL wijst naar het volgende teken
        DJNZ  WEER
        HALT
;
TEKST:  DEFM  'ABCDE'
```

Eerst wordt het HL register gevuld met het adres van TEKST. Dit betekent dat de inhoud van het HL register verwijst naar de geheugenplaats waar zich het eerste teken van de tekst bevindt. Vervolgens wordt dit teken in de accumulator geladen (LD A,(HL)) en naar het beeldscherm overgebracht (CALL COUT). Daarna wordt het HL register met één verhoogd (INC HL) en daardoor verwijst de inhoud van dit register nu naar het tweede teken uit de tekststring. Register B wordt gebruikt als lusteller en begint op 5.

Opgave 7.5
Herschrijf programma 7.3 zó dat het programma, in plaats van het (af)tellen van de uitgevoerde tekens, een geheugenplaats met 'nul' als inhoud herkent als het 'einde' van de tekststring.

De INC HL instructie is wellicht nieuw voor u. Alle registerparen BC, DE, HL, SP, IX en IY (aangeduid met rp) kunnen met één verhoogd of verlaagd worden met behulp van respectievelijk

`INC rp` èn `DEC rp`

Het verschil tussen het ophogen of verlagen van registerparen èn enkelvoudige registers is dat deze instructies bij registerparen géén invloed op het vlagregister (F) hebben, terwijl dat bij enkelvoudige registers wel het geval is. U kunt dit zelf in tabel C.7 van Appendix C nagaan.

7.5 Het subroutine mechanisme

Reeds eerder hebben we gebruik gemaakt van subroutines. Nu laten we zien hoe het mechanisme werkt, dat wil zeggen wat er gebeurt bij het aanroepen van (CALL) en bij het terugkeren uit (RET) subroutines.

Alvorens hiertoe over te gaan zullen we eerst iets algemeens zeggen over het uitvoeren van instructies. Van elk programma dat door de CPU moet worden uitgevoerd bevinden zich de instructies in het geheugen van de microprocessor. Daar wachten zij geduldig op het moment dat zij door de CPU worden opgehaald en uitgevoerd.

Als een bepaald programma moet worden uitgevoerd, weet de CPU waar zich de eerste instructie uit dit programma in het geheugen bevindt; de CPU weet dus waar dat programma begint. De CPU weet dit, omdat vlak voor het uitvoeren van het programma het adres van de eerste instructie uit het programma in het PC register geladen wordt.

Het PC register is de Program Counter en de inhoud van dit register is altijd het adres van de eerstvolgende uit te voeren instructie.

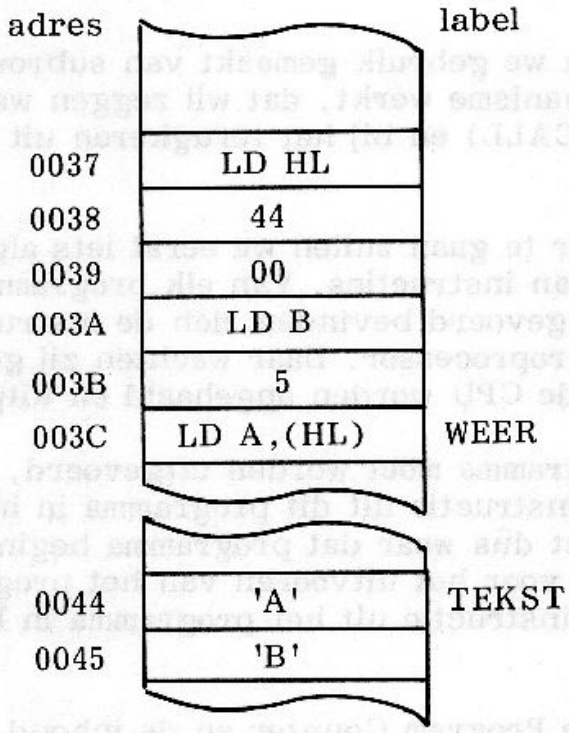
De CPU leest de eerste instructie in en voert deze uit. Doordat de CPU weet 'hoe lang' (het aantal bytes) elke instructie is, weet de CPU ook met welk bedrag de Program Counter (het PC register) moet worden opgehoogd om het adres van de eerstvolgende instructie te bepalen. Zo weet de CPU straks waar zich de volgende uit te voeren instructie bevindt. Zo worden de instructies één voor één ingelezen en uitgevoerd.

De hierboven geschetste werkwijze voldoet zolang de uit te voeren instructies géén sprong in het programma inhouden.

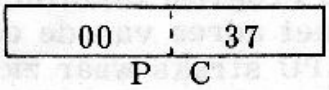
We zullen nu aan de hand van de eerste twee instructies uit programma 7.3 (zie hierboven) laten zien hoe een en ander werkt.

We nemen even aan dat de eerste instructie uit programma 7.3, de instructie LD HL,TEKST, zich op adres 0037 bevindt. Nu zal de

assembler ervoor gezorgd hebben dat het label TEKST vervangen wordt door het adres van de instructie met dat label. Dit zal trouwens ook gebeuren bij het label COUT in CALL COUT en bij het label WEER in DJNZ WEER. Gezien het beginadres 0037 en de 'lengte' van alle instructies zal het adres van de geheugenplaats met label TEKST gelijk zijn aan 0044 (denk erom: adresaanduiding is hexadecimaal!).
De eerste instructie LD HL,TEKST luidt dus eigenlijk LD HL,0044H. Een kijkje in het geheugen levert dus het volgende beeld op:

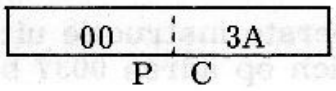


Vlak voor het uitvoeren van programma 7.3 bevat het PC register dus het adres 0037, dus:



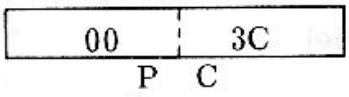
De CPU haalt de instructie die op dit adres begint op en voert de instructie uit. Hierbij wordt tevens de Program Counter, het PC register, verhoogd met het aantal bytes dat de instructie 'lang' is, in dit geval met 3 omdat de LD HL,0044H instructie 3 bytes in beslag neemt, dus:

LD HL,0044H ⇒ PC = PC + 3 ⇒



Nu wijst het PC register naar adres 003A, het adres waar de volgende instructie (LD B,5) begint. Deze instructie wordt gelezen (is 2 bytes lang) en uitgevoerd èn

LD B,5 ⇒ PC = PC + 2



Het PC register wordt met 2 opgehoogd en wijst nu naar de volgende instructie, de instructie LD A,(HL). Dit gaat zo door totdat de CPU de instructie CALL COUT tegenkomt.

Deze subroutine-aanroep zal een sprong in het programma betekenen. Het eenvoudig ophogen van PC met de lengte van de instructie CALL COUT zal niet voldoende zijn. Het PC register zal daarentegen geladen moeten worden met het adres van de eerste instructie uit de subroutine COUT.

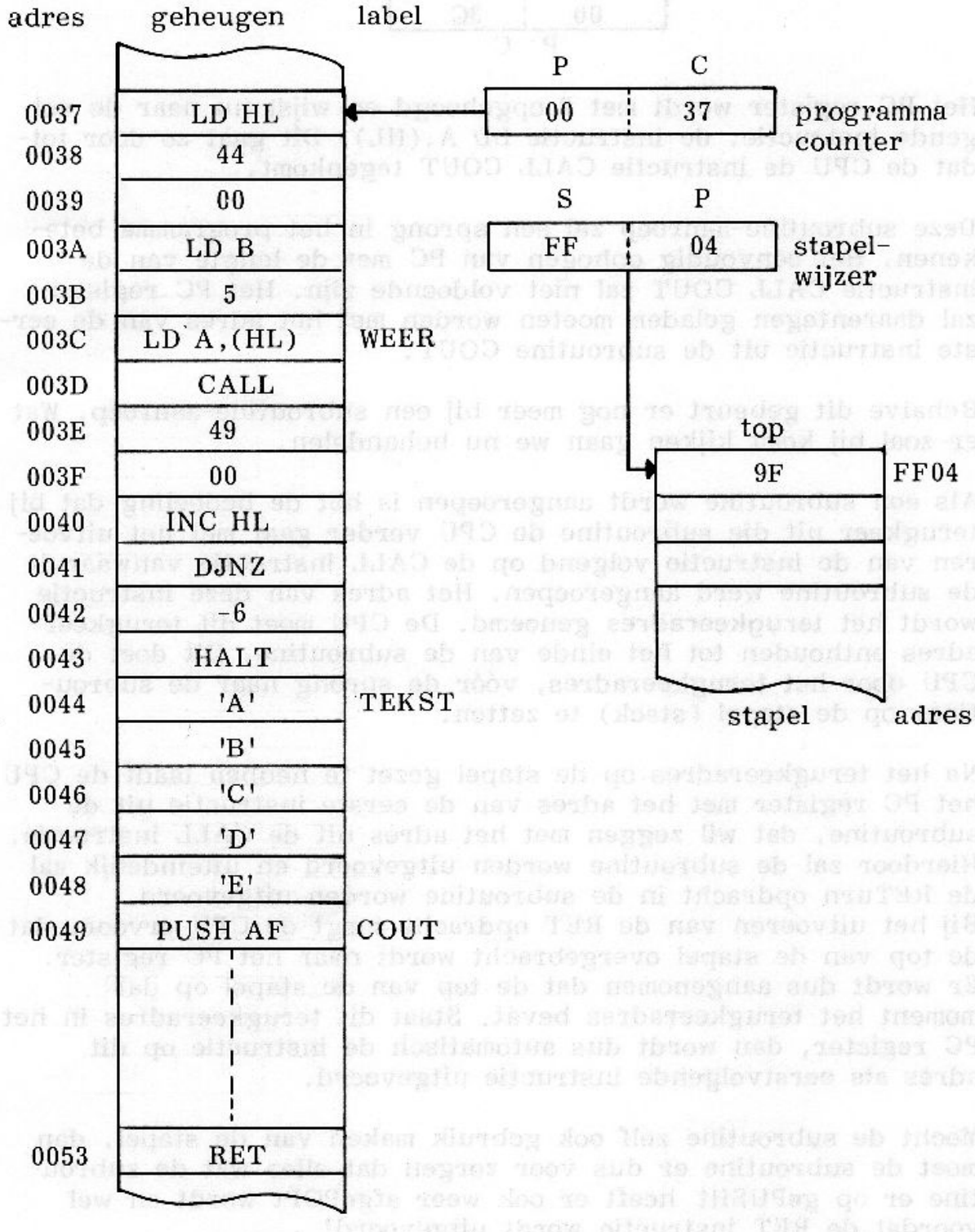
Behalve dit gebeurt er nog meer bij een subroutine-aanroep. Wat er zoal bij komt kijken gaan we nu behandelen.

Als een subroutine wordt aangeroepen is het de bedoeling dat bij terugkeer uit die subroutine de CPU verder gaat met het uitvoeren van de instructie volgend op de CALL instructie vanwaaruit de subroutine werd aangeroepen. Het adres van deze instructie wordt het **terugkeeradres** genoemd. De CPU moet dit terugkeeradres onthouden tot het einde van de subroutine. Dit doet de CPU door het terugkeeradres, vóór de sprong naar de subroutine, op de **stapel** (stack) te zetten.

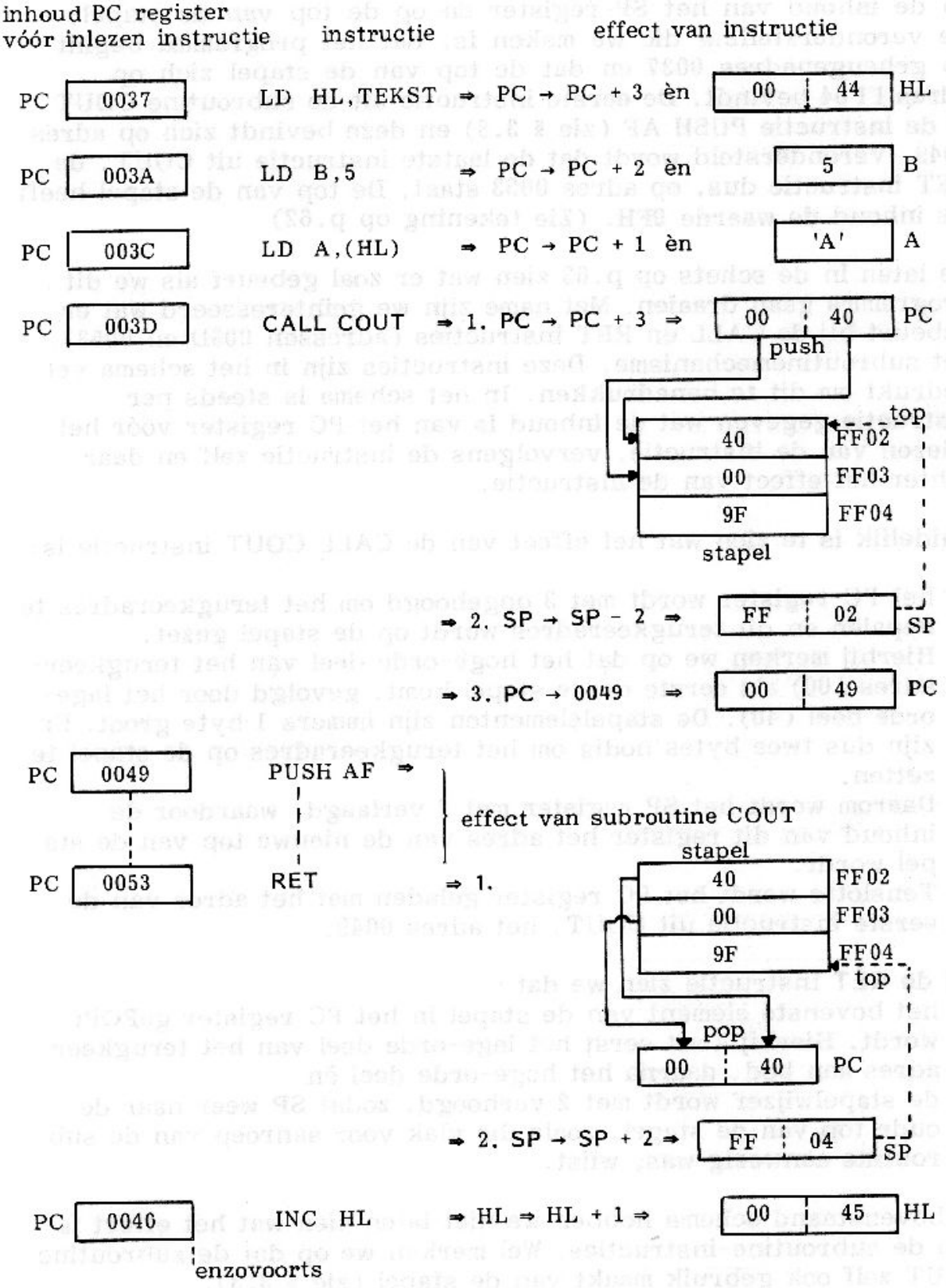
Na het terugkeeradres op de stapel gezet te hebben laadt de CPU het PC register met het adres van de eerste instructie uit de subroutine, dat wil zeggen met het adres uit de CALL instructie. Hierdoor zal de subroutine worden uitgevoerd en uiteindelijk zal de RETurn opdracht in de subroutine worden uitgevoerd.
Bij het uitvoeren van de RET opdracht zorgt de CPU ervoor, dat de top van de stapel overgebracht wordt naar het PC register. Er wordt dus aangenomen dat de top van de stapel op dat moment het terugkeeradres bevat. Staat dit terugkeeradres in het PC register, dan wordt dus automatisch de instructie op dit adres als eerstvolgende instructie uitgevoerd.

Mocht de subroutine zelf ook gebruik maken van de stapel, dan moet de subroutine er dus voor zorgen dat alles wat de subroutine er op gePUSHt heeft er ook weer afgePOPt wordt en wel voordat de RET instructie wordt uitgevoerd!

We zullen dit subroutinemechanisme nog eens schematisch weergeven. Als voorbeeld nemen we programma 7.3 op p.58. Eerst laten we het stukje geheugen zien met het programma, inclusief de subroutine COUT. Daarna laten we zien wat het effect is van het uitvoeren van een aantal instructies op het PC register,



Programma 7.3 in het geheugen.



Voor uitleg zie p.64.

op de inhoud van het SP-register en op de top van de stapel. De veronderstelling die we maken is, dat het programma begint op geheugenadres 0037 en dat de top van de stapel zich op adres FF04 bevindt. De eerste instructie uit de subroutine COUT is de instructie PUSH AF (zie § 3.3) en deze bevindt zich op adres 0049. Verondersteld wordt dat de laatste instructie uit COUT, de RET instructie dus, op adres 0053 staat. De top van de stapel heeft als inhoud de waarde 9FH. (Zie tekening op p.62)

We laten in de schets op p.63 zien wat er zoal gebeurt als we dit programma gaan draaien. Met name zijn we geïnteresseerd wat er gebeurt bij de CALL en RET instructies (adressen 003D en 0053), het subroutinemechanisme. Deze instructies zijn in het schema vetgedrukt om dit te benadrukken. In het schema is steeds per instructie gegeven wat de inhoud is van het PC register vóór het inlezen van de instructie, vervolgens de instructie zelf en daarachter het effect van de instructie.

Duidelijk is te zien wat het effect van de CALL COUT instructie is:

1. het PC register wordt met 3 opgehoogd om het terugkeeradres te bepalen en dit terugkeeradres wordt op de stapel gezet. Hierbij merken we op dat het hoge-orde deel van het terugkeeradres (00) als eerste op de stapel komt, gevolgd door het lage-orde deel (40). De stapelelementen zijn immers 1 byte groot. Er zijn dus twee bytes nodig om het terugkeeradres op de stapel te zetten.
2. Daarom wordt het SP register met 2 verlaagd, waardoor de inhoud van dit register het adres van de nieuwe top van de stapel wordt.
3. Tenslotte wordt het PC register geladen met het adres van de eerste instructie uit COUT, het adres 0049.

Bij de RET instructie zien we dat
1. het bovenste element van de stapel in het PC register gePOPt wordt. Hierbij komt eerst het lage-orde deel van het terugkeeradres aan bod, daarna het hoge-orde deel èn
2. de stapelwijzer wordt met 2 verhoogd, zodat SP weer naar de oude top van de stapel, zoals die vlak voor aanroep van de subroutine aanwezig was, wijst.

In bovenstaand schema hebben we niet laten zien wat het effect is van de subroutine-instructies. Wel merken we op dat de subroutine COUT zelf ook gebruik maakt van de stapel (zie § 3.3).

7.6 Een programmeeropdracht

Schrijf een subroutine REGELS waarmee een bepaald teken op m regels n keer kan worden afgedrukt. Bij aanroep van de subroutine bevat de accumulator de code van het gewenste teken, terwijl de waarden voor m en n worden bewaard in respectievelijk het B en C register.

Schrijf daarnaast nog een subroutine TEKST voor het afdrukken van een string met tekens. Bij aanroep bevat het HL register het adres van het eerste teken uit de string. Het einde van de string wordt aangegeven met een geheugenplaats met inhoud nul.

Gebruik nu deze twee en eerder gemaakte subroutines om een programma te maken dat het volgende op het beeldscherm afdrukt:

```
******************************
******************************
******************************
------------------------------
------------------------------
------------------------------

   uw naam
   uw adres - eerste regel
            - tweede regel
            - derde regel

------------------------------
******************************
******************************
******************************
```

Een extra uitdaging zou kunnen zijn om het laatste groepje sterretjes als volgt te laten afdrukken:

```
 ****************************
  **************************
   ************************
    **********************
     ********************
      ******************
       ****************
        **************
         ************
          **********
           ********
            ******
             ****
              **
```

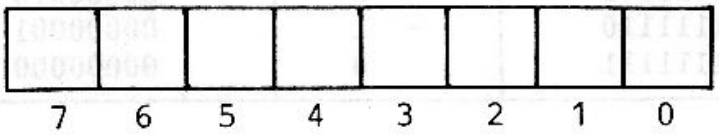
Het is 'netter' hiervoor van geneste lussen gebruik te maken dan het domweg programmeren van het veertien maal aanroepen van de TEKST subroutine steeds met een andere string!

8 Carry en Overflow

Carry en overflow zijn situaties die kunnen optreden bij optellen en aftrekken. Aan de Carry en Overflow bits uit het vlagregister kunnen we zien of na een optelling of aftrekking 'Carry' of 'Overflow' of beide zijn opgetreden.

8.1 Carry

Een byte bestaat uit 8 bits, genummerd 0 t/m 7.



Het achtste bit wordt wel het meest significante genoemd.
Stel we hebben een geheugenplaats met de volgende inhoud:

11111111

Als deze inhoud een 'getal zonder teken' voorstelt, dan is de waarde van dit getal 255. Dit is de 'grootste' waarde (zonder teken) die we in één byte kwijt kunnen. Zouden we hier één bij optellen, dan kan het resultaat nooit in 8 bits worden weergegeven.

```
  11111111      255
  00000001    +   1   +
[1]00000000      256
```

We hebben een negende bit (bit 8) nodig om het resultaat van de optelling (256) weer te geven. Dit 9e bit noemen we het **carrybit** of **overdrachtsbit**. Registers zijn in het algemeen slechts 8 bits 'breed'. We moeten kunnen nagaan of er 'carry' is opgetreden om al naar gelang de betekenis (er kan ook nog in andere situaties carry optreden) van de carry maatregelen te kunnen nemen.

We nemen nu aan dat we voor het opslaan van getallen de **twee-complementmethode** (zie Appendix A.8) gebruiken. Bij deze methode is het 8e bit het zogenaamde tekenbit. We verliezen dus één bit, hetgeen betekent dat we in één byte getallen kunnen opslaan van -128 tot +127. Hieronder is een overzicht gegeven van deze twee-complementmethode.

negatief		positief	
decimaal	twee-complement	decimaal	twee-complement
-128	10000000	+127	01111111
-127	10000001	+126	01111110
-126	10000010	+125	01111101
-125	10000011	+124	01111100
⋮		⋮	
- 10	11110110	+ 9	00001001
- 9	11110111	+ 8	00001000
- 8	11110000	+ 7	00000111
- 7	11111001	+ 6	00000110
- 6	11111010	+ 5	00000101
- 5	11111011	+ 4	00000100
- 4	11111100	+ 3	00000011
- 3	11111101	+ 2	00000010
- 2	11111110	+ 1	00000001
- 1	11111111	0	00000000

In het volgende voorbeeld treedt géén carry op:

$$\begin{array}{rr} 00110011 & (+51) \\ \underline{00011100} + & \underline{(+28)} + \\ 01001111 & (+79) \end{array}$$

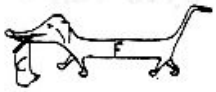
Maar in (twee-complementcode) dit voorbeeld:

$$\begin{array}{rrr} & 11111110 & (-2) \\ & \underline{11111111} & \underline{(-1)} + \\ [1] & 11111101 & (-3) \end{array}$$

treedt wel carry op. In dit geval kan het carrybit gewoon genegeerd worden, want de 8 bits 11111101 geven op zich al het juiste resultaat, namelijk -3 (zie tabel).

Opgave 8.1
Geeft 11000000 + 01000000 een carry?

8.2 De Carry vlag



Als er na een optelling carry is opgetreden wordt het carrybit (de carry vlag) in het F register op 1 gezet; zo niet, dan wordt deze vlag 0. We kunnen de waarde van deze carry vlag testen met behulp van één van de volgende voorwaardelijke sprongopdrachten:

```
JP  C,label
JP  NC,label
JR  C,label
JR  NC,label
```

C betekent Carry en NC betekent No Carry. JP C,label betekent spring naar label als C=1, dus als er carry is opgetreden.

De carry vlag wordt ook gebruikt in een SUB instructie, als bij aftrekking van de meest significante bits 'één geleend' moet worden. We spreken dan van een borrow.

Opgave 8.2
Zal na de JP instructie in het hiernavolgende programma de instructie met label NCARRY of die met label CARRY worden uitgevoerd?

```
        LD  A,7
        SUB 8
        JP NC,NCARRY
CARRY: -
```

We kunnen twee instructies gebruiken om de carry vlag te veranderen. Dit zijn de instructies

`SCF` èn `CCF`

respectievelijk Set Carry Flag en Complement Carry Flag.
SCF maakt het carrybit in het F register 1, terwijl CCF het carrybit nul maakt als het één is en één maakt als het nul is. Beide instructies zijn te vinden in tabel C.6 in Appendix C.

Opgave 8.3
Schrijf een reeks opdrachten om de carry vlag op nul te zetten.

8.3 Overflow

Het grootste getal in twee-complementcode dat we in één byte van 8 bits kunnen weergeven is het getal +127, ofwel

```
01111111    (+127)
```

Laten we hier eens 1 bij optellen:

```
bit 7 ──→
      01111111    (+127)
      00000001  + (+   1) +
      --------    -------
      10000000    (-128)
bit 7 ←──
```

We zien nu dat de computer het resultaat zal interpreteren als -128 en niet als +128. Dit komt omdat bit 7 één geworden is. Deze situatie noemen we **overflow**. Deze overflow ontstaat omdat het resultaat +128 te 'groot' is om in 8 bits (twee-complement) te worden weergegeven.

Bij een 8-bit optelling zullen twee getallen met verschillend teken nooit overflow opleveren. Dit kan pas gebeuren bij het optellen van twee positieve of twee negatieve getallen. Zo zal de optelling

```
01100100    (+100)
00110001  + (+ 49) +
--------    -------
10010101    (-107)
```

(+100) + (+49) niet het juiste resultaat +149 opleveren, maar in dit geval -107. Dit komt omdat +149 groter is dan +127. Hier treedt dus overflow op (bit 7 gaat van 0 naar 1).

Bij aftrekken kan alleen overflow optreden indien de getallen een verschillend teken hebben. Zo zal de aftrekking

```
bit 7 ──→
      01111110    (+126)
      11000000  - (- 64) -
      --------    -------
   [1]10111110    (- 66)
bit 8←┘ ↑bit 7
```

niet het juiste resultaat hebben. Dit komt omdat het goede antwoord +190 groter is dan +127. Bit 7 had dus 0 moeten zijn en niet één, overflow dus. We zien in dit voorbeeld dat er bovendien carry is opgetreden.

Opgave 8.4
Bij optellen kunnen ook carry en overflow samen optreden. Geef een voorbeeld van het optellen van twee 8-bit getallen waarbij zich dit voordoet.

8.4 Overflow vlag

De overflow vlag is tevens de pariteitsvlag (zie § 12.5) omdat de Z80 deze vlag ook voor andere doeleinden dan alleen het aangeven van overlow gebruikt, vandaar de letter P! Wordt de vlag, bij rekenkundige bewerkingen, gebruikt om al dan niet overflow te signaleren, dan noemen we deze vlag de 'overflow vlag'.

Als er overflow optreedt, wordt de overflow vlag één gemaakt, zo niet dan nul. Dit gebeurt bij ADD, SUB, INC, DEC, NEG en CP instructies. We kunnen de vlag testen met

```
JP  PO,label
JP  PE,label
```

PO betekent eigenlijk Parity Odd, oneven pariteit, en PE staat voor Parity Even, even pariteit. PE komt overeen met overflow en PO met géén overflow. PO en PE hebben betrekking op het gebruik van de vlag als pariteitsvlag en niet op het gebruik als overflow vlag, vandaar de wellicht kleine verwarring! Er zijn voor deze twee voorwaardelijke sprongopdrachten géén JR equivalenten.

8.5 Voorwaardelijke CALL en RET instructies

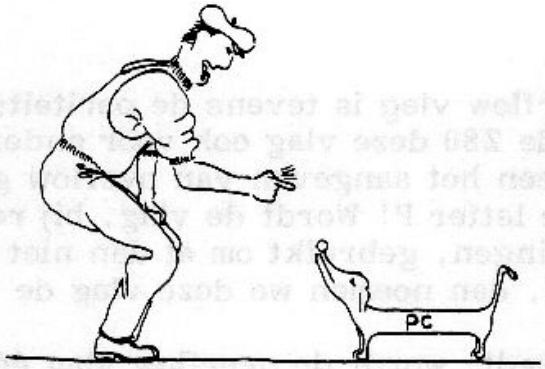
Naast de in het vorige hoofdstuk behandelde onvoorwaardelijke CALL en RET instructies (CALL COUT en RET) kunnen we ook een subroutine aanroepen of uit een subroutine terugkeren alleen als aan een bepaalde voorwaarde is voldaan. We spreken dan van een voorwaardelijke aanroep of voorwaardelijke terugkeer. Zo zouden we het volgende programmafragment

```
        JP    NZ,OVER
        CALL  SUBEX
OVER:
```

ook kunnen coderen met één instructie:

```
CALL Z,SUBEX
```

en daarmee besparen we dus één instructie.



"denk erom, je gaat alleen als ....."

De mogelijke voorwaarden in de voorwaardelijke CALL instructie zijn dezelfde als die bij de voorwaardelijke sprongopdrachten, dat wil zeggen dat als de voorwaarde mogelijk is Z,NZ; M,P; C,NC en PO en PE.

Op dezelfde wijze kunnen voorwaardelijke RET opdrachten gecodeerd worden. Het is echter ongebruikelijk dit soort voorwaardelijke terugkeerinstructies te gebruiken, een subroutine hoort immers maar één onvoorwaardelijke RET instructie te bevatten! Daarom zal deze situatie:

```
        -
        JP   NC,EXIT
        -
        JP   EXIT
        -
        JP   Z,EXIT
        -
EXIT:   RET
```

verkozen worden boven deze:

```
        -
        RET  NC
        -
        RET
        -
        RET  Z
        -
        RET
```

Het is netter, verstandiger, maar bovenal 'gewoon beter' om in elke subroutine slechts één RETurn instructie op te nemen en wel als laatste instructie van een subroutine. Niet alleen voor de overzichtelijkheid en leesbaarheid van een programma, maar vooral voor het voorkomen van fouten bij het eventueel wijzigen of aanvullen van een programma.

Doorgaans wordt het 'exit' label gebruikt als label van de eerste instructie uit een groep instructies, die een subroutine afsluit. Stel dat aan het einde van een subroutine nog enkele registers 'hersteld' moeten worden, dan zou zo'n laatste exit-groep er zó uit kunnen zien:

```
        -
EXIT:   POP BE
        POP DE
        RET
```

8.6 Een programmeeropdracht

Wijzig de subroutine GETIN (zie § 6.6) zo dat getallen met teken, in plaats van getallen zonder teken, ingevoerd kunnen worden. De invoer zal dan bestaan uit getallen als -121, +84 en 53. Als we geen teken opgeven impliceert dit een positief getal!

Schrijf vervolgens een programma dat vraagt om het invoeren van twee getallen. De dialoog is als volgt:

```
VOER EERSTE GETAL IN n1
VOER TWEEDE GETAL IN n2
```

De uitvoer van het programma moet als volgt zijn:

```
n1 + n2 IS nnop
```

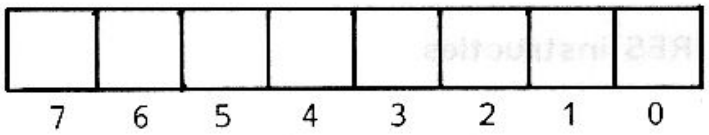
nnop staat voor een van de woorden NUL, NEGATIEF of POSITIEF. In voorkomende gevallen moet een volgende regel worden afgedrukt

met OVERFLOW OPGETREDEN, eventueel gevolgd door nog een regel met CARRY OPGETREDEN.
De twee invoergetallen n1 en n2 zijn tweecijferige decimale getallen met teken zoals -08, 63, +50.

Het programma moet meer dan één paar invoergetallen aankunnen.
Het programma moet stoppen als we voor n1 het woord END invoeren.

9 Bit Instructies en de Indexregisters

De Z80 bezit een groot aantal instructies waarmee we iets met individuele bits uit een register of geheugenplaats kunnen doen. We kunnen testen of een bit 0 of 1 is en we kunnen een bit op 0 of 1 zetten. We moeten in zo'n instructie het nummer van het desbetreffende bit opgeven. De bits zijn genummerd van rechts naar links met de nummers 0 t/m 7.



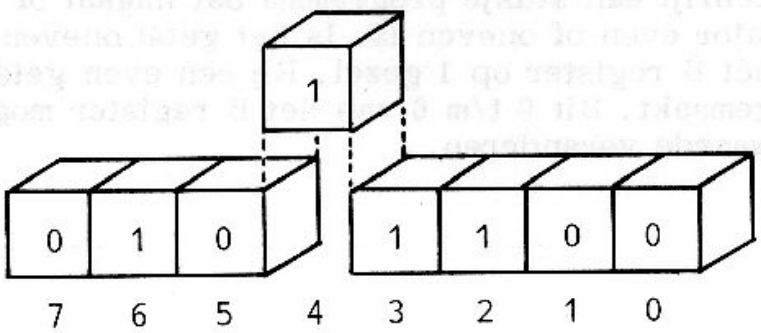
Bit instructies kunnen op bits uit elk enkelvoudig register worden uitgevoerd en tevens op een geheugenplaats waarvan het adres zich in het HL, IX of IY register bevindt.

9.1 De BIT-TEST instructie

Met de BIT instructie testen we of een bepaald bit nul of een is. Is het bit nul, dan wordt de zero vlag op 1 gezet; is het bit 1, dan wordt de zero vlag 0. Zo zal de instructie

```
BIT 4,E
```

de zero vlag op 1 zetten als bit 4 uit register E 0 is en de vlag op 0 zetten als bit 4 één is.
Dit laatste doet zich voor als register E bijvoorbeeld 01011100B bevat.



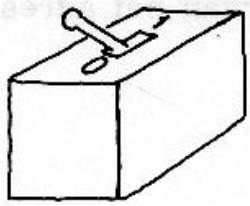
In programma 3.2 (zie § 3.3) zit een BIT-instructie waarmee bit 0 van de accumulator getest wordt.

Opgave 9.1
Wat zal de waarde van de Z-vlag zijn na het uitvoeren van de instructie BIT 1,A, indien de accumulator FDH bevat?

We kunnen de werking van de BIT instructie gemakkelijk onthouden door te bedenken dat de Z-vlag het complement wordt van het gespecificeerde bit (0→1 èn 1→0).

Doorgaans wordt een BIT instructie gevolgd door een 'jump-on-zero' of door een 'jump-on-non-zero' instructie, respectievelijk met de voorwaarden Z en NZ, corresponderend met de waarde 0 of 1 van het gespecificeerde bit. Dit betekent dat het eigenlijk niet nodig is precies te onthouden wat de waarde is van de Z-vlag!

9.2 De SET en RES instructies



Met de SET instructie zetten we een bepaald bit op 1. Zo zal de instructie

```
SET 2,C
```

bit 2 uit register C op 1 zetten. Alle andere bits van C blijven onveranderd.

De instructie

```
RES 5,(HL)
```

zal bit 5 van de geheugenplaats, waarvan het adres zich in HL bevindt, op 0 zetten, zonder daarbij de andere bits uit deze geheugenplaats te beïnvloeden.

Opgave 9.2
Schrijf een stukje programma dat nagaat of een getal in de accumulator even of oneven is. Is het getal oneven, dan wordt bit 7 van het B register op 1 gezet. Bij een even getal wordt dit bit 0 gemaakt. Bit 0 t/m 6 van het B register mogen hierbij niet van waarde veranderen.

9.3 De DEFS pseudo-instructie

Met de DEFS pseudo-instructie kunnen we een aantal geheugenplaatsen (bytes) **reserveren**. Dit gebeurt meestal in het datagebied achterin een programma. De DEFS instructie reserveert slechts geheugenruimte, terwijl we met de DEFM pseudo-instructie bepaalde waarden aan geheugenplaatsen kunnen toekennen.

In het volgende voorbeeld

```
GETAL:   DEFS 1
BUFFER:  DEFS 96
NAAM:    DEFM 'FRED'
```

reserveert de eerste DEFS één geheugenplaats met label GETAL, terwijl de tweede DEFS 96 aaneensluitende geheugenplaatsen reserveert, waarbij de eerste de label BUFFER krijgt. De DEFM instructie reserveert niet alleen geheugenruimte maar definieert een stukje data dat bestaat uit de tekst FRED. De label NAAM wordt toegewezen aan de geheugenplaats waar FRED begint.

Opgave 9.3
Stel dat in bovenstaand voorbeeld het adres van de label GETAL 0100H is. Wat is dan het adres van de label NAAM? (hexadecimaal!)

In de loop van dit hoofdstuk komen we nog een aantal voorbeelden van het gebruik van de DEFS pseudo-instructie tegen.

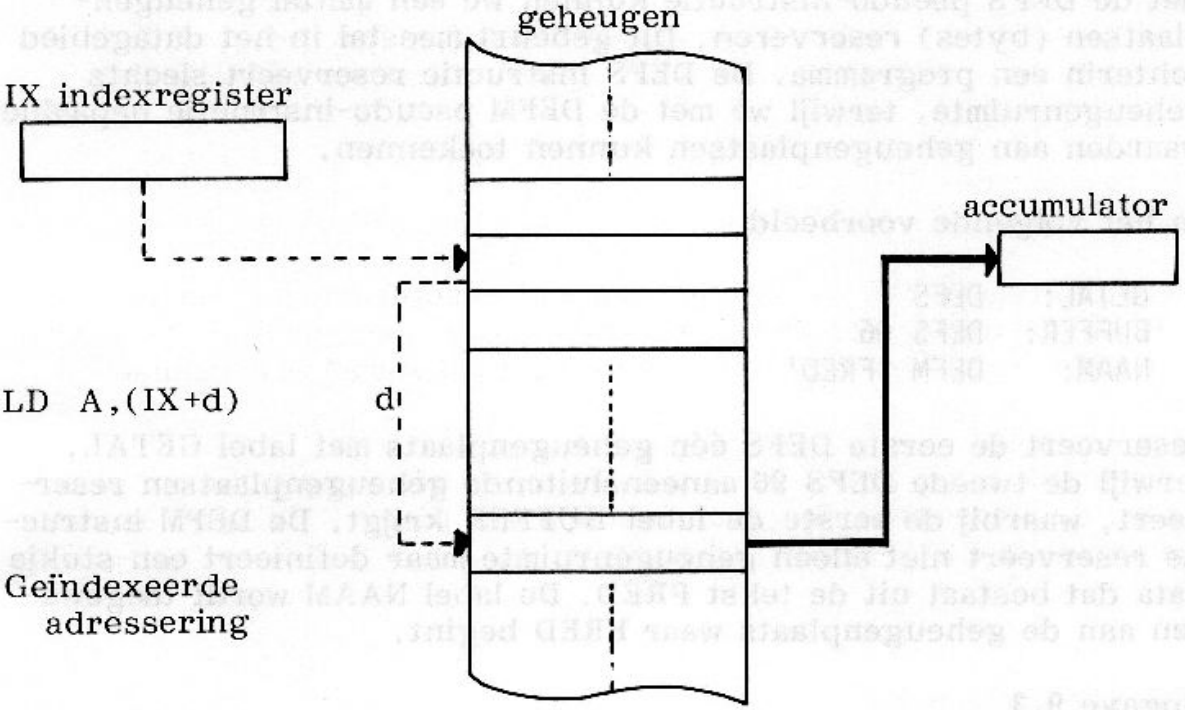
9.4 Indexregisters

De Z80 kent twee indexregisters, IX en IY. In zo'n indexregister kunnen we het adres van een bepaalde geheugenplaats specificeren. Dit adres is dan een beginadres van een '**geheugenblok**'. Als we nu een operand specificeren (dat wil zeggen het adres van de operand) relatief ten opzichte van het begin van dit blok, dan spreken we van **geïndexeerde adressering**. De volgende opdracht maakt van geïndexeerde adressering gebruik:

```
LD  A,(IX + d)
```

d is de **verplaatsing** (displacement) ten opzichte van het begin van het geheugenblok, waarvan het beginadres bepaald wordt door de

inhoud van het IX register. In onderstaande tekening is aangegeven hoe zo'n geïndexeerde adressering in een bepaalde situatie werkt.



In het hiernavolgende programma zien we een voorbeeld van het gebruik van geïndexeerde adressering.

```
        LD    IX,START ; IX wijst naar het begin van een blok
        -
        LD    A,(IX+2) ; A wordt gelijk aan het 3e byte van het blok
        -
        SET   3,(IX+5) ; bit 3 van het 6e byte uit het blok wordt 1
        -
        DEC   (IX+9)   ; verlaagt het laatste byte uit het blok met 1
        -
START:  DEFS  10
        -
```

We zien dat achteraan in het programma een blok van 10 geheugenplaatsen gereserveerd wordt met behulp van een DEFS pseudo-instructie. De eerste gereserveerde geheugenplaats krijgt de naam START. De LD IX,START instructie zorgt ervoor dat het indexregister IX verwijst naar deze eerste geheugenplaats.
Vervolgens wordt een bepaalde geheugenplaats in het blok gespecificeerd door een relatieve verplaatsing ten opzichte van het begin van het blok op te geven. Zo wordt bijvoorbeeld de vierde geheu-

genplaats uit het blok gespecificeerd met IX+3.

Opgave 9.4
Stel dat in bovenstaand programmavoorbeeld het blok gespecificeerd wordt met START: DEFM 'ABCDEFGHIJ'. Wat zal dan de inhoud van de accumulator zijn na uitvoering van de instructie LD A,(IX+7)?

Op precies dezelfde manier kunnen we van het IY indexregister gebruik maken.

De indexregisters zijn handig voor het raadplegen van en werken met bijvoorbeeld records uit een bestand of gegevens uit tabellen, dat wil zeggen aparte stukjes data die toch samen één geheel vormen.

Gewoonlijk is bij het gebruik van IX en IY de verplaatsing nul (zero displacement), hetgeen inhoudt dat IX en IY dan op eenzelfde wijze als HL gebruikt worden.

9.5 Uitdrukkingen in instructies

Tot nu toe hebben we alleen zogeheten 'enkelvoudige' operanden gebruikt, beter gezegd de operand bestond uit een adres, een register(paar) of een label. Een label zouden we als eenvoudigste vorm van een uitdrukking kunnen beschouwen. De operand IX+d is al meer een 'echte' uitdrukking. Een voorbeeld van een operand, waarin een label als deel van een uitdrukking voorkomt, zien we hieronder:

```
LD  A,(BIJNA+1)
```

Deze instructie zorgt ervoor dat de inhoud van de geheugenplaats volgend op de geheugenplaats met label BIJNA in de accumulator geladen wordt.

Dergelijke uitdrukkingen worden doorgaans eenvoudig gehouden. Toch kan en mag zo'n uitdrukking best vrij complex zijn. Deze complexiteit kan ontstaan doordat in zo'n uitdrukking van vele verschillende operatoren (zoals +, *, .AND., .OR., enz.) gebruik mag worden gemaakt. Een overzicht van alle operatoren, waarmee uitdrukkingen gevormd kunnen worden, vindt u in Appendix E.

De microprocessor evalueert zo'n uitdrukking van links naar rechts, waarbij wel prioriteiten worden vastgesteld. Aan de hand van het lijstje uit Appendix E kunnen we het volgende prioriteitenschema

opstellen. Met unaire plus (èn min) wordt bedoeld het plusteken (of minteken) voor een bepaald getal (variabele of uitdrukking), bijvoorbeeld +5 of -10. Dit in tegenstelling tot de + en - tekens die respectievelijk optellen en aftrekken betekenen.

OPERATOR	FUNCTIE	PRIORITEIT
+	unaire plus	1
-	unaire min	1
.NOT. of ~	logische negatie	1
.RES.	resultaat	1
* *	machtsverheffen	2
*	vermenigvuldiging	3
/	deling	3
.MOD.	modulo	3
.SHR.	logische shift rechts	3
.SHL.	logische shift links	3
+	optelling	4
-	aftrekking	4
.AND. of &	logische en	5
.OR. of ^	logische of	6
.XOR.	logische exclusieve of	6
.EQ. of =	gelijk aan	7
.GT. of >	groter dan met teken	7
.LT. of <	kleiner dan met teken	7
.UGT.	groter dan zonder teken	7
.ULT.	kleiner dan zonder teken	7

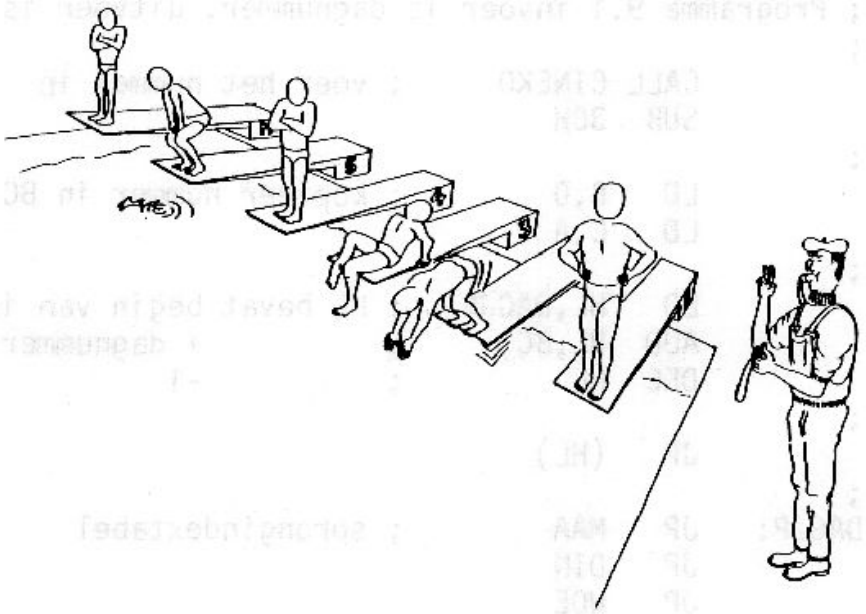
Bij het evalueren van een uitdrukking wordt gebruik gemaakt van 16-bit waarden met teken! Hieronder ziet u een voorbeeld van het gebruik van een uitdrukking in een instructie.

```
LANG:   EQU l
BREED:  EQU b
        -
        -
TABEL:  DEFS LANG * BREED
```

(We gebruiken de labels LANG en BREED hier als namen van twee variabelen met waarden l en b.)

Met een dergelijke instructie kunnen we afhankelijk van de waarden van l en b in de EQU opdrachten een stuk geheugen reserveren dat bestaat uit l×b geheugenplaatsen.

9.6 Sprongindextabel



Het komt in een programma dikwijls voor dat afhankelijk van de waarde van een variabele (label) gesprongen moet worden naar een bepaald stuk in dat programma. In zo'n geval is het handig, zo niet vereist, dat zo'n variabele slechts gehele waarden tussen 1 en n kan aannemen. Voor het springen naar het juiste programmasegment kunnen we in een dergelijke situatie gebruik maken van een zogeheten 'sprongindextabel' (**jump table**). Zo'n jump table werkt als volgt:

```
        -                   ; rp bevat JPTAB + N - 1, N = 1,2,...,n
        JP  (rp)
        -
JPTAB:  JP  N1CODE          ; begin van jump table
        JP  N2CODE
        -
        JP  NNCODE          ; eind van jump table
        -
```

Om de sprongindextabel te kunnen gebruiken moet één van de registerparen IX, IY of HL het adres van één van de spronginstructies uit de tabel bevatten. Daarna wordt de instructie JP (rp) uitgevoerd als gevolg waarvan de gewenste sprong wordt gemaakt, anders gezegd: als gevolg waarvan de juiste spronginstructie uit de tabel wordt uitgevoerd.

In programma 9.1 zien we hoe zo'n sprongindextabel gebruikt wordt

voor het afdrukken van de naam van de dag, waarvan het nummer (1 t/m 7) ingetoetst wordt.

```
; Programma 9.1 invoer is dagnummer, uitvoer is dagnaam
;
        CALL CINEKO     ; voer het nummer in
        SUB  30H
;
        LD   B,0        ; kopieer nummer in BC
        LD   C,A
;
        LD   HL,DAGJP   ; HL bevat begin van indextabel
        ADD  HL,BC      ;          + dagnummer
        DEC  HL         ;          -1
;
        JP   (HL)
;
DAGJP:  JP   MAA        ; sprongindextabel
        JP   DIN
        JP   WOE
        JP   DON
        JP   VRI
        JP   ZAT
        JP   ZON
;
MAA:    LD   HL,MAANDG
        CALL TEKST
        JP   FINI
;
        -
        -
;
ZON:    LD   HL,ZONDAG
        CALL TEKST
        JP   FINI
;
FINI:   HALT
;
MAANDG: DEFM 'MAANDAG'
        DEFB 0
        -
        -
ZONDAG: DEFM 'ZONDAG'
        DEFB 0
```

Het dagnummer, 1,2,3,4,5,6 of 7, dat ingetoetst wordt, representeert de dagen van de week, maandag t/m zondag.

Het adres van de juiste spronginstructie uit de indextabel wordt eerst berekend in het HL register. Het uitvoeren van de juiste sprong volgt direct op het uitvoeren van de JP (HL) instructie. Hierdoor wordt een sprong gemaakt naar dat deel van het programma dat de naam van de, met het ingetoetste nummer, corresponderende dag afdrukt.

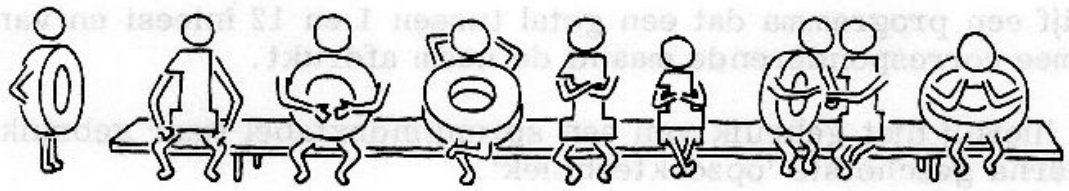
9.7 Een programmeeropdracht

Schrijf een programma dat een getal tussen 1 en 12 inleest en van de daarmee corresponderende maand de naam afdrukt.

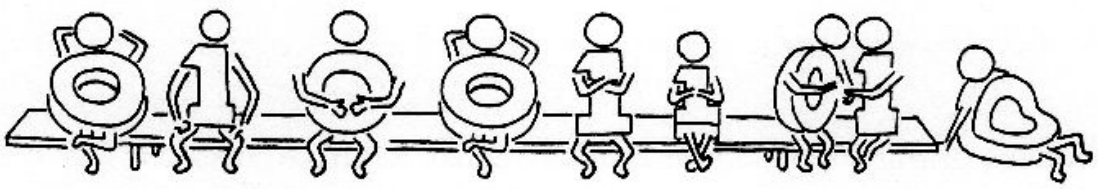
Maak hierbij niet gebruik van een sprongindextabel maar gebruik de hierna geschetste 'opzoektechniek'.

Maak twee datagebieden achteraan in het programma. Een datagebied moet de namen van de maanden bevatten, beginnend met januari en aanéengesloten tot en met december (76 bytes). Daarnaast nog een 12-byte datagebied met per byte de startpositie van de corresponderende naam van de maand uit het eerste datagebied. Deze startpositie relatief ten opzichte van het begin van dat eerste datagebied. (De eerste byte van de 12 zal dan 0 bevatten, de tweede 7, de derde 15, enzovoorts.) Om nu met het ingetoetste getal (1 t/m 12) de naam van de daarmee corresponderende maand te laten afdrukken, wordt dit ingetoetste getal gebruikt om in het tweede datagebied de startpositie (de ingang) van de maand in het eerste gebied te bepalen. Deze positie wordt vervolgens opgeteld bij het beginadres van het eerste datagebied. Tot slot wordt de daar beginnende naam afgedrukt.

10 Shift Instructies, Vermenigvuldigen en Delen

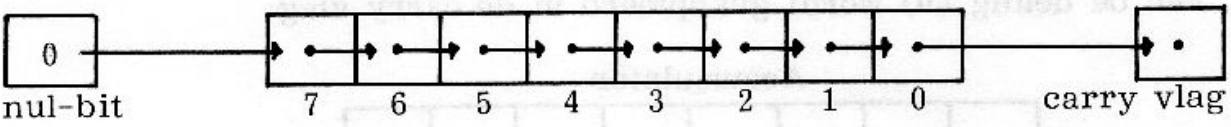


Met shift instructies kunnen we alle bits uit een register of een geheugenplaats één plaats naar links of rechts verschuiven. Er zijn twee soorten shiftinstructies, logische shifts en rekenkundige shifts. Logische shifts (logical shifts) zien de inhoud van een register, op het moment dat de shift plaatsvindt, gewoon als een 8-bit patroon. Rekenkundige shifts (arithmetic shifts) beschouwen de inhoud van een register of geheugenplaats als een twee-complement getal (getal met teken) zodat een shift naar links een vermenigvuldiging van het getal met 2 inhoudt en een shift naar rechts een deling door twee. De Z80 microprocessor kent één logische shift en twee rekenkundige shift instructies. In tabel C.8 uit Appendix C staan deze drie (SRL, SRA en SLA) instructies genoemd.



10.1 De logische shift (SRL) instructie

De logische shift naar rechts (Shift Right Logical) verschuift elk bit in een 8-bit register of geheugenplaats één bitpositie naar rechts. Hierbij wordt het 'uitschuivende' bit gekopieerd in de carry vlag. Tegelijkertijd schuift aan de andere kant een nulbit naar binnen. Bit 7 wordt dus 0.



De algemene vorm van deze logische shift is

```
SRL  m
```

waarbij m een enkelvoudig register voorstelt of een geheugenplaats waarvan het adres in één van de registerparen HL, IX of IY is opgeslagen.

Opgave 10.1
Stel dat de accumulator de waarde A7H bevat en stel dat de carry vlag nul is. Wat zal de waarde van de accumulator en van de carry vlag zijn na het uitvoeren van een SRL A instructie?

Bij alle shift instructies, zowel linker als rechter shifts, wordt het uitschuivende bit in de carry vlag geplaatst. Dit is in veel gevallen handig omdat dit bit vervolgens met een voorwaardelijke sprongopdracht, waarin de carry vlag de voorwaarde is, getest kan worden; bijvoorbeeld JP C,label of JR NC,label.

10.2 De rekenkundige shift naar rechts (SRA) instructie

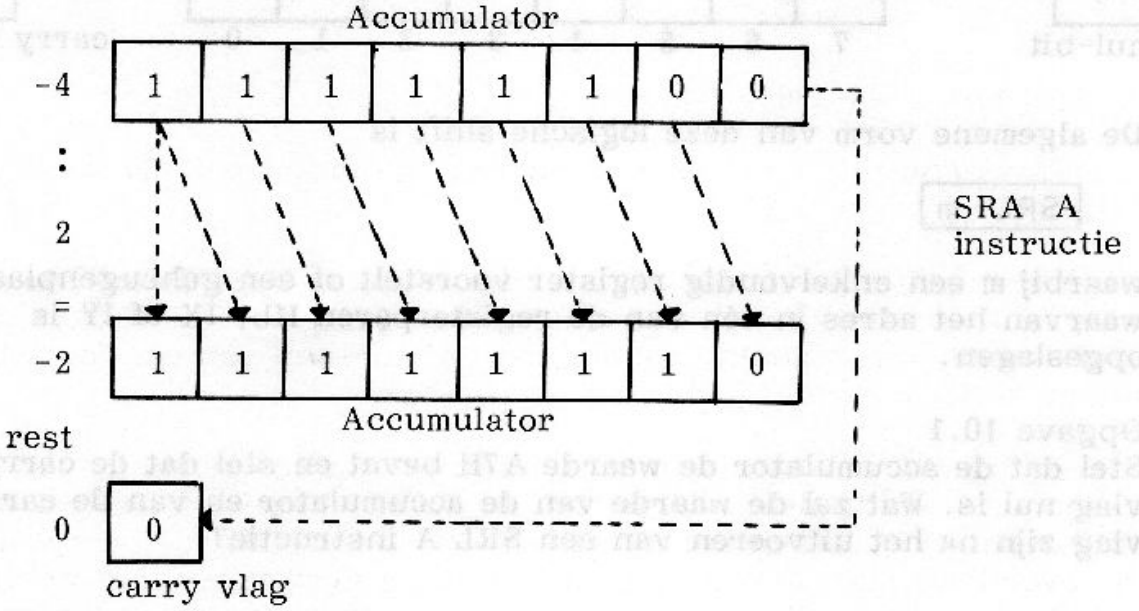
De rekenkundige shift naar rechts (Shift Right Arithmetic) is bijna identiek aan de logische shift naar rechts (SRL). Het verschil is dat bit 7 bij een rekenkundige shift naar rechts onveranderd blijft en niet nul wordt zoals bij een SRL instructie. Dit betekent dat het 'tekenbit' (twee-complement) onveranderd blijft. Een positief getal blijft dus positief, een negatief getal negatief. Het effect van een SRA instructie is dat de inhoud van het register of van de geheugenplaats, genoemd in de SRA-instructie, door twee gedeeld wordt, waarbij de rest van de deling (0 of 1) in de carry vlag geplaatst

wordt. De SRA instructie ziet er als volgt uit:

```
SRA  m
```

m vervult hier dezelfde rol als genoemd bij SRL in § 10.1.

Hieronder zien we een voorbeeld waarbij de inhoud van de accumulator (-4) door een SRA A instructie gedeeld wordt door 2, waarbij het resultaat van de deling (-2) in de accumulator blijft en de rest van de deling (0) wordt gekopieerd in de carry vlag.



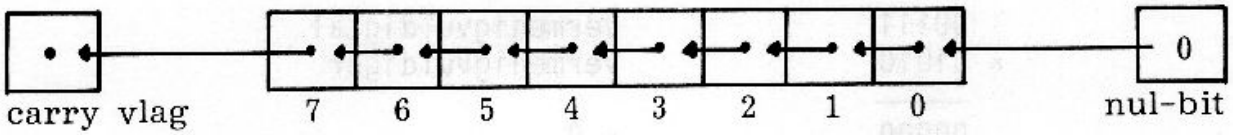
Opgave 10.2
Geef de inhoud van register B, register C en de carry vlag, zowel binair als decimaal, na het uitvoeren van elk van de volgende vier instructies.

```
LD   B,11
SRA  B
LD   C,-8
SRA  C
```

10.3 De rekenkundige shift naar links (SLA) instructie

De SLA instructie verschuift elk bit uit een register of geheugenplaats één bitpositie naar links. Hierbij wordt bit 0 nul gemaakt en schuift bit 7 door naar de carry vlag.

Het effect van deze naar links gerichte rekenkundige shift is dat de inhoud van het register of de geheugenplaats met twee wordt vermenigvuldigd.



Een 'logisch' onderscheid tussen een rekenkundige linker shift en een logische linker shift zou zijn dat bij een rekenkundige shift in bepaalde situaties de overflow vlag gezet wordt, terwijl dit bij een logische linker shift niet het geval zou zijn.
In het volgende voorbeeld:

```
01000001    (65)
  SLA       (×2)
10000010    (-126)
```

treedt overflow op (bit 7 is veranderd), de uitkomst -126 is niet juist.

Nu is het echter zo dat de rekenkundige linker shift de overflow vlag niet zet, hetgeen betekent dat we dus in plaats van rekenkundige linker shift eigenlijk zouden moeten spreken van een logische linker shift.

Als de inhoud van het desbetreffende register (of geheugenplaats) beschouwd wordt als een getal zonder teken (bereik is dan van 0 tot en met 255), zal in het geval van een linker shift de carry vlag als overflow vlag kunnen optreden. Het zal geen verbazing wekken dat de algemene gedaante van de rekenkundige (logische) linker shift instructie er zo uitziet:

```
SLA  m
```

Opgave 10.3
Schrijf met behulp van de SLA en ADD instructies een stukje programma waardoor de inhoud van de accumulator met 10 vermenigvuldigd wordt. Gebruik hierbij het feit dat 10×N gelijk is aan 2×N + 2×2×2×N.

10.4 8-BIT vermenigvuldigen en delen

Vermenigvuldigen door herhaald optellen is erg inefficiënt voor vermenigvuldigers groter dan vijf. Voor grote vermenigvuldigers is er een betere methode die we 'schuif en tel op' zouden kunnen noemen ('shift and add'). Dit principe illustreren we met een voorbeeld:

```
    00111          vermenigvuldigtal
  x 01010          vermenigvuldiger
  -------
    00000          x 0
   00111           x 10
  00000            x 000
 00111             x 1000
00000              x 00000
---------
001000110          produkt
---------
```

Elk bit in de vermenigvuldiger veroorzaakt een shift naar links van het vermenigvuldigtal, waarbij het resultaat bij het lopende produkt wordt opgeteld. Ook zien we dat de waarde die opgeteld moet worden steeds het vermenigvuldigtal (00111) of nul (00000) is.

We kunnen dus voor het vermenigvuldigen volgens de 'schuif en tel op' methode het volgende algoritme opstellen: voor elk bit uit de vermenigvuldiger wordt, werkend van rechts naar links, het vermenigvuldigtal opgeteld bij het deelprodukt, indien het desbetreffende bit uit de vermenigvuldiger 1 is; is dit bit 0 dan doen we niets (tellen 0 op bij het deelprodukt). Vervolgens wordt het vermenigvuldigtal één bitpositie naar links 'geshift' voordat het volgende bit uit de vermenigvuldiger aan de beurt is.

Programma 10.1 laat zien hoe de inhoud van de registers B en C met elkaar vermenigvuldigd worden, waarbij de accumulator het deelprodukt bijhoudt. Tenslotte zal de accumulator ook het 'eind'-produkt bevatten.

```
; programma 10.1 berekent A = B x C
;
        -
        LD    A,Ø
        LD    D,7
NEXBIT: SRL   C          ; test volgende bit uit vermenigvul-
        JP    NC,NOADD   ;                        diger
        ADD   A,B        ; tel vermenigvuldigtal op bij
        JR    PO,OVERFL  ;                        deelprodukt
NOADD:  DEC   D
        JR    Z,KLAAR
        SLA   B          ; shift vermenigvuldigtal
        JR    NEXBIT
        -
```

Dit stukje programma 'doet het alleen goed' voor positieve tweecomplement getallen. Het zou aangepast moeten worden om ook

negatieve getallen te kunnen vermenigvuldigen. Om dit te bewerkstelligen zou je eerst de twee absolute waarden van de getallen kunnen vermenigvuldigen om vervolgens het teken van het produkt, met behulp van de afzonderlijke tekens van de getallen, te bepalen.

De werking van programma 10.1 is als volgt.
Eerst wordt de accumulator nul gemaakt (LD A,0) om het cumulatieve deelprodukt op nul te laten beginnen; vervolgens wordt D geladen met het aantal uit te voeren shifts (7) om alle bits (7) uit de vermenigvuldiger (C) aan bod te laten komen. De lus NEXBIT zorgt ervoor dat steeds het volgende bit uit de vermenigvuldiger (C) naar de carry vlag geschoven wordt en wel met een logische shift naar rechts (SRL). Vervolgens wordt gesprongen naar NOADD indien dit bit nul is (NC), hetgeen betekent dat er niets bij het deelprodukt behoeft te worden opgeteld. Daarna wordt op het vermenigvuldigtal (B) een shift naar links (SLA B) uitgevoerd, natuurlijk alléén indien nog niet alle bits uit de vermenigvuldiger (D > 0) aan de beurt geweest zijn. Dan wordt het volgende bit uit de vermenigvuldiger getest (JR NEXBIT en NEXBIT: SRL C), enzovoorts. Na het vormen van een nieuw deelprodukt wordt op overflow van de accumulator getest (JR PO,OVERFL).

Delen zouden we kunnen doen door herhaald aftrekken of volgens de methode van de welbekende 'staartdeling'. Bij deze laatste methode kijken we of de deler nog van 'dat wat er van het deeltal nog over is' kan worden afgetrokken. Deze methode is in bepaald opzicht gelijk aan de 'schuif en tel op' methode en we zouden dan ook van een 'schuif en trek af' methode kunnen spreken.

10.5 Een programmeeropdracht

Schrijf een nieuwe VERM subroutine die met behulp van de 'schuif en tel op' methode twee twee-complement getallen met elkaar kan vermenigvuldigen.

Schrijf een subroutine DEEL om een getal met teken in register B te delen door een getal met teken in register C, waarbij het quotiënt in de accumulator achterblijft en de 'rest' van de deling in register D te vinden is.

Gebruik de DEEL routine om door 10 te delen. Schrijf een subroutine GETUIT waardoor de inhoud van de accumulator afgedrukt wordt als een decimaal getal tussen -128 en +127. Eventuele nullen vooraan moeten onderdrukt worden en positieve waarden moeten zonder + teken afgedrukt worden, negatieve waarden met een - teken.

Gebruik de GETUIT, DEEL en VERM en eerder geschreven subroutines om een programma te schrijven dat twee getallen met teken, gescheiden door óf een * òf een / en afgesloten met een = teken, inleest en vervolgens òf het produkt van de getallen (als een * teken is gelezen) òf het quotiënt van de getallen met rest (als een / teken is gelezen) afdrukt. In- en uitvoer zouden er zo uit kunnen zien:

```
-1 * 55 = -55
125/10  = 12 REST 5
```

invoer uitvoer

11 Logische Bewerkingen en Macro's

11.1 Logische operatoren

De Z80 kent verscheidene **logische instructies** (logical instructions) waarmee logische bewerkingen (logical operations) tussen overeenkomstige bits uit de accumulator en een 8-bit operand mogelijk zijn.

Om te kunnen begrijpen wat het effect is van logische instructies is het vereist enige kennis te bezitten over de regels rond het gebruik van logische operatoren als **AND**, **OR**, **XOR** (Exclusive OR) en **NOT**. Deze regels worden vaak als volgt weergegeven.

0 AND 0 = 0	0 OR 0 = 0	0 XOR 0 = 0	NOT 0 = 1
0 AND 1 = 0	0 OR 1 = 1	0 XOR 1 = 1	NOT 1 = 0
1 AND 0 = 0	1 OR 0 = 1	1 XOR 0 = 1	
1 AND 1 = 1	1 OR 1 = 1	1 XOR 1 = 0	

Afgezien van de NOT operator werken logische operatoren op twee bits en bestaat het resultaat uit één bit. Zo is het resultaat van de AND operator het 1-bit dan en alléén dan als beide 'ingangbits' ook 1 zijn, anders is het resultaat 0.

In tabel C.5 in Appendix C ziet u de AND, OR en XOR instructies gedefinieerd. Merk op dat in de tweede kolom van deze tabel de symbolen $\wedge$ voor AND, $\vee$ voor OR en $\oplus$ voor XOR gebruikt worden. Dit zijn symbolen die in de booleaanse algebra gebruikt worden.

Opgave 11.1
Geef een omschrijving van de werking van de XOR operator in 'gewoon Nederlands'.

11.2 Logische instructies

Elke logische instructie die de Z80 kent, verricht een logische bewerking op een bit uit de accumulator en op een overeenkomstig bit uit de operand (opgegeven in de instructie) waarbij het resultaat in de accumulator achterblijft.

Indien de accumulator 00001010B bevat en register B 11001111B, zal na de logische instructie AND B de inhoud van de accumulator 00001010B zijn, ongewijzigd dus.

Inhoud van A	00001010B
Inhoud van B	11001111B
Inhoud van A na AND B	00001010B

De logische AND operator werkt dus bit voor bit; elk bit onafhankelijk van de andere bits.

De operand in een logische instructie kan bestaan uit een register, een 8-bit waarde of een geheugenplaats aangewezen door HL, IX of IY. Dit kunnen we als volgt weergeven:

AND, OR, XOR	r
	n
	(HL)
	(IX)
	(IY)

Het gebruik van een logische instructie impliceert altijd het gebruik van de accumulator als één van de twee operanden waarop de instructie werkt, vandaar dat alleen de tweede operand opgegeven dient te worden.

De logische operator NOT wordt geactiveerd door de instructie CPL (ComPLement) waardoor alle nullen één en alle énen nul worden in de accumulator.

Opgave 11.2
Geef de inhoud van de accumulator, de S, de Z en de C vlag, als binaire waarden na het uitvoeren van elk van de volgende instructies.

```
LD   A,10110101B
LD   C,11110000B
AND  00011111B
OR   C
XOR  11001100B
CPL
```

11.3 Maskeren

Masker voor het verbergen van de bits 1, 2, 4 en 7.

Een van de belangrijkste toepassingen van de AND operator treffen we aan bij het zogeheten maskeren (masking). Maskeren doen we als we slechts een paar bits uit een 8-bit waarde nodig hebben. De ongewenste bits verbergen we als het ware achter een 'masker'.

Als we een teken dat een cijfer voorstelt willen omzetten in de 'waarde' van dat cijfer, kan dat met behulp van een aftrekking (SUB), maar het kan ook met maskeren. De waarde van een cijfer wordt gevormd door de laatste vier bits uit de code van dat cijfer. We moeten dus een masker maken waarmee we alléén die laatste vier bits uit de 8-bit code te pakken krijgen. Dit kan door deze 8-bit code te ANDen met het masker 00001111B; dat gaat zo:

```
LD   A,'7'      ; accumulator bevat 00110111B (7 als teken)
AND  0FH        ;          geAND met 00001111B (masker)
                ;   geeft de waarde 00000111B (7 als waarde)
```

Opgave 11.3
Schrijf een logische instructie waarmee de waarde van een cijfer (0 t/m 9) omgezet wordt in de code voor dat cijfer, in een teken dus.

11.4 Macro's

Met een macro kan de programmeur een eigen OP-code definiëren. Stel dat in een programma de carry vlag nogal vaak op 0 gezet moet worden. We kunnen nu aan het begin van het programma een OP code (een instructie) definiëren die voor ons de carry vlag op 0 zet. We zeggen dan dat we een macro definiëren:

```
RSF:  MACRO     ; reset-carryvlag-macro
      SCF       ;
      CCF       ; } macro-body
      ENDM      ; einde macrodefinitie
```

De MACRO pseudo-instructie vertelt de assembler dat we een macro met de naam RSF gaan definiëren. De instructies tussen de pseudo-instructies MACRO en ENDM noemen we de macro-body.

Hebben we een macro gedefinieerd, dan kunnen we overal in het programma de OP code RSF gebruiken; anders gezegd: we kunnen overal de macro aanroepen. Een voorbeeld hiervan is:

```
LD   (POINT),A
RSF                  ; reset-carryvlag-macro-aanroep
ADD  A,B
```

Dit is een stukje source code. De assembler zal tijdens het assembleren van de source code overal waar RSF staat deze OP code vervangen door de macro-body van de RSF-macro. Na assemblage ziet bovenstaand stukje programma er dan ook zo uit:

```
LD   (POINT),A
SCF                  ; macro RSF
CCF                  ; expansie
ADD  A,B
```

Opgave 11.4
Schrijf een macro om de accumulator met vier te vermenigvuldigen en geef een voorbeeld waarin deze macro wordt aangeroepen.

We zien een toenemend gebruik van macro's in assembleertaalprogramma's. Met macro's kunnen we enige structuur aanbrengen in dit soort programma's, iets wat bij hogere programmeertalen vanzelfsprekender, of zelfs vereist, is. In programma 11.1 worden macro's gebruikt voor het enigszins structureren van programmalussen.

In dit programma worden twee macro's, LUSIN en LUSUIT, de eerste voor de start van een lus, de tweede voor het einde daarvan, gebruikt. Ook zien we in dit voorbeeld dat beide macro-definities parameters bevatten. Alle in een macro te gebruiken parameters moeten achter de MACRO pseudo-instructie worden opgegeven en ze mogen overal in de macro-body gebruikt worden. Als we een macro met parameters aanroepen, worden tijdens het expanderen van de macro de parameters vervangen door de werkelijke parameters uit de macro-aanroep. Zo zal de laatste macro-aanroep in programma 11.1 (LUSUIT 'E','WEER') in de source code tijdens assemblage geëxpandeerd worden tot

```
DEC  E
JP   NZ,WEER
```

Hierin zijn dus de macro-parameters #R en #LABEL na het assemble-

ren vervangen door de werkelijke parameters E en WEER uit de macro-aanroep.

Parameters in een macro-definitie worden voorafgegaan door een hekje #, terwijl parameters in een macro-aanroep tussen apostroffen staan.

```
; Programma 11.1 gestructureerde lussen met gebruik van macro's
;
LUSIN:  MACRO #R,#N          ; macro voor begin van lus
        LD    #R,#N
        ENDM
;
LUSUIT: MACRO #R,#LABEL      ; macro voor einde van lus
        DEC   #R
        JP    NZ,#LABEL
        ENDM
;

        LUSIN  'C','99'
NEXTHG: -
        -
        LUSUIT 'C','NEXTHG'
        -
        LUSIN  'E','18'
WEER:   -
        -
        LUSUIT 'E','WEER'
        -
```

Macro-definities mogen niet genest worden, maar we mogen wel in een macro-body een 'eerder' gedefinieerde macro aanroepen.

Om in een macro van lokale labels gebruik te kunnen maken is er voorzien in een speciale symbolengenerator. We kunnen dit het beste aan de hand van een voorbeeld uitleggen. Kijk eens naar de volgende macro-definitie:

```
TIMER:  MACRO #N
        LD    B,#N
TM#$YM: DJNZ  TM#$YM
        ENDM
```

Deze macro geeft ons een nogal onnauwkeurige timing mogelijkheid en wel door het van N naar nul aftellen van register B.

Voor de lus in de macro is een lokaal label nodig. Zou de naam van

de label op de normale wijze gekozen wordt, bijvoorbeeld TIMLUS, dan zou bij meer dan één macro-aanroep tijdens het expanderen deze naam meer dan één keer in het programma voorkomen, hetgeen niet is toegestaan!
Dit probleem is als volgt opgelost. De laatste vier tekens van de naam van een lokaal label zien er uit als #$YM. Als nu de macro geëxpandeerd wordt, vervangt de assembler deze vier tekens door een '4-cijferig' hexadecimaal getal, beginnend met 0000 voor de eerste macro-expansie. Bij de volgende expansie (macro-aanroep) wordt dit getal met één verhoogd (0001), hetgeen zich herhaalt bij elke volgende macro-aanroep.

In het volgende voorbeeld zien we drie opeenvolgende expansies van de timer-macro-aanroep:

```
        -
        TIMER '200'
        -
        TIMER '50'
        -
        TIMER '76'
```

wordt geëxpandeerd tot

```
        -
        LD   B,200
TM0000: DJNZ TM0000
        -
        LD   B,50
TM0001: DJNZ TM0001
        -
        LD   B,76
TM0002: DJNZ TM0002
        -
```

Er is een groot verschil tussen macro's en subroutines. De instructies uit een subroutine-body worden slechts éénmaal in een programma opgenomen, zowel voor als na het assembleren. De instructies uit een macro-body worden na het assembleren even vaak in het programma opgenomen als de macro wordt aangeroepen. Dit betekent dat, naarmate een macro vaker wordt aangeroepen, het wenselijk is dat de macro-body minder instructies bevat, want anders wordt een programma te groot en dus te langzaam. In de regel bevat een macro-body slechts een handjevol instructies.

Elk Z80 microprocessorsysteem heeft (jammer genoeg) eigen regels voor het definiëren en aanroepen van macro's. Kijk in uw eigen Z80 assembleertaalhandleiding wat voor uw systeem de voorschriften met betrekking tot macro's zijn.

11.5 Voorwaardelijke pseudo-instructies

Met voorwaardelijke pseudo-instructies kunnen we de assembler vertellen dat slechts onder bepaalde voorwaarden bepaalde stukken uit een source programma geassembleerd moeten worden. De twee pseudo-instructies waarmee dit mogelijk is zijn COND en ENDC. Hieronder zien we een stukje source code:

```
-
COND  CODEIN
-
-
ENDC
-
```

De instructies tussen de COND en ENDC pseudo-instructies zullen alleen door de assembler geassembleerd worden als de waarde van CODEIN ongelijk nul is. Is CODEIN nul dan wordt dit stukje programma genegeerd en dus niet geassembleerd. CODEIN is een label van een geheugenplaats.

Met de DEFL (DEFine Label) pseudo-instructie wordt gewoonlijk een waarde aan de label, achter de COND pseudo-instructie, toegekend. Als bijvoorbeeld de instructie

```
CODEIN:  DEFL 1
```

in het bovenstaande programma zou zijn opgenomen vóór de COND CODEIN pseudo-instructie, zullen de instructies tussen COND en ENDC worden geassembleerd, immers CODEIN = 1, dus ongelijk nul.

Alhoewel de voorwaarde voor het al of niet assembleren doorgaans afhangt van de waarde van een label, zoals bij CODEIN hierboven, mogen we ook in plaats van een label een rekenkundige of logische uitdrukking gebruiken.

Opgave 11.5
Een bepaald programma kan geassembleerd worden voor het verzorgen van uitvoer op een beeldscherm of voor het afdrukken van de uitvoer op een regeldrukker.

Het programma bevat twee subroutines, met dezelfde naam, één voor het sturen van de uitvoer naar een beeldscherm en de andere voor het afdrukken van de uitvoer op een regeldrukker. Laat zien hoe de voorwaardelijke pseudo-instructies COND en ENDC en de pseudo-instructie DEFL in de desbetreffende programma-onderdelen gebruikt zouden kunnen worden om òf de éne subroutine te laten assembleren òf de andere.

De DEFL pseudo-instructie doet hetzelfde als de EQU pseudo-instructie, namelijk het toekennen van een waarde aan een label. Toch is er een verschil. De EQU pseudo-instructie mag voor een bepaald label maar één keer in een programma worden opgenomen. De DEFL instructie daarentegen mag vaker gebruikt worden om binnen één programma op verschillende plaatsen verschillende waarden aan hetzelfde label toe te kennen. Dit geeft een flexibeler stuk gereedschap om de techniek van 'voorwaardelijke assemblage' toe te passen.

Het zal duidelijk zijn dat de voorwaardelijke pseudo-instructies zeer geschikt zijn om een algemeen programma te schrijven, dat toch in verschillende situaties bruikbaar is.

11.6 Een programmeeropdracht

Er bestaan nog twee logische operatoren. Dit zijn NOR en NAND, afkortingen voor NOT OR en NOT AND. Hieronder zien we de logische regels voor deze operatoren.

0 NOR 0 = 1	0 NAND 0 = 1
0 NOR 1 = 0	0 NAND 1 = 1
1 NOR 0 = 0	1 NAND 0 = 1
1 NOR 1 = 0	1 NAND 1 = 0

Hieruit kunnen we opmaken, dat het resultaat van een NOR bewerking het complement is van het OR resultaat, dus NOT OR. Zo is het resultaat van NAND de NOT van een AND bewerking.

Schrijf een Logische-Operator-Oefen-Programma dat als invoer kan accepteren:

b lop b = b

waarin b een 0 of 1 is en lop één van de volgende logische operatoren:

OR
AND
XOR
NOR
of NAN (afkorting van NAND)

De invoer is dus b lop b = b. De uitvoer als reactie op een dergelijke invoer moet zijn (op dezelfde regel als de invoer):

TRUE als de invoer logisch correct is en
FALSE als de invoer niet logisch correct is.

Zo kan de gebruiker zichzelf testen of hij (of zij) de werking van de logische operatoren onder de knie heeft.

12 Roteerinstructies en Pariteit

Roteerinstructies onderscheiden zich van shiftinstructies doordat bij roteren het uitschuivende bit aan de andere kant wordt ingeschoven. Vandaar ook de naam 'roteren'.

De Z80 microprocessor bezit verscheidene roteerinstructies. Vier daarvan betreffen de accumulator, de andere hebben betrekking op een register of geheugenplaats.

Sommige roteerinstructies betrekken de carry vlag bij de rotatie, terwijl andere roteerinstructies dat niet doen. In het eerste geval kunnen we spreken van een 9-bit rotatie (8 bit + carry vlag), in het laatste geval van een 8-bit rotatie.

Net als bij shift-instructies zijn er linker en rechter roteerinstructies en er wordt steeds 1 bitpositie geroteerd. Alle rotaties vallen onder de noemer 'logische shift', er zijn dus géén rekenkundige rotaties!

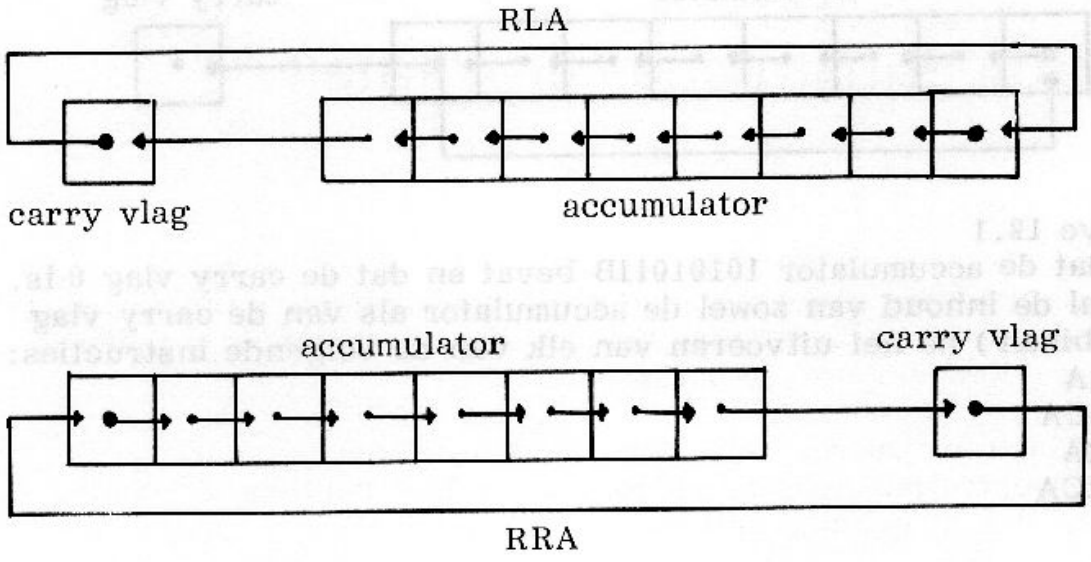
In tabel C.8 van Appendix C staat de beschrijving van alle Z80 roteerinstructies.

12.1 Accumulator-rotatie

De accumulator kan links en rechts geroteerd worden, waarbij al dan niet de carry vlag meeroteert. De vier accumulator-roteerinstructies zijn:

RLA	-	roteer accumulator links
RRA	-	roteer accumulator rechts
RLCA	-	roteer accumulator links cyclisch
RRCA	-	roteer accumulator rechts cyclisch

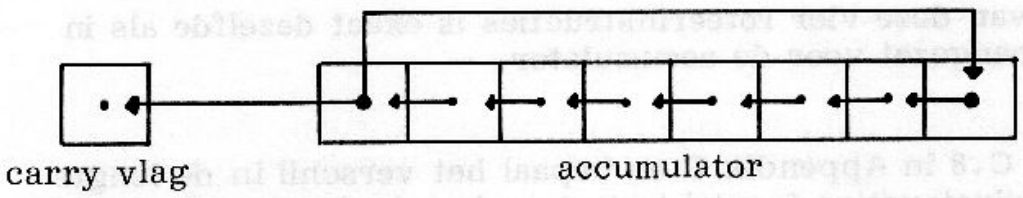
De RLA (Rotate Left Accumulator) en de RRA (Rotate Right Accumulator) betrekken beide de carry vlag bij de rotatie en wel als volgt:



Bij de RLA instructie worden alle bits uit de accumulator één bitpositie naar links verschoven; hierbij komt bit 7 van de accumulator in de carry vlag, terwijl de vlag zelf naar bit 0 van de accumulator geschoven (geroteerd) wordt.

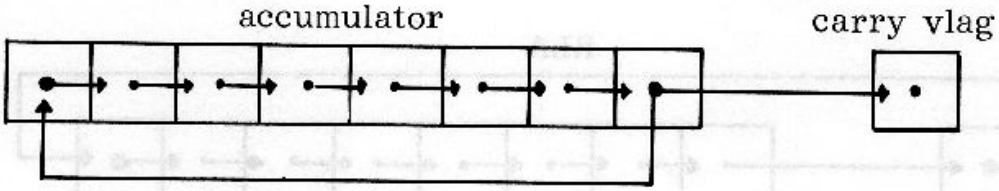
Bij de RRA instructie verschuiven alle bits van de accumulator één bitpositie naar rechts. Bit 0 verschuift naar de carry vlag en de carry vlag verschuift naar bit 7 van de accumulator.

De RLCA (Rotate Left Circular Accumulator) instructie is een cyclische rotatie van alleen de accumulatorbits en daarom doet de carry vlag niet mee. Toch is ook de carry vlag bij deze instructie betrokken en wel op de volgende manier:



De inhoud van de accumulator schuift één bit naar links op. Bit 7 van de accumulator roteert naar bit 0 van dezelfde accumulator, maar schuift tevens naar de carry vlag. Na een RLCA instructie bevatten de carry vlag en bit 0 van de accumulator altijd dezelfde waarde, namelijk de waarde van bit 7 vóór het uitvoeren van de RLCA instructie.

Op dezelfde wijze, maar dan de 'andere kant' om, werkt de RRCA instructie:



Opgave 12.1
Stel dat de accumulator 10101011B bevat en dat de carry vlag 0 is. Wat zal de inhoud van zowel de accumulator als van de carry vlag zijn (binair) na het uitvoeren van elk van de volgende instructies:

RLA
RLCA
RRA
RRCA

12.2 Register- en geheugenplaats-rotatie

De vier soorten rotaties die in § 12.1 zijn behandeld kunnen we ook uitvoeren op een register of geheugenplaats. Hier zijn ze:

RL m	- roteer register of geheugenplaats links
RR m	- roteer register of geheugenplaats rechts
RLC m	- roteer register of geheugenplaats links cyclisch
RRC m	- roteer register of geheugenplaats rechts cyclisch

m is één van de enkelvoudige registers of een door HL, IX of IY aangewezen geheugenplaats.

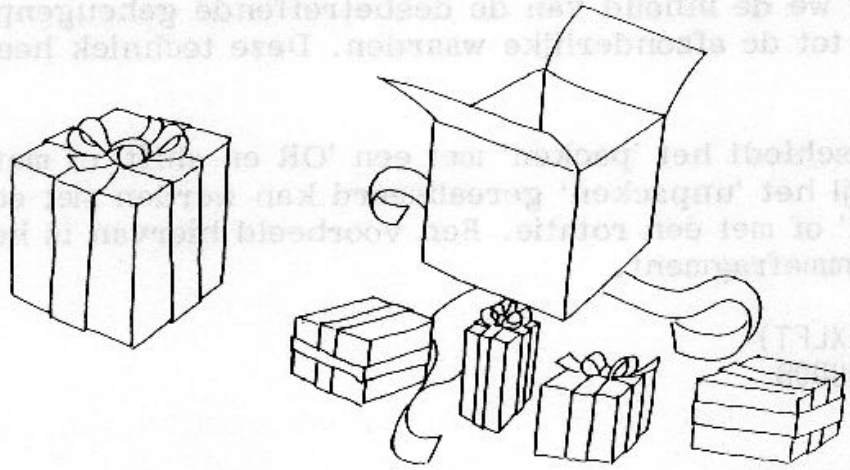
De werking van deze vier roteerinstructies is exact dezelfde als in § 12.1 is uiteengezet voor de accumulator.

Opgave 12.2
Kijk in tabel C.8 in Appendix C en bepaal het verschil in de lengte van de roteerinstructies (aantal bytes) en hun invloed op de diverse vlaggen. Doe dit voor de volgende twee groepen instructies:

RLCA		RLC
RLA		RL
RRCA	èn	RRC
RRA		RR

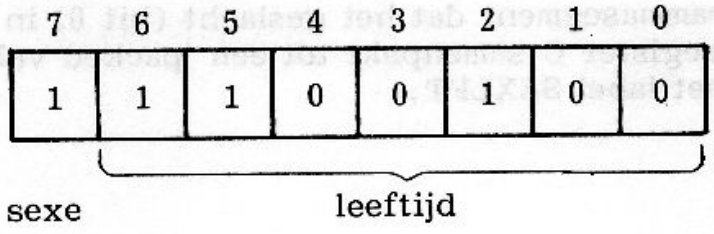
Omdat een roteer-accumulator-instructie slechts één byte in beslag neemt en een shift-accumulator-instructie twee bytes, gebruiken sommige programmeurs een roteerinstructie om een shift uit te voeren. Tegenover het uitsparen van één byte, hetgeen de tijd nodig voor het uitvoeren van de instructie korter maakt, staat het minder begrijpelijk worden van het programma. Immers daar waar je een shift-instructie zou verwachten staat een roteerinstructie!

12.3 Packing en unpacking



Packing is een techniek waarbij verschillende gegevens in één register of geheugenplaats worden samengepakt. Eigenlijk hebt u allang kennis gemaakt met deze techniek. Het vlagregister is hiervan een uitstekend voorbeeld. In het vlagregister (één byte) zijn namelijk zes verschillende stukjes data samengepakt, te weten de afzonderlijke vlaggen. Het vlagregister wordt haast nooit als één waarde gezien; we bekijken steeds één vlag (bit) uit het register. Dit samenpakken kunnen we ook zelf in onze programma's toepassen en we doen dat dan om geheugenruimte te sparen.

Zo zouden we in een bepaald programma bijvoorbeeld leeftijd en geslacht van een persoon kunnen samenpakken in één geheugenplaats en wel als volgt:



Bit 7 geeft het geslacht aan, bijvoorbeeld 0 = man en 1 = vrouw. Bit 6 t/m 0 geeft de leeftijd aan; daarmee kunnen we dus leeftijden tot 1111111B = 127 jaar coderen. In bovenstaand voorbeeld betekent de inhoud van de desbetreffende geheugenplaats dat het om een 100-jarige vrouw gaat.

Als we even aannemen dat op een bepaald moment de leeftijd en het geslacht van 1000 personen in het geheugen zou moeten worden opgeslagen, dan kunnen we door dit samenpakken (packing) 1000 bytes uitsparen, omdat we nu per persoon maar één byte in plaats van twee nodig hebben voor de registratie van leeftijd en geslacht.

Om van de waarden van leeftijd en geslacht gebruik te kunnen maken moeten we de inhoud van de desbetreffende geheugenplaats uiteenrafelen tot de afzonderlijke waarden. Deze techniek heet **unpacking**.

Doorgaans geschiedt het 'packen' met een 'OR en shift' of met een rotatie, terwijl het 'unpacken' gerealiseerd kan worden met een 'AND en shift' of met een rotatie. Een voorbeeld hiervan is het volgende programmafragment.

```
LD   A,(SEXLFT)
AND  10000000B
RLCA
LD   B,A
LD   A,(SEXLFT)
AND  01111111B
LD   C,A
```

De inhoud van de geheugenplaats met label SEXLFT wordt in de accumulator geladen. Door met een AND bit 7 uit de accumulator te maskeren en vervolgens de accumulator links cyclisch te roteren bevat de accumulator tenslotte de waarde 00000001. Deze waarde wordt in register B gekopieerd. Register B bevat dus na het uitvoeren van LD B,A de waarde van het geslacht. Op een dergelijke manier wordt vervolgens de leeftijd in register C opgeslagen. Hoe dit unpacken moet gebeuren hangt dus af van het aantal samengepakte waarden, van de 'lengte' van deze waarden en van hun positie.

Opgave 12.3
Schrijf een programmasegment dat het geslacht (bit 0) in register B en de leeftijd in register C samenpakt tot een 'packed value' in de geheugenplaats met label SEXLFT.

12.4 Pariteit

Met pariteit wordt het aantal énen in een binaire waarde aangeduid. Een binaire waarde heeft een even pariteit als het aantal énen in die waarde even is en oneven pariteit als dit aantal oneven is.

01101000	heeft oneven pariteit
11111100	heeft even pariteit
01110011	heeft oneven pariteit

In computersystemen vindt een voortdurend transport van gegevens plaats. Bij dit 'overzenden' wordt gebruik gemaakt van het begrip pariteit. Aan de over te zenden gegevens wordt aan de bron een pariteitsbit toegevoegd en wel om het totaal aantal énen of even of oneven te maken. Aan de kant waar de gegevens ontvangen worden wordt vervolgens getest of de pariteit nog altijd even of oneven is. Zo niet dan is er tijdens het transport iets gebeurd!

Deze zogeheten pariteitscheck is echter geen waterdichte test voor de kwaliteit van het gegevenstransport. Als gedurende het transport namelijk een even aantal bits van waarde verandert, blijft de pariteit gelijk, maar er is wel iets fout gegaan. Toch is een dergelijke kwaliteitstest beter dan helemaal geen test.

Opgave 12.4
Stel dat we een 'evenpariteitscheck' hebben en stel dat bit 7 van een byte als pariteitsbit dienst doet. Wat zal de hexadecimale inhoud zijn van een geheugenplaats die
a. de ASCII code van de letter X en
b. de ASCII code van het + teken bevat,
indien het pariteitsbit zorgt voor de juiste pariteit.

12.5 De pariteitsvlag

De pariteit/overflow (P/V) vlag wordt gebruikt om de pariteit aan te geven na het uitvoeren van roteer-, shift- en logische instructies. Niet alle roteerinstructies zetten echter deze vlag (zie tabel C.8 in Appendix C).

Indien het aantal énen in een register of geheugenplaats na het uitvoeren van één van bovengenoemde instructies even is, wordt de pariteitsvlag gezet, 1 gemaakt dus. Is het aantal énen echter nul, dan wordt de P/V vlag 0.

Opgave 12.5
Stel de accumulator bevat de waarde B9H. Geef de waarde van de P/V vlag na elk van onderstaande instructies.

```
AND ØFEH
SLA A
RLA
```

De pariteitsvlag kan getest worden met de volgende voorwaardelijke instructies:

```
JP    PE,label
JP    PO,label
CALL  PE,label
CALL  PO,label
RET   PE
RET   PO
```

PE is even pariteit (Parity Even) en PO is oneven pariteit (Parity Odd).

Opgave 12.6
Schrijf een subroutine CHKPAR waarmee de pariteit van de accumulator getest kan worden. Bij aanroep van de subroutine bevat het register B 0 om even pariteit aan te geven of 1 om oneven pariteit aan te geven. Na terugkeer uit de subroutine moet dit register 0 bevatten, indien de pariteit van de accumulator overeenkwam met de inhoud van B en 1 als dit niet het geval is.

12.6 Een programmeeropdracht

Schrijf een subroutine BINOUT waarmee de inhoud van de accumulator als acht énen en nullen (bits) op het beeldscherm kan worden afgedrukt. De subroutine pakt bit voor bit uit de accumulator en geeft de code van het teken overeenkomend met de waarde van het bit (0 of 1) als uitvoer.

Schrijf een subroutine PACK waarmee vier waarden uit opeenvolgende geheugenplaatsen in de accumulator worden samengepakt. Het 'packen' moet als volgt geschieden:

```
bits 1 en 0    uit M   naar bits 7 en 6    van de accumulator
bits 1 en 0    uit M+1 naar bits 5 en 4    van de accumulator
bit  0         uit M+2 naar bit 3          van de accumulator
bits 2, 1 en 0 uit M+3 naar bits 2, 1 en 0 van de accumulator
```

M stelt de eerste van de vier geheugenplaatsen voor.

Bij aanroep van de subroutine PACK bevindt het adres van M zich in het HL register.

Schrijf vervolgens een routine UNPACK die precies het tegenovergestelde doet van de PACK routine.

Gebruik tenslotte de subroutines BINOUT, PACK en UNPACK en reeds eerder geschreven subroutines om een programma te maken waarin tien groepen van vier getallen worden ingelezen en waarbij elke groep volgens de routine PACK in de accumulator wordt samengepakt. De vier getallen uit een groep worden als decimale getallen met de volgende waarden ingevoerd:

eerste getal : 0, 1, 2 of 3
tweede getal : 0, 1, 2 of 3
derde getal : 0 of 1
vierde getal : 0, 1, 2, 3, 4, 5, 6 of 7

De tien 'samengepakte' waarden moeten op tien opeenvolgende geheugenplaatsen worden opgeslagen. Het programma moet deze tien samengestelde waarden ook afdrukken; dit om te kunnen constateren of het programma het 'goed' doet.

13 Rekenen met 16 Bits en meer

Tot nu toe hebben we gerekend met 8-bit waarden, dat wil zeggen met 8-bit operanden en 8-bit resultaten. Dit rekenen met 8-bit waarden betekent een grote beperking ten aanzien van de waarden waarmee we kunnen rekenen. Een 8-bit twee-complement waarde ligt zoals u weet in het bereik van -128 tot +127. Willen we met grotere getallen kunnen rekenen (of met meer cijfers achter de komma!), dan moeten we kunnen rekenen met 16-bit waarden, 32-bit waarden of zelfs met 'nog-meer-bit' waarden.

De Z80 microprocessor heeft een aantal instructies waarmee rechtstreeks '16-bits' gerekend kan worden. Deze instructies kunnen we ook gebruiken om rekenkundige bewerkingen op/met 32 bit of 48 bit waarden uit te voeren. De 16-bit rekenkundige instructies bieden daarboven nog enkele nieuwe mogelijkheden voor het werken met programmalussen.

Alle 16-bit rekenkundige instructies zijn opgenomen in tabel C.7 in Appendix C.

13.1 De DEFW pseudo-instructie

We hebben tot nu toe de DEFB (DEFine Byte) pseudo-instructie gebruikt om in het datagebied achterin een programma 8-bit waarden te definiëren. Willen we met 16-bit waarden kunnen werken, dan moeten we in staat zijn twee bytes als één waarde te definiëren. We spreken dan van woorden (1 word = 2 bytes). Een woord komt dus overeen met twee bytes. Zo'n 16-bit woord definiëren we met de DEFW (DEFine Word) pseudo-instructie, waarvan hieronder een voorbeeld:

```
DUBBEL:  DEFW  56ABH
```

Met bovenstaande opdracht kennen we de 16-bit waarde 56ABH toe aan de label DUBBEL. Om precies te zijn wordt DUBBEL geassocieerd met de eerste byte uit het woord en wel met de waarde ABH, in de daaropvolgende geheugenplaats komt de tweede byte uit het woord, te weten de waarde 56H.

Dat ABH eerst komt, vloeit voort uit de wijze waarop instructies de inhoud van twee opeenvolgende geheugenplaatsen (een woord) in een registerpaar laden. Zo zal de instructie LD HL,(LABEL) het register L laden met de inhoud van de eerste geheugenplaats (LABEL), terwijl register H geladen wordt met de inhoud van de daaropvolgende geheugenplaats (LABEL+1).

Door de DEFW pseudo-instructie te gebruiken hoeven we ons niet druk te maken over die bepaalde volgorde. De LD HL,(LABEL) instructie in het volgende voorbeeld

```
        LD   HL,(LABEL)
        -
LABEL:  DEFW 7B9AH
```

zorgt er gewoon voor dat het HL register geladen wordt met de waarde 7B9AH; u had waarschijnlijk ook niets anders verwacht!

13.2 De 16-bit ADD, ADC en SBC instructies

De meeste 16-bit rekenkundige instructies gebruiken als 'accumulator' het HL register(paar). Zo zal de 16-bit optelling

```
ADD  HL,DE
```

de inhoud van het (16-bit) DE register optellen bij de inhoud van het HL register en het resultaat ook in dit laatste register achterlaten. De algemene vorm van deze 16-bit optelling is

```
ADD HL,ss
```

waarin ss één van de registerparen BC, DE, HL of SP voorstelt.

In het volgende programmafragment zien we hoe twee 16-bit getallen bij elkaar opgeteld worden in het HL register. Na het uitvoeren van de laatste instructie zal HL de waarde 3416 bevatten.

```
LD  BC,2054
LD  HL,1362
ADD HL,BC
```

Opgave 13.1
Wat is het bereik van een 16-bit getal met teken?

Er is een 16-bit optelinstructie waarmee de carry, opgetreden als gevolg van een eerder uitgevoerde bewerking, mee opgeteld wordt. De algemene vorm van deze ADd met Carry is

```
ADC HL,ss
```

voor ss zie hierboven

De inhoud van het ss registerpaar wordt samen met de carry vlag opgeteld bij de inhoud van HL om zo de nieuwe inhoud van HL te vormen.

Met de ADC HL,ss instructie kunnen we op eenvoudige manier '32-bits' rekenen. Bekijk hiertoe programma 13.1, waarin twee 32-bit getallen (met teken) bij elkaar worden opgeteld.

```
; Programma 13.1 het optellen van twee 32-bit getallen (met
;                teken)
;
        LD   HL,(N1LS)
        LD   DE,(N2LS)
        ADD  HL,DE          ; tel minst significante 16-bits op
        LD   (RESLS),HL
        LD   HL,(N1MS)
        LD   DE,(N2MS)
        ADC  HL,DE          ; tel meest significante 16-bits op
        LD   (RESMS),HL
        JP   PO,OVERF       ; overflow?
        -
```

```
N1MS:   DEFW 05A1H          ; eerste getal
N1LS:   DEFW 63B2H
N2MS:   DEFW 00C6H          ; tweede getal
N2LS:   DEFW 0A57EH
RESMS:  DEFW 0              ; resultaat
RESLS:  DEFW 0
```

Eerst worden de minst significante 16 bits uit de twee 32-bit getallen bij elkaar opgeteld met de ADD HL,DE instructie. Het resultaat van deze optelling wordt opgeslagen in het geheugenwoord dat hiervoor gereserveerd wordt (RESLS). Gedurende deze optelling zal een mogelijke carry van het meest significante bit opgenomen worden in de carry vlag. Dit laatste betekent dat we eigenlijk 17 bits nodig hebben om het resultaat van het optellen van de 16 minst significante bits weer te geven. Deze carry moeten we wel onthouden, want deze moet opgeteld worden bij de optelling van de 16 meest significante bits van de twee getallen. Met andere woorden: we moeten nu een ADC HL,DE instructie gebruiken, waardoor de carry mee opgeteld wordt.

Zo zal dit programmafragment twee 32-bit getallen (met teken) optellen. Er wordt op overflow getest na het uitvoeren van de ADC instructie.

Opgave 13.2
Wat zal na het uitvoeren van de laatste instructie uit programma 13.1 de inhoud, opgegeven als hexadecimaal getal, zijn van RESMS en RESLS?

Bovenstaande techniek kan ook gebruikt worden voor het optellen van 16×n-bit getallen, met n groter dan 2!

De Z80 kent slechts één 16-bit aftrekking. Dit is een 'aftrekking met carry'. De algemene gedaante van deze 'aftrek met carry' instructie is

SBC HL,ss	ss is BC, DE, HL of SP

Deze instructie zorgt ervoor dat zowel de inhoud van het registerpaar ss alsmede de carry vlag afgetrokken wordt van HL, waarbij het resultaat van de bewerking in het HL register achterblijft.

Ook kan de SBC HL,ss instructie gebruikt worden voor 32-bit, 48-bit, en nog-hogere-bit aftrekkingen.

Als we 'echte' 16-bit getallen van elkaar willen aftrekken, moeten

we ervoor zorgen dat vlak voor het uitvoeren van de SBC HL,ss instructie de carry vlag op 0 gezet wordt; dit om ervoor te zorgen dat de uitkomst juist is!

Opgave 13.3
Schrijf een stukje programma waarmee de inhoud van het BC register van HL wordt afgetrokken. Er wordt gewerkt met 16-bit getallen.

Belangrijk bij 16-bit ADD, ADC en SBC instructies is de wijze waarop deze instructies de diverse vlaggen beïnvloeden. Als u in tabel C.7 in Appendix C naar de beschrijvingen van deze instructies kijkt, zult u zien dat de 16-bit ADC en SBC instructies de carry-, zero-, overflow- en sign-vlag zetten (u had wellicht niet anders verwacht), maar dat de 16-bit ADD instructie slechts de carry vlag zet. Willen we dat bij een 'gewone' 16-bit optelling de zero-, overflow- en sign-vlag gezet worden, dan kunnen we dit bereiken door voorafgaande aan een 16-bit ADC instructie een instructie op te nemen waarmee de carry vlag op nul gezet wordt.

13.3 Meer lus-mogelijkheden

De 16-bit load, increment, decrement en rekenkundige instructies die we tot nu toe bekeken hebben, kunnen gebruikt worden voor het werken met programmalussen, waarbij de lusindex (die bijhoudt hoeveel keer de instructies in de lus nog uitgevoerd moeten worden) ligt in het bereik van 0 tot en met 65535. Er zijn echter speciale 16-bit instructies voor de IX en IY indexregisters die we bij dergelijke lussen kunnen gebruiken.

Hieronder ziet u een eenvoudige lusstructuur waarbij van het IX indexregister gebruik wordt gemaakt.

```
        LD   IX,nn              ; of LD  IX,(LABEL)
LUS:    -
        -                       ; lusbody
        -
        INC  IX                 ; of DEC  IX
        JP   LUS
```

Eerst wordt het IX indexregister met een beginwaarde geladen, direct met een LD IX,nn instructie of indirect via LD IX,(LABEL). Aan het einde van de lusbody (de instructies in de lus) wordt het indexregister met INC IX één opgehoogd of met DEC IX één ver-

laagd. Vervolgens wordt gesprongen naar de eerste instructie uit de lusbody.

De lus zal afgebroken worden door een voorwaardelijke instructie ergens in de lusbody of als IX een bepaalde waarde heeft. Hierbij moeten we echter wel bedenken dat noch de INC IX, noch de DEC IX instructie één enkele vlag beïnvloedt. In plaats van het IX indexregister kunnen we ook van het IY indexregister gebruik maken.

Er bestaat een speciale ADD instructie waarmee we de indexregisters met een waarde ongelijk één kunnen ophogen of verlagen. We kunnen hiermee lussen maken waarbij de lusindex of lusteller niet steeds met 1 opgehoogd of verlaagd wordt maar met een waarde ongelijk 1. Deze instructies zijn:

`ADD IX,pp` en `ADD IY,rr`

waarin pp één van de registerparen BC, DE, IX of SP voorstelt
en rr één van de registerparen BC, DE, IY of SP.
De instructies zorgen ervoor dat de inhoud van het registerpaar pp en rr wordt opgeteld bij de inhoud van respectievelijk IX en IY.

In programma 13.2, waarmee de getallen van 1000 tot 0, steeds met 5 afnemend, worden afgedrukt, zien we een voorbeeld van het gebruik van de ADD IY,rr instructie.

```
; Programma 13.2 uitvoer bestaat uit 1000,995,900,...,5,0
;
        LD    DE,-5           ; DE bevat de afname
        LD    IY,1000         ; IY bevat het eerste getal
;
NEXNUM: CALL  IYOUT
        CALL  CRLF

        ADD   IY,DE           ; verlaag IY
                              ; check of IY nul is
        PUSH  IY              ; verplaats IY naar BC
        POP   BC
        LD    A,B
        CP    0               ; B nul?
        JP    NZ,NEXNUM       ; nee
        LD    A,C
        CP    0               ; C nul?
        JP    N2,NEXNUM       ; nee
        HALT
```

Het indexregister IY, dat als lusindex gebruikt wordt, wordt geladen met het getal 1000, dat als eerste wordt afgedrukt, Even daarvoor is de stapsgewijze afname (decrement) als waarde aan DE toegekend.

Bij elke doorgang van de lus wordt een carriage-return line-feed naar het display uitgevoerd, na het afdrukken van IY. (De IYOUT subroutine drukt de inhoud van het IY register als getal zonder teken (0-65535) af.)

Vervolgens wordt IY verlaagd met 5, dat wil zeggen bij IY wordt de inhoud van DE (-5) opgeteld. Nu zal getest moeten worden of IY nul geworden is. Dit is niet zo gemakkelijk als u wellicht zoudt vermoeden. We kunnen deze test op verschillende manieren uitvoeren; géén ervan verdient echter het predikaat 'netjes'! In programma 13.2 wordt de test uitgevoerd door het register IY te splitsen in twee registers, namelijk B en C, waarna vervolgens elk van deze twee registers op nul getest wordt.

U zult ontdekken dat het werken met de indexregisters nogal eens leidt tot 'onhandige' constructies, terwijl de andere registerparen, in het bijzonder het HL register, gemakkelijker in het gebruik zijn. Dit komt omdat voor deze 'andere' registers een groter aanbod aan instructies in de Z80 microprocessor aanwezig is.

13.4 Rekenen met waarden van n-bytes

Met een n-byte waarde bedoelen we een waarde van n×8 bits. Een 16-bit getal zouden we ook een 2-byte getal kunnen noemen; een 32-bit getal is een 4-byte getal. Hierbij is n steeds even (2,4,6..). We willen echter ook iets zeggen over oneven waarden van n, dus 24 bit, 40 bit, enz.

Voor dit 'n-byte' rekenen' staan ons twee instructies ter beschikking. Ze komen overeen met 16-bit, 32-bit, 48-bit,..., waarden. De twee instructies zijn:

`ADC A,s` en `SBC A,s`

waarin s een 8-bit waarde, een enkelvoudig register of een geheugenplaats aangewezen door HL, IX of IY voorstelt.
De linker instructie telt s èn de carry vlag op bij de accumulator, de rechter instructie trekt s èn de carry vlag van de inhoud van de accumulator af.

Het principe van het 'n-byte rekenen' is dat de twee minst significante bytes van twee bij elkaar op te tellen of van elkaar af te trekken getallen met behulp van een ADD (of SUB) instructie worden opgeteld (of afgetrokken) en dat de nog resterende paren bytes beginnende bij de volgende minst significante bytes en eindigend met de meest significante bytes, met de ADC (of SBC) instructie worden opgeteld (of afgetrokken).

Als we twee getallen, die elk uit een bepaald aantal bytes bestaan, bij elkaar moeten optellen, kan dat doorgaans door verschillende combinaties van 8-bit en 16-bit rekenkundige instructies. Stel de getallen zijn elk 6-bytes (48 bits) 'lang'. Nu is 48 gelijk aan 1×16 + 4×8 of 1×16 + 1×16 + 2×8 of 1×16 + 1×16 + 1×16 of 1×16 + 2×8 + 1×16 of 1×16 + 1×8 + 1×16 + 1×8; dus zijn er vijf combinaties van 8-bit en 16-bit instructies mogelijk (als we uitgaan van een 16-bit basisoptelling).
Willen we echter in het algemeen getallen bestaande uit een willekeurig aantal bytes bij elkaar kunnen optellen, dus ook getallen met een oneven aantal bytes, dan kunnen we dat doen door een bepaald aantal 8-bit optellingen uit te voeren.

Programma 13.3 bevat zo'n n-byte optelling.

```
; Programma 13.3 optellen van getallen van n-bytes
;
        -
        SCF             ; reset carry vlag
        CCF
;
NEXBYT: LD   A,(IX)
        ADC  A,(IY)     ; tel volgende paar bytes bij
                          elkaar op
        LD   (HL),A     ; sla resultaat op
        DEC  IX
        DEC  IY         ; wijs naar volgende meer signifi-
                          cante bytes
        DEC  HL
        DJNZ NEXBYT
                        , optelling klaar
```

In het begin verwijzen de registers IX, IY en HL naar de minst significante byte van respectievelijk het eerste getal, het tweede getal en de som, terwijl het B-register het aantal bytes bevat dat bij elkaar moet worden opgeteld. Voor het ingaan van de lus moet de carry vlag op nul gezet worden om ervoor te zorgen dat de eerste optelling (ADC A,(IY)) uitgevoerd wordt als een gewone ADD instructie.

Opgave 13.4
Welke aanpassingen vereist het programma 13.3 als we het tweede getal van het eerste zouden willen aftrekken?

13.5 Een programmeeropdracht

Schrijf een programma dat de getallen m tot en met n afdrukt, in stappen van k, dus m, m+k, m+2k, ..., n. Elk getal op een nieuwe regel.

De getallen m, n en k zijn hexadecimale getallen zonder teken, die via het toetsenbord worden ingevoerd.

De moeilijkheidsgraad van dit programma kan gevarieerd worden door te eisen dat de waarden van m, n en k aan één van de volgende voorwaarden voldoen:

8 bit	getallen	(8-bit-rekenen)
16-bit	getallen	(16-bit- of dubbel 8-bit-rekenen)
24-bit	getallen	(16-bit-instructies voor de minst significante 16 bits en 8-bit-instructies voor de meest significante 8 bits)
32-bit	getallen	(dubbele 16-bit-instructies)
n×16-bit	getallen	(meervoudig 16-bit-instructies)
of n×8-bit	getallen	(n-byte-instructies)

Bovendien zou het programma nog geschreven kunnen worden voor het accepteren van decimale in plaats van hexadecimale invoer.

14 Verplaatsen van en Zoeken in Geheugenblokken

De Z80 heeft acht bijzonder handige en efficiënte **blokinstructies**. Een blokinstructie voert een bewerking uit op een geheugenblok. Een geheugenblok is een aantal opeenvolgende geheugenplaatsen. Er zijn vier blokinstructies voor het **verplaatsen** (kopiëren) van een geheugenblok binnen het geheugen; de andere vier blokinstructies zijn bedoeld voor het **opzoeken** van een bepaalde geheugenplaats binnen een blok.

Alle blokinstructies zijn beschreven in tabel C.4 in Appendix C.

14.1 Blok-verplaats-instructies

Stel we moeten de inhoud van een geheugenblok, bestaande uit tien geheugenplaatsen, binnen het geheugen verplaatsen. Het blok begint op een geheugenplaats met label VAN en het moet verplaatst worden naar een ander blok dat begint met label NAAR. Programma 14.1 regelt dit voor ons:

```
; Programma 14.1 blok-overdracht - moeizame oplossing
;
        LD   HL,VAN          ; initieer pointers
        LD   DE,NAAR
        LD   B,10            ; en teller
;
NEXBYT: LD   A,(HL)          ; breng byte over
        LD   (DE),A
;
        INC  HL              ; verhoog pointers
        INC  DE
        DJNZ NEXBYT          ; en teller
        HALT
;
VAN:    DEFM 'ABCDEFGHJI'
NAAR:   DEFM '0000000000'
```

De registerparen HL en DE worden geïnitieerd met de adressen van respectievelijk de labels VAN en NAAR. Het B register wordt als teller gebruikt en krijgt een beginwaarde van 10. In de programmalus wordt steeds de inhoud van één geheugenplaats verplaatst van het VAN-blok naar het NAAR-blok; de verplaatsing geschiedt via de accumulator. Vervolgens worden de HL en DE registers beide met 1 opgehoogd, zodat ze de adressen van de volgende posities in beide blokken bevatten.

Na afloop van de lus en van de uitvoering van de HALT instructie zijn alle tien geheugenplaatsen uit het VAN-blok gekopieerd in het NAAR-blok. Er staan nu dus twee strings 'ABCDEFGHIJ' in het geheugen en de tien nullen, die eerst voor NAAR gedefinieerd waren, zijn overschreven.

Opgave 14.1
Stel we gebruiken het registerpaar BC als blokteller. Welke aanpassingen behoeft programma 14.1 om blokken met duizenden geheugenplaatsen te kunnen verplaatsen, in plaats van blokken met enkele tientallen geheugenplaatsen?

De Z80 bezit één instructie die hetzelfde doet als alle instructies in de lusbody van programma 14.1. Dit is de **LDIR** instructie (LoaD, Increment and Repeat). Vóórdat de instructie uitgevoerd wordt, moet het HL register het adres van de eerste geheugenplaats uit het VAN-blok bevatten; DE moet het adres van de eerste geheugenplaats uit het NAAR-blok bevatten, terwijl BC gelijk moet zijn aan het aantal plaatsen in beide blokken, dat wil zeggen het aantal te verplaatsen geheugenplaatsen. Programma 14.2 is een verbeterde versie van programma 14.1.

```
; Programma 14.2 blok-overdracht - handiger oplossing
;
        LD    HL,VAN          ; initieer pointers
        LD    DE,NAAR
        LD    BC,10           ; en teller
;
        LDIR                  ; breng het blok over
        HALT
;
VAN:    DEFM 'ABCDEFGHIJ'
NAAR:   DEFM '0000000000'
```

Programma 14,2 doet hetzelfde als programma 14.1. Het enige verschil is dat door het gebruik van BC als teller in plaats van B we blokken met maximaal 64K geheugenplaatsen kunnen verplaatsen in plaats van blokken van maximaal 255 bytes. De LDIR instructie

zorgt ervoor dat HL en DE met één opgehoogd worden, dat BC met één verminderd wordt en dat dit net zolang doorgaat totdat BC gelijk is aan nul.

De LDDR (LoaD, Decrement and Repeat) instructie is op één aspect na identiek aan de LDIR instructie. Zoals de mnemonic LDDR al aangeeft verlaagt de LDDR instructie de registers HL en DE met 1 in plaats van ze met 1 op te hogen.

Opgave 14.2
Herschrijf programma 14.2 door gebruik te maken van LDDR in plaats van LDIR.

De LDIR en LDDR instructies hebben elk nog een zusje (of broertje) in de vorm van respectievelijk de LDI en LDD instructie. Deze laatste twee instructies onderscheiden zich van LDIR en LDDR doordat ze niet automatisch de volgende verplaatsing uitvoeren (de R is immers uit de mnemonic!).

De LDI instructie (LoaD and Increment) verplaatst de inhoud van een geheugenplaats, verhoogt vervolgens de HL en DE registers met 1 en vermindert BC met 1. De LDD instructie verplaatst de inhoud van de door HL aangewezen geheugenplaats naar de door DE aangewezen nieuwe bestemming, verlaagt vervolgens de inhoud van HL en DE met 1 en vermindert BC met 1. LDI werkt een blok dus af van een laag adres naar een hoog adres en LDD werkt vanuit een hoog adres naar een laag adres.

Belangrijk is op te merken dat de toestand "BC=0" aangegeven wordt door de P/V vlag en niet door de zero vlag. De P/V vlag wordt op nul gezet (PO mnemonic) als BC nul is, anders op 1 (PE mnemonic).

Opgave 14.3
Herschrijf programma 14.2 door gebruik te maken van de LDI instructie in plaats van de LDIR instructie.

Als we van te voren precies weten hoeveel geheugenplaatsen er overgebracht moeten worden (hoe groot het blok is), kunnen we van de LDIR en LDDR instructies gebruik maken. Is dit echter niet bekend, dan moeten we van LDI of LDD gebruik maken en zullen we zelf in het programma moeten bepalen welke geheugenplaatsen voor verplaatsing in aanmerking komen. Een voorbeeld van een hierboven bedoelde situatie (LDI in plaats van LDIR) vindt u in opgave 14.4.

Opgave 14.4
Schrijf een stukje programma dat een blok met tekens binnen het geheugen verplaatst. Het programma moet een blok van maximaal 1000 tekens aankunnen. Afgezien van dit maximale aantal moet het 'verplaatsen' stoppen, indien het programma in het blok een geheugenplaats met 'nul' als inhoud tegenkomt. Deze 'stop-byte' behoeft zelf niet verplaatst te worden naar het nieuwe blok.

We moeten oppassen dat bij het verplaatsen van blokken het VAN-blok en het NAAR-blok elkaar niet overlappen. Bekijk de volgende reeks instructies eens:

```
LD   HL,START
LD   DE,START+100
LD   BC,500
LDIR
```

De eerste honderd bytes uit het VAN-blok (beginnend bij label START) worden gekopieerd in de eerste honderd bytes van het NAAR-blok (beginnend bij START+100), maar deze laatste honderd bytes vormen tevens de tweede honderd bytes uit het VAN-blok. Het VAN-blok loopt van START tot START+500, het NAAR-blok loopt van START+100 tot START+600. Er is dus een overlap van 400 bytes. Deze 400 bytes zullen overschreven worden voordat de CPU de gelegenheid krijgt ze te verplaatsen.

Opgave 14.5
Kunt u de vier instructies hierboven aanpassen zodat het 500-bytes-blok wel netjes verderop in het geheugen gekopieerd wordt?

14.2 Blok-opzoek-instructies

We hebben vier blok-opzoek-instructies om in een blok te zoeken naar een geheugenplaats met dezelfde inhoud als de accumulator. Alle vier instructies gaan er van uit dat de accumulator inderdaad de waarde bevat waarnaar gezocht moet worden. Bovendien gaan ze er van uit dat HL het adres van de eerste geheugenplaats uit het blok bevat en BC het aantal bytes in het blok.

Overeenkomstig de verplaatsinstructies zorgen twee van de vier blok-opzoekinstructies voor een zoekprocedure door het hele blok, terwijl de andere twee dat niet doen.

De twee 'automatische' blok-opzoekinstructies zijn CPIR (ComPare,

Increment and Repeat) en CPDR (ComPare, Decrement and Repeat). Programma 14.3 toont het gebruik van CPIR bij het zoeken naar de waarde nul als inhoud van één van de geheugenplaatsen binnen een blok.

```
; Programma 14.3 zoek in blok naar een 0-byte
;
        LD   HL,START        ; initieer pointer,
        LD   BC,10           ; teller
        LD   A,0             ; en accumulator
;
        CPIR                 ; zoek blok af naar de waarde nul
;
        LD   A,C             ; druk de waarde van de teller af
        ADD  A,30H
        CALL COUT
        HALT
;
START:  DEFB 1
        DEFB 2
        DEFB 3
        DEFB 0
        DEFB 4
        DEFB 5
        DEFB 0
        DEFB 6
        DEFB 0
        DEFB 7
```

Eerst wordt HL geladen met het adres van de eerste byte uit het blok, gelabeld met START. Het registerpaar BC wordt geladen met het aantal bytes in het STARTblok (10) en de accumulator krijgt de waarde (0) waarnaar we willen zoeken.

De CPIR instructie zoekt nu byte voor byte het blok af totdat òf de zoekwaarde (ook wel zoekargument genoemd) gevonden is òf het einde van het blok bereikt is. Dit laatste zal optreden als BC nul wordt. De inhoud van elke byte uit het blok wordt vergeleken met de inhoud van de accumulator. Zijn beide waarden gelijk dan wordt de zero vlag gehesen, dat wil zeggen 1 gemaakt; het HL register wordt 1 opgehoogd en het BC register 1 verlaagd.

Tenslotte beëindigt de CPIR instructie zijn bezigheden als òf de Z-vlag 1 is òf BC de waarde 0 heeft. Is dit niet het geval dan komt de volgende byte aan de beurt.

De instructies LDI en LDD gebruiken de P/V vlag om aan te geven of BC gelijk aan nul is. De blok-opzoekinstructies doen dit ook!

Opgave 14.6
Welk getal zal er door programma 14.3 met de daarbij gegeven blokinhoud afgedrukt worden? Wat zou er afgedrukt worden als geen der bytes in het blok de waarde nul zou bevatten? Mocht u er behoefte aan hebben, de CPIR instructie vindt u in tabel C.4 in Appendix C.

De CPDR instructie kunnen we gebruiken om een blok in omgekeerde volgorde te doorzoeken. In dit geval wordt HL eerst geladen met het adres van de laatste geheugenplaats uit het blok. Tijdens het uitvoeren van de CPDR instructie wordt het HL register steeds 1 verlaagd in plaats van opgehoogd.

De CPI (ComPare Increment) en CPD (ComPare Decrement) instructies zoeken niet uit eigen beweging een heel blok door. Daar zijn extra instructies voor nodig. Ze zetten echter wel de P/V vlag als BC ongelijk is aan nul en de Z-vlag indien de inhoud van de accumulator gelijk is aan de inhoud van een byte in het blok. Dus $P/V = 1 \Rightarrow BC \neq 0$ en $P/V = 0 \Rightarrow BC = 0$.

De CPI en CPD instructies gebruiken we in plaats van de CPIR en CPDR instructies als we het doorzoeken van een blok willen kunnen onderbreken of als er gezocht moet worden naar meer dan één geheugenplaats met de gezochte inhoud.

Opgave 14.7
Hoe moet programma 14.3 veranderd worden opdat bij elke geheugenplaats binnen het blok met inhoud nul de waarde van de teller wordt afgedrukt?

Soms is het wenselijk om na afloop van een blokinstructie LDIR, LDDR, CPIR of CPDR te weten wat de inhoud is van de registers HL en DE.

Na afloop van de LDIR instructie zullen HL en DE wijzen naar de geheugenplaatsen direct volgend op die welke het einde van respectievelijk het VAN-blok en het NAAR-blok aangeven.

Na afloop van de LDDR instructie bevatten HL en DE de adressen van de geheugenplaatsen die vlak voor het begin van respectievelijk het VAN-blok en het NAAR-blok liggen.

Na afloop van de CPIR instructie verwijst HL naar de geheugenplaats direct volgend op het einde van het zoek-blok.

Na afloop van de CPDR instructie zal HL verwijzen naar de geheugenplaats vlak vóór het begin van het zoek-blok.

14.3 Een programmeeropdracht

Uw programma moet een onderhoudsprogramma worden voor het onderhouden van een 'in-memory-bestand' van maximaal negen records. Elk record bevat 20 tekens.

De ruimte voor het bestand moet aan het einde van uw programma als volgt gereserveerd worden:

```
FILE:  DEFS 180
```

Tot zover de programmaschets. De inhoud van een record (de 20 tekens) is bij deze opgave onbelangrijk. Het programma identificeert een record met een volgnummer, de positie van een record in het bestand. De recordnummers zijn dus de nummers 1 tot en met 9.

Een gebruiker van uw programma moet het volgende onderhoud kunnen plegen, dat wil zeggen uw programma moet de volgende invoer kunnen verwerken:

D n	Het verwijderen (Delete) van het record met nummer n. Alle overblijvende records moeten weer aaneengeschoven worden.
I n record van 20 tekens	Toevoegen (Insert) van een record met de opgegeven 20 tekens als record met nummer n. Eventuele records met een hoger nummer moeten dus eerst worden opgeschoven.
R n record van 20 tekens	Vervangen (Replace) van een bestaand record met nummer n door het nieuw opgegeven record.
L	Drukt (List) het hele bestand op het beeldscherm af. Eerst het recordnummer, dan de inhoud.
F zoekstring	Zoekt (Find) het eerste record in het bestand dat de opgegeven 'zoekstring' als deelstring bevat en drukt dit record af. De opgegeven zoekstring bevat 1, 2 of 3 tekens.

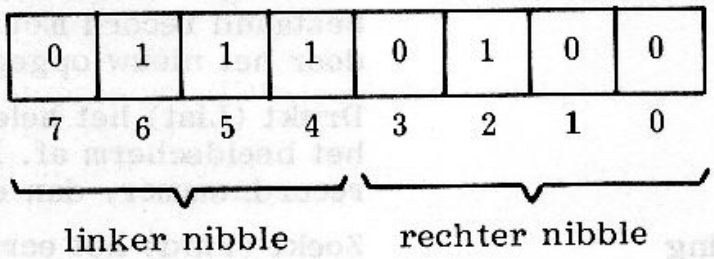
15 Decimaal Rekenen

Tot nu toe hebben we alleen de binaire voorstelling van getallen in de Z80 microprocessor bekeken. Rekenen deden we tot nu toe met binaire getallen zonder teken (0-255) en met twee-complement getallen (-128 tot +127). We kunnen echter getallen ook anders dan binair in de Z80 microprocessor opslaan en wel in de zogeheten 'Binary Coded Decimal form', kortweg BCD genoemd.

15.1 De BCD code

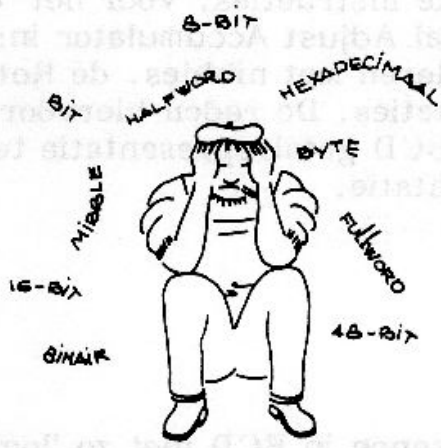
Een decimaal cijfer (0,1,2,...,9) in BCD code betekent dat we een 4-cijferig binair getal gebruiken om het decimale cijfer te coderen. Zo wordt 9 in BCD gecodeerd als 1001B.

Zo'n 4-bits BCD code noemen we een **nibble**. Een nibble is dus een groep van vier bits of een halve byte. In één byte gaan 2 nibbles dus 2 BCD cijfers. Een voorbeeld:



De linker nibble, bits 7 tot en met 4, bevat de BCD code voor het getal 7 en de rechter nibble, bits 3 tot en met 0, bevat het BCD cijfer 4. De byte als zodanig bevat het BCD getal 74.

Opgave 15.1
Geef de binaire inhoud van een byte waarin zich de BCD code voor het getal 57 bevindt.



Een nibble kan een BCD-cijfer van 0 tot en met 9 bevatten, een decimaal cijfer dus, terwijl binair gezien een nibble een getal (zonder teken) tussen 0 en 15 kan bevatten. Conclusie is dan ook dat de BCD-code nogal slordig met de geheugenruimte omspringt in vergelijking met de binaire codering. Dit is nog eens weergegeven in onderstaand tabelletje.

BCD-cijfer	BCD-code	BCD-cijfer	BCD-code
0	0000	8	1000
1	0001	9	1001
2	0010	10	ongebruikt
3	0011	11	
4	0100	12	
5	0101	13	
6	0110	14	
7	0111	15	

Opgave 15.2
Vergelijk het bereik van een byte met BCD-representatie en het bereik van een byte met binaire representatie voor het coderen van getallen (zonder teken), met andere woorden: wat is in één byte de kleinste waarde en de grootste waarde bij het coderen in BCD en binair?

Gewoonlijk worden getallen als decimale getallen via het toetsenbord ingevoerd en op het beeldscherm afgedrukt. Willen we echter met deze getallen kunnen rekenen, dan moeten ze eerst geconverteerd worden tot binaire getallen en bij uitvoer geconverteerd worden van binair naar decimaal. Zou de computer echter beschikken over rekenkundige instructies waarin met de decimale representatie gewerkt zou kunnen worden, dan zouden deze conversies achterwege kunnen blijven.

Welnu de Z80 bezit dergelijke instructies. Voor het 'decimale rekenen' bezit de Z80 een Decimal Adjust Accumulator instructie en instructies voor het manipuleren met nibbles, de Rotate Left Digit en Rotate Right Digit instructies. De reden hiervoor is dat het soms gewoon handiger is met de BCD getalrepresentatie te werken dan met de binaire getalrepresentatie.

15.2 Rekenen met BCD

U zult begrijpen dat het rekenen in BCD niet zo 'logisch' is als binair rekenen op een computer die zelf binair georiënteerd is. De gewone rekenkundige instructies kunnen we niet gebruiken omdat het rekenen in BCD gewoon anders gaat dan binair rekenen. Zo zal de som van BCD 28 en BCD 39 een fout resultaat BCD 61 opleveren als we de standaard binaire ADD instructie hiervoor zouden gebruiken, kijk maar:

```
  00101000      BCD 28
+ 00111001      BCD 39
  --------
  01100001      BCD 61 - fout!
```

De fout ontstaat doordat bij BCD optelling een carry van de rechter nibble naar de linker nibble nodig is, indien de som van de beide rechter nibbles groter is dan 9. Bovendien produceert een binaire optelling soms nibbles met waarden tussen 1010B en 1111B, waarvoor géén BCD-representatie bestaat!

Het is wel mogelijk na een binaire ADD instructie op twee BCD-getallen het resultaat te corrigeren, zodat een goed BCD-resultaat ontstaat. Deze correctieprocedure gaat als volgt. Als de nibble een waarde uit de reeks 1010B tot en met 1111B mocht bevatten of als uit het meest significante bit van een nibble een carry ontstaat, moeten we 0110B bij de nibble optellen. In alle andere gevallen hoeven we niets te doen. De hiernavolgende voorbeelden laten dit nog eens zien:

```
    0110        BCD 6
+   0111        BCD 7
    ----
    1101        -fout; resultaat is geen BCD-cijfer
+   0110
    ----
00010011        BCD 13 - dit is goed
```

```
    1000        BCD 8
  + 1001        BCD 9
  ------
    00010001    BCD 11 - fout
  +   0110
  ------
    00010111    BCD 17 - goed
  ------
```

Opgave 15.3
Laat de correctie zien op BCD 17 + BCD 69 (zoals hierboven is gedaan).

Een carry vanuit de linker nibble van een 2-cijferig BCD-getal betekent overflow, dat wil zeggen een waarde groter dan 99.

Bij het aftrekken van twee BCD-getallen moeten we 0110B van het resultaat aftrekken, indien een borrow nodig was voor de rechter nibble, of indien de rechter nibble uit het resultaat een waarde heeft uit de reeks 1010B tot en met 1111B.

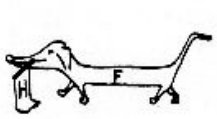
Opgave 15.4
Laat zien hoe de aftrekking BCD 82 - BCD 56 verloopt.

Een borrow, optredend in de linker nibble van een 2-cijferige BCD-aftrekking zou overflow moeten betekenen.

15.3 De DAA instructie

Om de mogelijkheid tot decimaal rekenen te creëren moet een computer of uitgerust zijn met een aparte serie decimale rekeninstructies of er moet een mogelijkheid zijn het resultaat van een binaire rekenkundige instructie om te zetten in een decimaal resultaat, zoals dat in de vorige paragraaf gedaan is. Welnu, de Z80 ontwerpers hebben voor het laatste gekozen en wel in de vorm van de DAA (Decimal Adjust Accumulator) instructie.

Bij het uitvoeren van een rekenkundige instructie (ADD, SUB, NEG, enz.) worden de tot nu toe nog niet besproken vlaggen uit het vlagregister, de Halfcarry en Subtract vlaggen, op de hieronder beschreven manier beïnvloed.



De Halfcarry vlag wordt door een optelinstructie op 1 gezet indien deze instructie een carry vanuit de rechter nibble van een register veroorzaakt; zo niet dan wordt de vlag 0. De Halfcarry vlag wordt ook 1 als bij een aftrekopdracht voor het aftrekken van de rechter nibbles een borrow voor een rechter nibble nodig is. Is er geen borrow nodig dan wordt de vlag 0.

De Subtract vlag wordt 1 bij een optelinstructie en 0 bij een aftrekinstructie.

Alhoewel de halfcarry en subtract vlaggen alleen door de DAA instructie gebruikt worden, worden ze toch door elke rekenkundige instructie gezet. Geen van beide vlaggen kan door ons (de programmeurs) gebruikt worden en wel omdat er geen instructies zijn om de status van deze vlaggen te testen of om zelf deze vlaggen te zetten!

Afhankelijk van de status van deze twee vlaggen past de DAA instructie zonodig op de inhoud van de accumulator een bepaalde correctie toe, zodat de inhoud van de accumulator beschouwd kan worden als zijnde het resultaat van een rekenkundige BCD-bewerking. Zo zullen de volgende vier instructies

```
LD  A,43H
LD  B,28H
ADD A,B
DAA
```

de accumulator en het B register respectievelijk laden met BCD 43 en BCD 28; deze waarden optellen (met een binaire ADD instructie) en het resultaat door middel van de DAA instructie corrigeren zodat de accumulator de BCD som bevat. Ziet u hoe handig hexadecimale getallen zijn om BCD-constanten mee aan te geven?
(Een hexadecimaal cijfer wordt immers ook met 4 bits gecodeerd.)

Opgave 15.5
Wat is de inhoud van de accumulator, als hexadecimale waarde, na het uitvoeren van elk van de vier bovenstaande instructies?

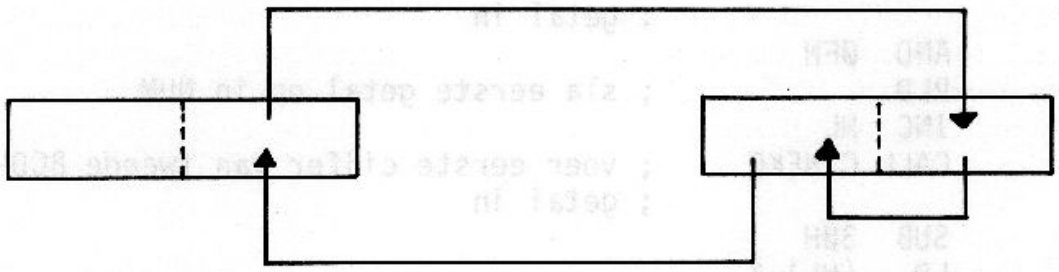
De DAA instructie kan gebruikt worden om 'BCD te rekenen' na de binaire rekenkundige instructies ADD A, SUB, INC A, DEC A, CP, NEG, ADC A, SBC en na de vier blok-opzoekinstructies. Denk er om: de DAA instructie werkt alleen op de accumulator.

Voor de programmeur zijn de twee belangrijkste vlaggen, die door de DAA gezet worden, de carry vlag en de zero vlag. De carry vlag geeft in dit geval BCD-overflow aan, terwijl de zero vlag een BCD 0 aangeeft. We zullen straks zien dat we de status van de carry vlag ook kunnen gebruiken bij het 'n-byte BCD rekenen'.

15.4 De nibble roteerinstructies

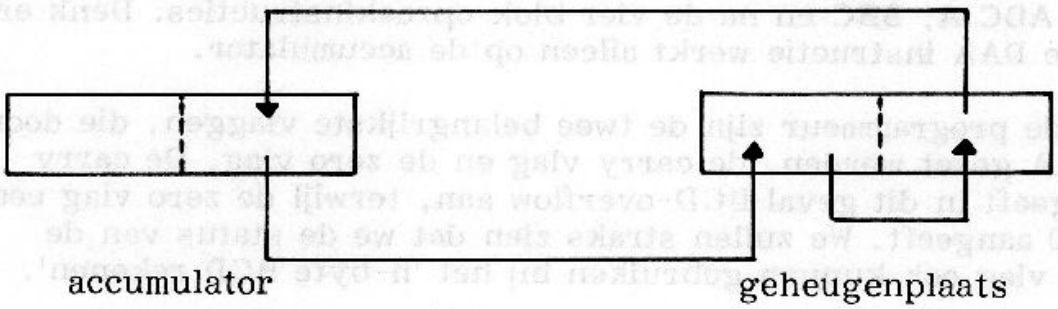
Voor het roteren van nibbles, BCD-cijfers dus, staan ons twee instructies ter beschikking: een rechter en een linker roteer-instructie.

Bij een rotatie zijn betrokken de rechter nibble van de accumulator en de twee nibbles van een geheugenplaats aangewezen door het HL registerpaar. Hieronder zien we werking van de Rotate Left Digit instructie, kortweg RLD genoemd:



De rechter nibble van de accumulator verschuift naar de rechter nibble van de geheugenplaats; de rechter nibble van de geheugen-plaats verschuift naar de daarnaast liggende linker nibble, terwijl tenslotte deze linker nibble van de geheugenplaats verschuift naar de rechter nibble van de accumulator. Met andere woorden: de linker nibble van de accumulator doet niet mee in de rotatie.

De **Rotate Right Digit** instructie (RRD) werkt als volgt:



De RLD en RRD instructies zijn zeer nuttig bij het rekenen met BCD-getallen. Programma 15.1, waarin twee 2-cijferige BCD-getallen worden ingelezen en opgeteld en waarin de som wordt afgedrukt, toont het nut van beide roteerinstructies.

```
; Programma 15.1 optellen van twee 2-cijferige BCD-getallen
;
        LD   HL,NUM
        CALL CINEKO      ; voer eerste cijfer van eerste BCD-
                         ; getal in
        AND  ØFH
        LD   (HL),A
        CALL CINEKO      ; voer tweede cijfer van eerste BCD-
                         ; getal in
        AND  ØFH
        RLD              ; sla eerste getal op in NUM
        INC  HL
        CALL CINEKO      ; voer eerste cijfer van tweede BCD-
                         ; getal in
        SUB  3ØH
        LD   (HL),A
        CALL CINEKO      ; voer tweede cijfer van tweede BCD-
                         ; getal in
        SUB  3ØH
        RLD              ; sla tweede getal op in NUM+1
        LD   HL,NUM
        LD   A,(HL)
        INC  HL
        ADD  A,(HL)      ; tel de twee getallen op
        DAA              ; decimal adjust voor BCD
        LD   HL,SOM      ; redt het resultaat
        LD   (HL),A
        LD   A,Ø         ; druk resultaat af van
        RLD
        ADD  A,3ØH
        CALL COUT        ; het eerste BCD-getal en van
```

```
        LD   A,Ø
        RLD
        ADD  A,3ØH
        CALL COUT       ; het tweede BCD-getal
        HALT
;
NUM:    DEFS 2
SOM:    DEFS 1
```

Als we het programma even doorlopen zien we dat eerst de twee cijfers uit het eerste BCD getal worden ingelezen en dat die vervolgens met de RLD instructie in NUM worden opgeslagen. Direct daarna worden de twee cijfers uit het tweede BCD getal ingelezen en met een volgende RLD instructie opgeslagen in NUM+1. Het HL register moet de adressen van respectievelijk NUM en NUM+1 bevatten omdat de RLD instructie hierop rekent!

Vervolgens worden beide getallen opgeteld en direct daarna wordt het resultaat 'decimaal gecorrigeerd' (decimal adjust) door de DAA instructie. Het resultaat wordt cijfer voor cijfer opgeslagen voordat het wordt afgedrukt. De RLD instructie zorgt ervoor dat de nibbles uit SOM één voor één naar de accumulator worden geschoven, van waaruit ze worden afgedrukt.

We merken op dat voor het uitvoeren van elk van de twee laatste RLD instructies de accumulator nul gemaakt wordt om ervoor te zorgen dat de linker nibble van de accumulator nul is. Alhoewel de linker nibble niet 'meedoet' in de decimale roteerinstructies is het toch goed ervoor te zorgen dat de inhoud van de linker nibble van de accumulator een door ons gewenste waarde heeft!

15.5 Een programmeeropdracht

De opdracht is een programma te maken voor het inlezen, optellen, aftrekken en afdrukken van meercijferige positieve en negatieve BCD-getallen.

Zo'n BCD-getal wordt in het geheugen als volgt gespecificeerd: de eerste byte van het getal geeft het teken van het getal aan (bit 7 is 0 bij een positief getal en 1 bij een negatief getal) en het aantal cijfers waaruit het BCD-getal bestaat (bits 6 t/m 0); de daarop volgende bytes bevatten de BCD-cijfers uit het getal en wel steeds twee per byte.

Schrijf een subroutine INBCD waarmee een positief of negatief BCD-

getal ingelezen kan worden (geen teken betekent positief) vanaf het toetsenbord. Het ingelezen getal wordt volgens de hierboven omschreven wijze door de subroutine in het geheugen opgeslagen. Bij aanroep van de subroutine wijst het HL register naar de eerste van het groepje geheugenplaatsen waar dit getal zal worden opgeslagen.

Schrijf ook een subroutine UITBCD voor het afdrukken van een BCD-getal (onderdruk het + teken) op het beeldscherm. Bij aanroep van de routine bevat HL het adres van de eerste geheugenplaats van het groepje bytes waar het af te drukken getal ligt opgeslagen.

Schrijf vervolgens een routine BCDADD voor het optellen van twee 'even lange' BCD-getallen (positief of negatief). Bij aanroep van deze subroutine verwijst IX naar het éne en IY naar het andere getal, terwijl HL het adres bevat vanwaaraf het resultaat van de optelling zal worden opgeslagen. Bij terugkeer uit de subroutine bevat de accumulator de waarde 1 indien er overflow is opgetreden en 0 als dit niet het geval is.

Schrijf dan een routine BCDSUB waarmee het tweede getal (IY) afgetrokken kan worden van het eerste getal (IX). Verder dezelfde eisen als bij BCDADD.

Gebruik nu alle subroutines om een programma te maken waarmee herhaald twee BCD-getallen (positief of negatief), gescheiden door een + teken of een - teken en afgesloten met een = teken kunnen worden ingevoerd, waarbij het resultaat achter het = teken wordt afgedrukt. Het programma moet BCD getallen van maximaal 20 cijfers aan kunnen. Hieronder een voorbeeld van de werking van dit programma:

```
  444 + 111   =   555
89540 - 76541 = 12999
```

16 Diverse Instructies

We hebben een aantal instructies nog niet besproken. Het betreft instructies die erg weinig gebruikt worden en instructies die buiten het bestek van dit boek vallen. Toch zullen we ze, voor alle volledigheid, in vogelvlucht de revue laten passeren.

16.1 De NOP instructie

De NOP instructie voert NIETS uit, doet NIETS, maar wordt, paradoxaal genoeg, nog wel eens gebruikt. We kunnen deze instructie bijvoorbeeld gebruiken om in de uitvoering van een programma een PAUZE in te lassen:

```
PAUZE:  NOP
        DJNZ PAUZE
```

Als we bovenstaande instructies inlassen in een programma zal dat een PAUZE van (N-1) x (4+13) + 8 kloktijden tot gevolg hebben, aangenomen dat N de waarde in het B register is vóór het uitvoeren van de PAUZE-lus. Het uitvoeren van de NOP instructie kost 14 kloktijden, het uitvoeren van de DJNZ instructie 13 als B ongelijk nul is en 8 kloktijden als B wel nul is.

16.2 Alternatieve registergroep

Zoals aan het begin van dit boek is verteld, bezit de Z80 een alternatief stel registers, aangeduid met A', F', B', C', D', E', H' en L' (ook wel auxiliary registers genoemd). Deze groep registers kan op exact dezelfde manier gebruikt worden als de 'gewone' registers, echter niet tegelijkertijd.

We kunnen de gewone groep registers uitwisselen met de alterna-

tieve groep met de instructie

```
EXX     ; EXchange standard en auXiliary registers
```

De instructies die na een EXX instructie in een programma opgenomen worden zullen nu werken met de nieuwe groep registers. Een tweede EXX instructie zorgt ervoor dat de standaard registers weer actief worden.

De meeste programma's hebben echter genoeg aan de groep standaard registers. Is er toch behoefte aan meer, dan komt het zelden voor dat alle acht registers uitgewisseld moeten worden, hetgeen wel het effect is van de EXX instructie. Wat wel vaker voorkomt is de behoefte aan een tweede accumulator. Om aan deze behoefte tegemoet te komen bestaat de instructie

```
EX  AF,AF'     ; Exchange AF en AF'
```

waarmee de accumulator en het vlagregister uitgewisseld worden met de op dat moment alternatieve accumulator en vlagregister.

16.3 In- en Uitvoer instructies

Er zijn bij elkaar twaalf in- en uitvoerinstructies. U kunt ze allemaal vinden in tabel C.12 in Appendix C. Twee hiervan hebben we in het boek gebruikt, namelijk IN A,(n) en OUT (n),A voor het invoeren en uitvoeren van gegevens in respectievelijk uit de accumulator.

We kunnen gegevens invoeren en uitvoeren vanuit elk willekeurig register door gebruik te maken van de instructies IN r,(C) en OUT (C),r. Hierin specificeert de inhoud van register C de gewenste in- of uitvoerpoort.

De andere in- en uitvoerinstructies werken met gegevensblokken. Met deze instructies en hun varianten kunnen we een heel blok, byte voor byte in- of uitvoeren. Hierbij doet het B register dienst als blokteller en niet het BC register, zoals dat bij de blok-opzoek instructies het geval is.

Deze blok-in- en uitvoerinstructies lijken op het eerste gezicht erg handig. We moeten echter wel bedenken dat de blok in- en uitvoerinstructies, die automatisch een heel blok in- of uitvoeren (in de mnemonic zit de R van Repeat!), alleen gebruikt kunnen worden voor in- en uitvoerapparatuur die met dezelfde hoge snelheid werkt als waarmee instructies worden uitgevoerd; het toetsenbord en het beeldscherm behoren echter niet tot deze categorie randapparatuur.

16.4 Interrupt instructies

Met een interrupt bedoelen we een signaal van 'buiten' dat het uitvoeren van de instructies door de CPU onderbreekt. Zo'n signaal kan bijvoorbeeld een stuk randapparatuur zijn dat bediend wil worden.

Vijf van de instructies die betrekking hebben op het Z80 interruptsysteem kunt u vinden in tabel C.8 van Appendix C. De Z80 microprocessor is voorzien van drie interrupt-mechanismen. Met OM 0, IM 1 en IM 2 kunnen we een bepaald interrupt-mechanisme selecteren. Met de DI en EI instructies (Disable Interrupt en Enable Interrupt) kunnen we vanuit een programma regelen dat interrupts niet meer 'bediend' worden (DI) of dat er weer interrupts zijn toegestaan (EI).

Interrupts worden afgehandeld door zogeheten Interrupt Service Routines. Dit zijn stukjes programma die elk op een bepaalde manier een bepaald soort interrupt afhandelen. De laatste instructie in zo'n Interrupt routine is een RETI (RETurn from Interrupt) of een RETN (RETurn from Non-maskable interrupt) instructie. Na het uitvoeren van deze instructies gaat de CPU verder met het uitvoeren van die instructies waarvan de uitvoering door de interrupt onderbroken werd. De RETI en RETN instructies zijn samen met de RST (ReSTart) instructie, die het interrupt-mechanisme 0 bedient, opgenomen in Tabel 11 van Appendix C. De hele problematiek rond het afhandelen van interrupts valt echter buiten het bestek van dit boek.

Appendix A
Binaire en Hexadecimale Getalstelsel

Om te kunnen begrijpen hoe gegevens in computergeheugens worden opgeslagen moeten we iets afweten van binaire en hexadecimale getallen. In het dagelijkse leven zijn we gewend te werken met decimale getallen. Computers zijn gewend met binaire getallen te werken.
Het gebruik van hexadecimale getallen is een compromis tussen beide bovengenoemde getalstelsels. Hexadecimale getallen zijn voor ons iets gemakkelijker te hanteren dan binaire getallen, terwijl hexadecimale getallen tevens geschikt zijn om binaire getallen 'compact' weer te geven.

Kort samengevat: decimaal betekent rekenen met machten van 10; binair betekent rekenen met machten van 2; hexadecimaal betekent rekenen met machten van 16. Nog korter: decimaal = tientallig stelsel; binair = tweetallig stelsel en hexadecimaal = zestientallig stelsel.

A.1 Binaire en hexadecimale getallen

Elk decimaal getal kunnen we schrijven als optelling van een aantal machten van 10. Zo kunnen we 453 schrijven als:

$$453 = 4\times10^2 + 5\times10^1 + 3\times10^0$$

Elk hexadecimaal getal kunnen we weergeven als een optelling van een aantal machten van zestien. Zo kunnen we 974H schrijven als:

$$974H = 9\times16^2 + 7\times16^1 + 4\times16^0$$

Zo kunnen we een binair getal schrijven als optelling van een aantal machten van 2:

$$101B = 1\times2^2 + 0\times2^1 + 1\times2^0$$

Als we in het vervolg willen aangeven dat een getal een 'hexadecimaal' getal is zullen we een H achter het getal zetten. Achter een binair getal zetten we een B. Achter een decimaal getal zetten we niets!

Het getal 10, het getal 2 en het getal 16 vormen het grondtal (of radix) van respectievelijk het tientallig, tweetallig en zestientallig stelsel. Decimale getallen hebben 10 als grondtal; binaire getallen hebben 2 als grondtal en hexadecimale getallen hebben 16 als grondtal.
Theoretisch kunnen we elk getal als grondtal voor een talstelsel kiezen. Bij computers is het gebruikelijk te werken met de grondtallen twee en zestien.

Opgave A.1
Wat zijn de decimale equivalenten van de getallen 974H en 101B?

U weet dat het tientallig stelsel de cijfers 0,1,2,3,...,9 omvat. Het grootste cijfer is één lager dan het grondtal.

Opgave A.2
Welke cijfers (digits) worden er in het binaire talstelsel gebruikt?

Het decimale stelsel kent tien verschillende cijfers (0 t/m 9), het binaire talstelsel twee (0,1), dus het hexadecimale stelsel ..., juist zestien! Als eerste tien cijfers kunnen we gewoon de cijfers 0 tot en met 9 gebruiken, maar dan? Cijfers geven we doorgaans aan met één teken; met meer tekens worden het immers getallen! We kiezen voor de overige zes cijfers in het zestientallig stelsel de letters A,B,C,D,E en F. Het hexadecimale cijfer A is een decimale 10 en het hexadecimale cijfer F is een decimale 15.

In de tabel op p.139 zien we de decimale getallen 0 tot en met 15 met daarachter de daarmee corresponderende waarden in zowel het hexadecimale als binaire talstelsel.

Maak u zelf zo vertrouwd met de inhoud van deze tabel dat u zonder nadenken kunt zeggen: 0111, dat is 7!

Opgave A.3
Wat is de decimale waarde voor het getal E8A5H?

decimaal	hexadecimaal	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

A.2 Binair en hexadecimaal rekenen

Optellen en aftrekken kan in elk talstelsel. De techniek is hetzelfde, alleen de grondtallen zijn anders. Bij het optellen van twee decimale cijfers ontstaat een carry (wij zeggen "1 onthouden"), als de som van de cijfers groter is dan 9. Bij het optellen van twee hexadecimale cijfers ontstaat zo'n carry indien de som groter is dan F (decimaal 15).

Hieronder staan enkele voorbeelden van hexadecimaal en binair rekenen.

```
  3A7FH          10110110B
+ 10BBH        - 01011010B
  ------         --------
  4B3AH          01011100B
```

Opgave A.4
Voer de volgende berekeningen uit:

```
  C7BAH          01101101B
- 9FF8H        + 01011110B
  ------         --------
  .....          ........
```

A.3 Van decimaal naar hexadecimaal

Om van een decimaal getal een hexadecimaal getal te maken gaan we als volgt te werk. We delen het decimale getal door zestien. We noteren de rest als hexadecimaal cijfer (kleiner dan 16) en gaan met de uitkomst verder. Deze uitkomst delen we weer door 16, noteren de rest als hexadecimaal cijfer (kleiner dan 16) en gaan met de uitkomst verder, enzovoorts. We gaan door met noteren van de rest tot de uitkomst van de deling nul is. Lezen we nu alle door ons genoteerde 'hexadecimale resten' in omgekeerde volgorde, dan hebben we het hexadecimale getal. Zo wordt het decimale getal: 745

```
16/745\46        16/46\2        16/2 \ 0
  64              32              0
 ---             ---             ---
 105              E rest         2 rest
  96
 ---
   9 rest
```

in het hexadecimale talstelsel het getal 2E9.

Opgave A.5
Zet het decimale getal 1582 om in een hexadecimaal getal.

A.4 Van hexadecimaal naar decimaal

Om een hexadecimaal getal om te zetten in een decimaal getal behoeven we alleen het hexadecimale getal te schrijven als een som van machten van 16. Zo wordt de waarde 3AB2H omgezet in

$$\begin{aligned} 3AB2H &= 3\times16^3 + 10\times16^2 + 11\times16^1 + 2\times16^0 \\ &= 3\times4096 + 10\times256 + 11\times16 + 2 \\ &= 15026 \end{aligned}$$

het decimale getal 15026.

Een snellere manier om een hexadecimaal getal om te zetten in een decimaal getal (en andersom) is het gebruiken van een conversietabel. Die moet dan wel voorhanden zijn!

Opgave A.6
Gebruik de conversietabel in Appendix B om FBH en A3B2H om te zetten in decimale getallen en om 142 en 9467 om te zetten in hexadecimale getallen.
U kunt ook de tabellen op de inlegkaart gebruiken!

A.5 Binaire-hexadecimale conversie

Bij de conversie van binair naar hexadecimaal en omgekeerd maken we gebruik van het feit dat een hexadecimaal getal vervangen kan worden door een '4-cijferig' binair getal (en omgekeerd).

Om bijvoorbeeld het hexadecimale getal 6B naar binair te converteren vervangen we simpelweg de afzonderlijke hexadecimale cijfers uit het getal door hun binaire equivalenten. Dus:

6BH = 0110 1011

Laten we nu de eventuele nullen aan het begin weg en voegen we de twee binaire getallen samen, dan krijgen we

6BH = 1101011B

In het omgekeerde geval splitsen we een binair getal van rechts af in groepjes van 4 binaire cijfers; vervolgens vervangen we elk groepje door het hexadecimale equivalent en klaar is kees.
Voorbeeld: het binaire getal 1111100111 splitsen we als volgt:

11 1110 0111

Elke groep vervangen we door hun hexadecimale equivalent (de meest linkse groep vullen we met nullen aan tot we een groepje van 4 krijgen):

```
0011  1110  0111
  ↓     ↓     ↓
  3     E     7
```

Dus 1111100111B is equivalent met 3E7H.

Opgave A.7
Converteer 9AB3H naar een binair getal en 110011101111B tot een hexadecimaal getal.

A.6 Decimale-binaire conversie

Deze conversie kunnen we op dezelfde wijze uitvoeren als de in A.3 en A.4 beschreven conversies, behalve dat daar waar we 16 gebruiken we nu 2 moeten gebruiken.

Deze methoden zijn echter voor conversies van decimaal naar binair en omgekeerd nogal arbeidsintensief. Daarom raden we aan bij dit soort conversies:

maak gebruik van een conversietabel

of

gebruik de hexadecimale notatie als een tussenstap.
Converteer bijvoorbeeld eerst een decimaal getal
naar een hexadecimaal getal en converteer vervolgens
dit hexadecimale getal naar een binair getal.

Opgave A.8
Converteer 1290 naar een binair getal en 101110111101B naar een decimaal getal.

A.7 Bytes

De eenheid die als basis dient voor het opslaan van gegevens in de Z80 microprocessor is een byte. Een byte bestaat uit acht binaire cijfers (BInary DigiTs), kortweg bits genoemd. Een byte bestaat tevens uit twee hexadecimale cijfers. De inhoud van een byte (de acht nullen en/of enen) representeert

een teken
een getal (zonder teken) of
een getal met teken

De representatie van tekens wordt besproken in hoofdstuk 3.

Als de inhoud van een byte een getal voorstelt dan is dat de waarde van het binaire getal dat door de nullen en énen in de byte wordt gevormd. Zo kan de byte met inhoud 01100110 het getal 01100110B weergeven (66H of 102). In een byte kunnen we binaire getallen van 00000000B tot en met 11111111B opslaan (dus van 00H tot en met FFH of gewoon van 0 tot en met 255).

Soms is de betekenis van een byte echter een andere dan 'de waarde van een binair getal'. We moeten er dus wel op letten wat in een bepaalde situatie de precieze betekenis is van de inhoud van een byte.

Opgave A.9
Wat is de maximale waarde van de getallen die we met twee bytes (16 bits) kunnen weergeven?

A.8 Getallen met teken (twee-complement)

Getallen die zowel positief als negatief kunnen zijn, getallen met teken dus, worden in computers dikwijls als zogeheten twee-complement getallen opgeslagen. De Z80 gebruikt ook deze methode, alhoewel er ook andere methoden zijn voor het weergeven van negatieve getallen in een computer.

De twee-complement-methode is een methode die werkt met het weergeven van een getal in een vast aantal binaire digits. Zo bestaan in de Z80 twee-complement-getallen altijd uit acht binaire digits (bits).

In het twee-complement-systeem wordt een negatief getal weergegeven door het twee-complement te nemen van de overeenkomstige positieve waarde; dit 'twee-complement' nemen betekent dat in de binaire representatie van het positieve getal alle nullen in énen en alle énen in nullen veranderd worden en dat vervolgens één bij het resultaat wordt opgeteld. Een voorbeeld:

```
                +5      is      00000101B

dus             -5      is      11111010B
                                    + 1
                                ---------
                                11111011B
```

Het getal -5 wordt dus als 1111011B in de computer opgeslagen. De twee-complement-waarde van een getal kunnen we ook verkrijgen door de positieve waarde van 2 af te trekken.

Opgave A.10
Bereken de twee-complement-waarden voor -1, -2 en -126 en de decimale equivalenten van 10000000B en 10000001B.

Bij het optellen van twee twee-complement-getallen worden de twee getallen bit voor bit (van rechts af) opgeteld. Hierbij wordt een eventuele carry van het meest significante bit (het meest linkse bit) gewoon genegeerd. Dit kunnen we laten zien aan de volgende optelling:

```
     00000101B      +5
+    11111011B  +   -5
---------------  -----
[1]  00000000B       0
```

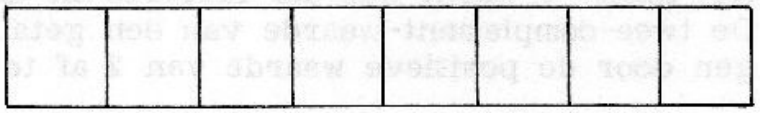
De optelling +5 + (-5) levert een carry (een negende bit) op. Als we dit bit gewoon negeren klopt het resultaat dat door de andere acht bits wordt aangegeven, in dit geval nul!; we hadden in dit geval niets anders verwacht.

Opgave A.11
Bereken de uitkomsten van -60 + 70, -23 + (-46), 85 - 96 en 5 - (-121). Voer de berekeningen in 'twee-complement' uit.

Bij gebruik van de twee-complement-methode kunnen we in een byte de volgende reeks getallen weergeven:

-128	10000000
-127	10000001
:	:
-2	11111110
-1	11111111
0	00000000
+1	00000001
+2	00000010
:	:
+126	01111110
+127	01111111

Wellicht kan het volgende u helpen bij de interpretatie van getallen zonder teken en getallen met teken (twee-complement-getallen). Let op het 'gewicht' van de plaats van een bit in de byte.



getallen zonder teken	128	64	32	16	8	4	2	1
getallen met teken	-128	64	32	16	8	4	2	1

Zo zal het getal zonder teken 10010001B gelijk zijn aan

1×128 + 1×16 + 1×1 = 145

terwijl het twee-complement-getal 10010001B gelijk is aan

1×-128 + 1×16 + 1×1 = -111.

Appendix B
Hexadecimale-decimale Conversietabellen

De onderstaande tabel geeft de directe conversie van hexadecimale getallen tussen 00 en FF naar decimale getallen tussen 0 en 255.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	000	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015
10	016	017	018	019	020	021	022	023	024	025	026	027	028	029	030	031
20	032	033	034	035	036	037	038	039	040	041	042	043	044	045	046	047
30	048	049	050	051	052	053	054	055	056	057	058	059	060	061	062	063
40	064	065	066	067	068	069	070	071	072	073	074	075	076	077	078	079
50	080	081	082	083	084	085	086	087	088	089	090	091	092	093	094	095
60	096	097	098	099	100	101	102	103	104	105	106	107	108	109	110	111
70	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
80	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
90	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A0	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B0	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C0	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D0	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E0	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F0	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Voor de conversie van grotere getallen kunt u deze tabel op indirecte wijze gebruiken door eerst gebruik te maken van de tabel op p.146.

Hexadecimaal	Decimaal
100	256
200	512
300	768
400	1024
500	1280
600	1536
700	1792
800	2048
900	2304
A00	2560
B00	2816
C00	3072
D00	3328
E00	3584
F00	3840
1000	4096
2000	8192
3000	12288
4000	16384
5000	20480
6000	24576
7000	28672
8000	32768
9000	36864
A000	40960
B000	45056
C000	49152
D000	53248
E000	57344
F000	61440

Appendix C
Samenvatting van de Z80 Instructies

Deze appendix bevat een volledig overzicht van alle Z80-instructies.

De eerste tabel, C.1, geeft een overzicht van de wijze waarop de diverse instructies het vlagregister beïnvloeden.

De tabellen C.2 tot en met C.12 bevatten gegevens over de Z80-instructies. Elke tabel bevat een aantal logisch bij elkaar horende instructies (een instructiegroep). Voor elke instructie bevat een tabel de mnemonic (logische OP-code), de numerieke OP-code, de symbolische bewerking, de inhoud van het vlagregister na afloop van het uitvoeren van de instructie, het aantal bytes dat een instructie 'lang' is, het aantal geheugencycles en het totaal aantal 'T states' (externe klokperioden) nodig voor het ophalen en uitvoeren van de instructie. Elke tabel spreekt praktisch voor zich, dat wil zeggen zo min mogelijk wordt verwezen naar de tekst of naar andere tabellen.

Instructie	C	Z	P/V	S	N	H	
ADD A,s;ADC A,s	↕	↕	V	↕	0	↕	8-bit optelling of optelling met carry
SUB s;SBC A,s,CP s,NEG	↕	↕	V	↕	1	↕	8-bit aftrekking, aftrekking met carry, compare en negatie accumulator
AND s	0	↕	P	↕	0	1	Logische bewerkingen
OR s;XOR s	0	↕	P	↕	0	0	And beïnvloedt diverse vlaggen
INC s	•	↕	V	↕	0	↕	8-bit ophoging (increment)
DEC m	•	↕	V	↕	1	↕	8-bit verlaging (decrement)
ADD DD,ss	↕	•	•	•	0	X	16-bit optelling
ADC HL,ss	↕	↕	V	↕	0	X	16-bit optelling met carry
SBC HL,ss	↕	↕	V	↕	1	X	16-bit aftrekking met carry
RLA;RLCA,RRA,RRCA	↕	•	•	•	0	0	Roteer accumulator
RL m;RLC m;RR m;RRC m SLA m;SRA m;SRL m	↕	↕	P	↕	0	0	Roteer en shift geheugenplaats m
RLD,RRD	•	↕	P	↕	0	0	Roteer digit links en rechts
DAA	↕	↕	P	↕	•	↕	Decimal adjust accumulator
CPL	•	•	•	•	1	1	Complement accumulator
SCF	1	•	•	•	0	0	Zet carry vlag
CCF	↕	•	•	•	0	X	Complement carry vlag
IN r,(C)	•	↕	P	↕	0	0	Invoer register indirect
INI;IND;OUTI;OUTD	•	↕	X	X	1	X	Blok invoer en uitvoer
INIR;INDR;OTIR;OTDR	•	1	X	X	1	X	Z=0 als B≠0, anders Z=1
LDI,LDD	•	X	↕	X	0	0	Blok-verplaats instructies
LDIR,LDDR	•	X	0	X	0	0	P/V=1 als BC≠0, anders P/V=0
CPI,CPIR,CPD,CPDR	•	↕	↕	↕	1	X	Blok-opzoek instructies Z=1 als A=(HL), anders Z=0 P/V=1 als BC≠0, anders P/V=0
LD A,I;LD A,R	•	↕	IFF	↕	0	0	De inhoud van de interrupt enable flip-flop (IFF) wordt gekopieerd in de P/V vlag
BIT b,s	•	↕	X	X	0	1	De status (0 of 1) van bit b van geheugenplaats s wordt gekopieerd in de Z-vlag
NEG	↕	↕	V	↕	1	↕	Negatie van accumulator

Tabel C.1 - Samenvatting vlagzetting. Courtesy Zilog, Inc.

Verklaring van de gebruikte tekens

Symbool	Verklaring
C	Carry/link vlag. C=1 indien carry van Meest Significante Bit van operand of uitkomst
Z	Zerovlag. Z=1 als uitkomst van bewerking nul is.
S	Sign vlag. S=1 als MSB of resultaat één is.
P/V	Pariteit of Overflow vlag. Pariteit (P) en Overflow (O) gebruiken dezelfde vlag. Logische bewerkingen zetten de P-vlag naar gelang de pariteit van de uitkomst terwijl rekenkundige bewerkingen de V-vlag zetten bij overflow van de uitkomst. Als P/V pariteit aangeeft dan P/V=1 als resultaat van de bewerking even is en P/V=0 als uitkomst oneven is. Als P/V overflow aangeeft dan P/V=1 als een bewerking overflow ten gevolge heeft.
H	Half carry vlag. H=1 geeft carry bij optelling (of aftrekking) in bit 4 van de accumulator.
N	Add/Subtract vlag. N=1 als vorige bewerking een aftrekking was. H en N vlaggen worden samen met de DAA instructie gebruikt om de uitkomst van een optelling of aftrekking met BCD-getallen te corrigeren tot een goede BCD-uitkomst.
↕	De vlag wordt beïvloed afhankelijk van de uitkomst van de bewerking.
•	De vlag wordt niet beïnvloed.
0	De vlag wordt 'gereset'.
1	De vlag wordt 'geset'.
X	De vlag is een 'don't care'.
V	P/V vlag wordt beïvloed door overflow.
P	P/V vlag wordt beïnvloed door pariteit.
r	Een van de CPU registers A,B,C,D,E,H of L.
s	Een 8-bit geheugenplaats in alle adresseermethoden toegestaan bij die instructie.
ss	Een 16-bit geheugenplaats.
ii	Een van de indexregisters IX en IY.
R	Refresh counter.
n	8-bit waarde <0,255>.
nn	16-bit waarde <0,65535>.
m	Een 8-bit geheugenplaats in alle adresseertechnieken voor een bepaalde instructie.

Mnemonic	symbolische bewerking	vlaggen C Z P/V S N H	OP-code 76 543 210	aantal bytes	aantal M cycles	aantal T states	opmerking
LD r,r'	r ← r'	. . . . .	01 r r'	1	1	4	r,r' Reg.
LD r,n	r ← n	. . . . .	00 r 110	2	2	7	000 B
			← n →				001 C
LD r,(HL)	r ← (HL)	. . . . .	01 r 110	1	2	7	010 D
LD r,(IX+d)	r ← (IX+d)		11 011 101	3	5	19	011 E
			01 r 110				100 H
			← d →				101 L
LD r,(IY+d)	r ← (IY+d)	. . . . .	11 111 101	3	5	19	111 A
			01 r 110				
			← d →				
LD (HL),r	(HL) ← r	. . . . .	01 110 r	1	2	7	
LD (IX+d),r	(IX+d) ← r	. . . . .	11 011 101	3	5	19	
			01 110 r				
			← d →				
LD (IY+d),r	(IY+d) ← r	. . . . .	11 111 101	3	5	19	
			01 110 r				
			← d →				
LD (HL),n	(HL) ← n	. . . . .	00 110 110	2	3	10	
			← n →				
LD (IX+d),n	(IX+d) ← n	. . . . .	11 011 101	4	5	19	
			00 110 110				
			← d →				
			← n →				
LD (IY+d),n	(IY+d) ← n	. . . . .	11 111 101	4	5	19	
			00 110 110				
			← d →				
			← n →				
LD A,(BC)	A ← (BC)	. . . . .	00 001 010	1	2	7	
LD A,(DE)	A ← (DE)	. . . . .	00 011 010	1	2	7	
LD A,(nn)	A ← (nn)	. . . . .	00 111 010	3	4	13	
			← n →				
			← n →				
LD (BC),A	(BC) ← A	. . . . .	00 000 010	1	2	7	
LD (DE),A	(DE) ← A	. . . . .	00 010 010	1	2	7	
LD (nn),A	(nn) ← A	. . . . .	00 110 010	3	4	13	
			← n →				
			← n →				
LD A,I	A ← I	. ↕ IFF ↕ 0 0	11 101 101	2	2	9	
			01 010 111				
LD A,R	A ← R	. ↕ IFF ↕ 0 0	11 101 101	2	2	9	
			01 011 111				
LD I,A	I ← A	. . . . .	11 101 101	2	2	9	
			01 000 111				
LD R,A	R ← A	. . . . .	11 101 101	2	2	9	
			01 001 111				

r,r' staan voor één van de registers A,B,C,D,E,H, of L
IFF : de inhoud van de interrupt 'enable flip-flop' (IFF) wordt gekopieerd in de P/V vlag.

Vlagnotatie: . = wordt niet beïnvloed, 0 = vlag reset, 1 = vlag set, X = vlag onbekend,
↕ = vlag wordt afhankelijk van de uitkomst van de bewerking beïnvloed.

Tabel C.2 - 8-bit load groep. Courtesy Zilog, Inc.

Mnemonic	symbolische bewerking	vlaggen C Z P/V S N H	OP-code 76 543 210	aantal bytes	aantal M cycles	aantal T states	opmerking
LD dd,nn	dd ← nn	. . . . .	00 dd0 001 ← n → ← n →	3	3	10	dd Paar 00 BC 01 DE
LD IX,nn	IX ← nn	. . . . .	11 011 101 00 100 001 ← n → ← n →	4	4	14	10 HL 11 SP
LD IY,nn	IY ← nn	. . . . .	11 111 101 00 100 001 ← n → ← n →	4	4	14	
LD HL,(nn)	H ← (nn+1) L ← (nn)	. . . . .	00 101 010 ← n → ← n →	3	5	16	
LL dd,(nn)	dd_H←(nn+1) dd_L←(nn)	. . . . .	11 101 101 01 dd1 011 ← n → ← n →	4	6	20	
LD IX,(nn)	IX_H←(nn+1) IX_L←(nn)	. . . . .	11 011 101 00 101 010 ← n → ← n →	4	6	20	
LD IY,(nn)	IY_H←(nn+1) IY_L←(nn)	. . . . .	11 111 101 00 101 010 ← n → ← n →	4	6	20	
LD (nn),HL	(nn+1) ← H (nn) ← L	. . . . .	00 100 010 ← n → ← n →	3	5	16	
LD (nn),dd	(nn+1)←dd_H (nn) ← dd_L	. . . . .	11 101 101 01 dd0 011 ← n → ← n →	4	6	20	
LD (nn),IX	(nn+1)←IX_H (nn) ← IX_L	. . . . .	11 011 101 00 100 010 ← n → ← n →	4	6	20	
LD (nn),IY	(nn+1)←IY_H (nn) ← IY_L	. . . . .	11 111 101 00 100 010 ← n → ← n →	4	6	20	
LD SP,HL	SP ← HL	. . . . .	11 111 001	1	1	6	
LD SP,IX	SP ← IX	. . . . .	11 011 101 11 111 001	2	2	10	
LD SP,IY	SP ← IY	. . . . .	11 111 101 11 111 001	2	2	10	
PUSH qq	(SP-2)←qq_L (SP-1)←qq_H	. . . . .	11 qq0 101	1	3	11	qq Paar 00 BC 01 DE
PUSH IX	(SP-2)←IX_L (SP-1)←IX_H	. . . . .	11 011 101 11 100 101	2	4	15	10 HL 11 AF
PUSH IY	(SP-2)←IY_L (SP-1)←IY_H	. . . . .	11 111 101 11 100 101	2	4	15	
POP qq	qq_H←(SP+1) qq_L ← (SP)	. . . . .	11 qq0 001	1	3	10	
POP IX	IX_H←(SP+1) IX_L ← (SP)	. . . . .	11 011 101 11 100 001	2	4	14	
POP IY	IY_H←(SP+1) IY_L ← (SP)	. . . . .	11 111 101 11 100 001	2	4	14	

Tabel C.3 - 16-bit load groep

Courtesy Zilog, Inc.

Legenda tabel C.3

dd is één van de registerparen BC, DE, HL, SP
qq is één van de registerparen AF, BC, DE, HL
De indexen H en L refereren naar respectievelijk het hoge-orde 8-bit register en het lage-orde 8-bit register van een registerpaar:
BC_L = C en BC_H = B.

Vlagnotatie: • wordt niet beïnvloed, 0 = vlag reset, 1 = vlag set,
X = vlag onbekend,
↕ = vlag wordt afhankelijk van de uitkomst van de bewerking beïnvloed.

Legenda tabel C.4

① P/V vlag is 0 als het resultaat van BC-1 nul is, anders P/V = 1
② Z vlag is 1 als A = (HL), anders Z = 0

Vlagnotatie: • wordt niet beïnvloed, 0 = vlag reset, 1 = vlag set,
X = vlag onbekend,
↕ = vlag wordt afhankelijk van de uitkomst van de bewerking beïnvloed.

Mnemonic	symbolische bewerking	vlaggen C Z V/P S N H	OP-code 76 543 210	aantal bytes	aantal M cycles	aantal T states	opmerking
EX DE,HL	DE ↔ HL	· · · · · ·	11 101 011	1	1	4	
EX AF,AF'	AF ↔ AF'	· · · · · ·	00 001 000	1	1	4	
EXX	(BC DE HL) ↔ (BC' DE' HL')	· · · · · ·	11 011 001	1	1	4	Uitwisselen van gewone en alternatieve registers
EX (SP),HL	H ↔ (SP+1) L ↔ (SP)	· · · · · ·	11 100 011	1	5	19	
EX (SP),IX	IX_H ↔ (SP+1) IX_L ↔ (SP)	· · · · · ·	11 011 101 11 100 011	2	6	23	
EX (SP),IY	IY_H ↔ (SP+1) IY_L ↔ (SP)	· · · · · ·	11 111 101 11 100 011	2	6	23	
LDI	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	· · ① · 0 0	11 101 101 10 100 000	2	4	16	Laadt (HL) in (DE), verhoogt de pointers en verlaagt de byte-teller (BC)
LDIR	(DE)←(HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 Herhaal tot BC=0	· · 0 · 0 0	11 101 101 10 110 000	2 2	5 4	21 16	Als BC ≠ 0 Als BC = 0
LDD	(DE)←(HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	· · ① · 0 0	11 101 101 10 101 000	2	4	16	
LDDR	(DE)←(HL) DE ← DE-1 HL ←HL-1 BC ← BC-1 Herhaal tot BC=0	· · 0 · 0 0	11 101 101 10 111 000	2 2	5 4	21 16	Als BC≠0 Als BC=0
CPI	A - (HL) HL ← HL+1 BC ← BC-1	· ① ① ① 1 ①	11 101 101 10 100 001	2	4	16	
CPIR	A - (HL) HL ← HL+1 BC ← BC-1 Herhaal tot A=(HL) of BC=0	· ① ① ① 1 ①	11 101 101 10 110 001	2 2	5 4	21 16	Als BC≠0 en A≠(HL) Als BC=0 of A=(HL)
CPD	A - (HL) HL ← HL-1 BC ← BC-1	· ① ① ① 1 ①	11 101 101 10 101 001	2	4	16	
CPDR	A - (HL) HL ← HL-1 BC ← BC-1 Herhaal tot A=(HL) of BC=0	· ① ① ① 1 ①	11 101 101 10 111 001	2 2	5 4	21 16	Als BC≠0 en A≠(HL) Als BC=0 of A=(HL)

Tabel C.4 - Uitwissel (Exchange)-, Blok-verplaats en Blok-opzoek groep. Courtesy Zilog, Inc.

Mnemonic	symbolische bewerking	vlaggen C Z P/V S N H	OP-code 76 543 210	aantal bytes	aantal M cycles	aantal T states	opmerking
ADD A,r	A ← A+r	↕ ↕ V ↕ 0 ↕	10 [000] r	1	1	4	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
ADD A,n	A ← A+n	↕ ↕ V ↕ 0 ↕	11 [000] 110 ← n →	2	2	7	
ADD A,(HL)	A←A+(HL)	↕ ↕ V ↕ 0 ↕	10 [000] 110	1	2	7	
ADD A,(IX+d)	A←A+(IX+d)	↕ ↕ V ↕ 0 ↕	11 011 101 10 [000] 110 ← d →	3	5	19	
ADD A,(IY+d)	A←A+(IY+d)	↕ ↕ V ↕ 0 ↕	11 111 101 10 [000] 110 ← d →	3	5	19	
ADC A,s	A←A+s+CY	↕ ↕ V ↕ 0 ↕	[001]				s is r,n,(HL), (IX+d) of (IY+d) zoals weergegeven bij de ADD instructie De aangegeven bits vervangen de 000 in de ADD instructies hierboven
SUB s	A ← A-s	↕ ↕ V ↕ 1 ↕	[010]				
SBC A,s	A←A-s-CY	↕ ↕ V ↕ 1 ↕	[011]				
AND s	A ← A∧s	0 ↕ P ↕ 0 1	[100]				
OR s	A ← A∨s	0 ↕ P ↕ 0 0	[110]				
XOR s	A ← A ⊕ s	0 ↕ P ↕ 0 0	[101]				
CP s	A - s	↕ ↕ V ↕ 1 ↕	[111]				
INC r	r ← r+1	• ↕ V ↕ 0 ↕	00 r [100]	1	1	4	
INC (HL)	(HL)←(HL)+1	• ↕ V ↕ 0 ↕	00 110 [100]	1	3	11	
INC (IX+d)	(IX+d) ← (IX+d)+1	• ↕ V ↕ 0 ↕	11 011 101 00 110 [100] ← d →	3	6	23	
INC (IY+d)	(IY+d) ← (IY+d)+1	• ↕ V ↕ 0 ↕	11 111 101 00 110 [100] ← d →	3	6	23	
DEC m	m ← m-1	• ↕ V ↕ 1 ↕	[101]				m is r,(HL), (IX+d) of (IY+d) zoals aangegeven bij INC. Zelfde format en toestanden als INC. Vervang 100 door 101 in de OP-code.

Legenda:
De V in de kolom van de P/V vlag betekent dat de P/V vlag overflow in het resultaat van de bewerking aangeeft. Zo betekent de letter P dat niet overflow, maar pariteit aangegeven wordt. V=1 betekent overflow, V=0 geen overflow. P=1 betekent even pariteit, P=0 oneven pariteit.
Vlagnotatie: • wordt niet beïnvloed, 0 = vlag reset, 1 = vlag set, X = vlag onbekend,
↕ = vlag wordt afhankelijk van de uitkomst van de bewerking beïnvloed.

Tabel c.5 - 8-bit rekenkundige en logische groep.
Courtesy Zilog, Inc.

Mnemonic	symbolische bewerking	vlaggen C Z P/V S N H	OP-code 76 543 210	aantal bytes	aantal M cycles	aantal T states	opmerking
DAA	Converteert acc.inhoud in packed BCD na een add of subtract met packed BCD operanden	↕ ↕ P ↕ • ↕	00 100 111	1	1	4	Decimal adjust accumulator
CPL	$A \leftarrow \overline{A}$	• • • • 1 1	00 101 111	1	1	4	Complement accumulator (één-complement)
NEG	A ← 0-A	↕ ↕ V ↕ 1 ↕	11 101 101 01 000 100	2	2	8	Negatie acc. (twee-complement)
CCF	$CY \leftarrow \overline{CY}$	↕ • • • 0 X	00 111 111	1	1	4	Complement carry vlag
SCF	CY ← 1	1 • • • 0 0	00 110 111	1	1	4	Zet carry vlag
NOP	Geen bewerking	• • • • • •	00 000 000	1	1	4	
HALT	CPU halt	• • • • • •	01 110 110	1	1	4	
DI	IFF ← 0	• • • • • •	11 110 011	1	1	4	
EI	IFF ← 1	• • • • • •	11 111 011	1	1	4	
IM 0	Zet interrupt mode 0	• • • • • •	11 101 101 01 000 110	2	2	8	
IM 1	Zet interrupt mode 1	• • • • • •	11 101 101 01 010 110	2	2	8	
IM 2	Zet interrupt mode 2	• • • • • •	11 101 101 01 011 110	2	2	8	

Legenda:
IFF geeft de Interrupt enable Flip-Flop aan.
CY staat voor de CarrY flip-flop.

Vlagnotatie: • wordt niet beïnvloed, 0 = vlag reset, 1 = vlag set, X = vlag onbekend,
↕ = vlag wordt afhankelijk van de uitkomst van de bewerking beïnvloed.

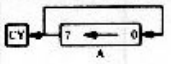
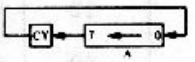
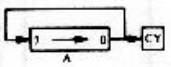
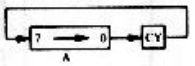
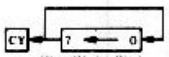
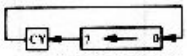
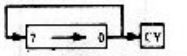
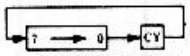
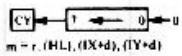
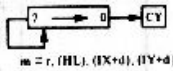
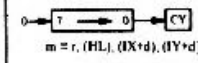
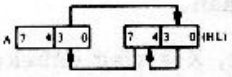
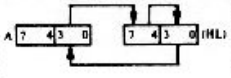
Tabel C.6 - Algemene rekenkundige en CPU besturingsgroep.
Courtesy Zilog, Inc.

Mnemonic	symbolische bewerking	vlaggen C Z V S N H	OP-code 76 543 210	aantal bytes	aantal M cycles	aantal T states	opmerking
ADD HL,ss	HL←HL+ss	↕ . . . 0 X	00 ss1 001	1	3	11	ss Reg. 00 BC
ADC HL,ss	HL←HL+ss+CY	↕ ↕ V ↕ 0 X	11 101 101 01 ss1 010	2	4	15	01 DE 10 HL
SBC HL,ss	HL←HL-ss-CY	↕ ↕ V ↕ 1 X	11 101 101 01 ss0 010	2	4	15	11 SP
ADD IX,pp	IX←IX+pp	↕ . . . 0 X	11 011 101 00 pp1 001	2	4	15	pp Reg. 00 BC 01 DE 10 IX 11 SP
ADD IY,rr	IY←IY+rr	↕ . . . 0 X	11 111 101 00 rr1 001	2	4	15	rr Reg. 00 BC 01 DE 10 IY 11 SP
INC ss	ss←ss+1	. . . . .	00 ss0 011	1	1	6	
INC IX	IX←IX+1	. . . . .	11 011 101 00 100 011	2	2	10	
INC IY	IY←IY+1	. . . . .	11 111 101 00 100 011	2	2	10	
DEC ss	ss←ss-1	. . . . .	00 ss1 011	1	1	6	
DEX IX	IX←IX-1	. . . . .	11 011 101 00 101 011	2	2	10	
DEC IY	IY←IY-1	. . . . .	11 111 101 00 101 011	2	2	10	

Legenda:
ss is één van de registerparen BC, DE, HL, SP
pp is één van de registerparen BC, DE, IX, SP
rr is één van de registerparen BC, DE, IY, SP

Vlagnotatie: . wordt niet beïnvloed, 0 = vlag reset, 1 = vlag set, X = vlag onbekend,
↕ = vlag wordt afhankelijk van de uitkomst van de bewerking beïnvloed.

Tabel C.7 - 16-bit rekenkundige groep. Courtesy Zilog, Inc.

Mnemonic	symbolische bewerking	Vlaggen C	Z	P/V	S	N	H	OP-code 76 543 210	aantal bytes	aantal M cycles	aantal T states	opmerking
RLCA		‡	•	•	•	0	0	00 000 111	1	1	4	Roteer acc. links cyclisch
RLA		‡	•	•	•	0	0	00 010 111	1	1	4	Roteer accumulator links
RRCA		‡	•	•	•	0	0	00 001 111	1	1	4	Roteer acc. rechts cyclisch
RRA		‡	•	•	•	0	0	00 011 111	1	1	4	Roteer accumulator rechts
RLC r		‡	‡	P	‡	0	0	11 001 011 00 000 r	2	2	8	Roteer reg. links cyclisch
RLC (HL)		‡	‡	P	‡	0	0	11 001 011 00 000 110	2	4	15	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
RLC (IX+d)		‡	‡	P	‡	0	0	11 011 101 11 001 011 ← d → 00 000 110	4	6	23	
RLC (IY+d)		‡	‡	P	‡	0	0	11 111 101 11 001 011 ← d → 00 000 110	4	6	23	
RL m		‡	‡	P	‡	0	0	010				Voor format en variaties zie RLC,m. Nieuwe OP-code door 000 uit RLC,m te vervangen door 010.
RRC m		‡	‡	P	‡	0	0	001				
RR m		‡	‡	P	‡	0	0	011				
SLA m		‡	‡	P	‡	0	0	100				
SRA m		‡	‡	P	‡	0	0	101				
SRL m		‡	‡	P	‡	0	0	111				
RLD		•	‡	P	‡	0	0	11 101 101 01 101 111	2	5	18	Roteer digit links en rechts tussen acc. en (HL). De inhoud v.d. meest significante helft v.d. acc. blijft onveranderd.
RRD		•	‡	P	‡	0	0	11 101 101 01 100 111	2	5	18	

Vlagnotatie: • wordt niet beïnvloed, 0 = vlag reset, 1 = vlag set, X = vlag onbekend,
‡ = vlag wordt afhankelijk van de uitkomst van de bewerking beïnvloed.

Tabel C.8 - Roteer- en shift-groep. Courtesy Zilog, Inc.

Mnemonic	symbolische bewerking	vlaggen C Z P/V S N H	OP-code 76 543 210	aantal bytes	aantal M cycles	aantal T states	opmerking
BIT b,r	$Z \leftarrow \overline{r}_b$	• ↕ X X 0 1	11 001 011 01 b r	2	2	8	r | Reg. 000 | B 001 | C 010 | D 011 | E 100 | H 101 | L 111 | A
BIT b,(HL)	$Z \leftarrow \overline{(HL)}_b$	• ↕ X X 0 1	11 001 011 01 b 110	2	3	12	
BIT b,(IX+d)	$Z \leftarrow \overline{(IX+d)}_b$	• ↕ X X 0 1	11 011 101 11 001 011 ← d → 01 b 110	4	5	20	
BIT b,(IY+d)	$Z \leftarrow \overline{(IY+d)}_b$	• ↕ X X 0 1	11 111 101 11 001 011 ← d → 01 b 110	4	5	20	b | getest bit 000 | 0 001 | 1 010 | 2 011 | 3 100 | 4 101 | 5 110 | 6 111 | 7
SET b,r	$r_b \leftarrow 1$	• • • • • •	11 001 011 [11] b r	2	2	8	
SET b,(HL)	$(HL)_b \leftarrow 1$	• • • • • •	11 001 011 [11] b 110	2	4	15	
SET b,(IX+d)	$(IX+d)_b \leftarrow 1$	• • • • • •	11 011 101 11 001 011 ← d → [11] b 110	4	6	23	
SET b,(IY+d)	$(IY+d)_b \leftarrow 1$	• • • • • •	11 111 101 11 001 011 ← d → [11] b 110	4	6	23	
RES b,m	$s_b \leftarrow 0$ m ≡ r,(HL), (IX+d), (IY+d)		[10]				Nieuwe OP-code: vervang [11] uit SET b,m door [10]. Vlaggen en tijden gelijk aan set instructie.

Legenda:
De notatie s_b geeft bit b (0 tot en met 7) uit geheugenplaats s aan.

Vlagnotatie: • wordt niet beïnvloed, 0 = vlag reset, 1 = vlag set, X = vlag onbekend,
↕ = vlag wordt afhankelijk van de uitkomst van de bewerking beïnvloed.

Tabel C.9 - Bit set, reset en test groep. Courtesy Zilog, Inc.

Mnemonic	symbolische bewerking	vlaggen C Z P/V S N H	OP-code 76 543 210	aantal bytes	aantal M cycles	aantal T states	opmerking
JP nn	PC ← nn	. . . . .	11 000 011 ← n → ← n →	3	3	10	
JP cc,nn	Als conditie cc true is PC ← nn anders ga door	. . . . .	11 cc 010 ← n → ← n →	3	3	10	cc | Voorwaarde 000 | NZ niet nul 001 | Z nul 010 | NC geen carry 011 | C carry 100 | PO oneven pariteit 101 | PE even pariteit 110 | P teken pos. 111 | M teken neg.
JR e	PC ← PC+e	. . . . .	00 011 000 ← e-2 →	2	3	12	
JR C,e	Als C=0 ga door	. . . . .	00 111 000 ← e-2 →	2	2	7	Niet aan voorwaarde voldaan
	Als C=1 PC ← PC+e			2	3	12	Wel aan voorwaarde voldaan
JR NC,e	Als C=1 ga door	. . . . .	00 110 000 ← e-2 →	2	2	7	Niet aan voorwaarde voldaan
	Als C=0 PC ← PC+e			2	3	12	Wel aan voorwaarde voldaan
JR Z,e	Als Z=1 ga door	. . . . .	00 101 000 ← e-2 →	2	2	7	Niet aan voorwaarde voldaan
	Als Z=0 PC ← PC+e			2	3	12	Wel aan voorwaarde voldaan
JR NZ,e	Als Z=1 ga door	. . . . .	00 100 000 ← e-2 →	2	2	7	Niet aan voorwaarde voldaan
	Als Z=0 PC ← PC+e			2	3	12	Wel aan voorwaarde voldaan
JP (HL)	PC ← HL	. . . . .	11 101 001	1	1	4	
JP (IX)	PC ← IX	. . . . .	11 011 101 11 101 001	2	2	8	
JP (IY)	PC ← IY	. . . . .	11 111 101 11 101 001	2	2	8	
DJNZ,e	B ← B-1 Als B=0 ga door	. . . . .	00 010 000 ← e-2 →	2	2	8	Als B=0
	Als B≠0 PC ← PC+e			2	3	13	Als B≠0

Legenda:
e is het relatieve sprongadres
e is een twee-complement getal <-126,129>
e-2 in de OP-code geeft een effectief adres van pc+e terwijl voor het optellen van e PC met 2 opgehoogd wordt

Vlagnotatie: . wordt niet beïnvloed, 0 = vlag reset, 1 = vlag set, X = vlag onbekend,
↕ = vlag wordt afhankelijk van de uitkomst van de bewerking beïnvloed.

Tabel C.10 - Spring (jump) groep. Courtesy Zilog, Inc.

Mnemonic	symbolische bewerking	vlaggen C Z P/V S N H	OP-code 76 543 210	aantal bytes	aantal M cycles	aantal T states	opmerking
CALL nn	$(SP-1) \leftarrow PC_H$ $(SP-2) \leftarrow PC_L$ $PC \leftarrow nn$	. . . . .	11 001 101 ← n → ← n →	3	5	17	
CALL cc,nn	Als conditie cc false is	. . . . .	11 cc 100 ← n → ← n →	3	3	10	Als cc false
	ga door			3	5	17	Als cc true
	anders hetzelfde als bij CALL nn						
RET	$PC_L \leftarrow (SP)$ $PC_H \leftarrow (SP+1)$	. . . . .	11 001 001	1	3	10	
RET cc	Als conditie cc false is	. . . . .	11 cc 000	1	1	5	Als cc false
	ga door			1	3	11	Als cc true
	anders hetzelfde als RET						
RETI	Return uit interrupt	. . . . .	11 101 101 01 001 101	2	4	14	
RETN	Return uit non maskable interrupt	. . . . .	11 101 101 01 000 101	2	4	14	
RST p	$(SP-1) \leftarrow PC_H$ $(SP-2) \leftarrow PC_L$ $PC_H \leftarrow 0$ $PC_L \leftarrow P$	. . . . .	11 t 111	1	3	11	

cc	Voorwaarde
000	NZ niet nul
001	Z nul
010	NC geen carry
011	C carry
100	PO oneven pariteit
101	PE even pariteit
110	P teken pos.
111	M teken neg.

t	P
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

Vlagnotatie: . wordt niet beïnvloed, 0 = vlag reset, 1 = vlag set, X = vlag onbekend,
↕ = vlag wordt afhankelijk van de uitkomst van de bewerking beïnvloed.

Tabel C.11 - Call en return groep. Courtesy Zilog, Inc.

Mnemonic	symbolische bewerking	vlaggen C Z P/V S N H	OP-code 76 543 210	aantal bytes	aantal M cycles	aantal T states	opmerking
IN A,(n)	A ← (n)	. . . . . .	11 011 011 ← n →	2	3	11	n naar A_0~A_7 Acc naar A_8~A_{15}
IN r,(C)	r ← (C) als r=110 worden alleen de vlaggen beïnvloed	. ↕ P ↕ 0 ↕	11 101 101 01 r 000	2	3	12	C naar A_0~A_7 B naar A_8~A_{15}
INI	(HL)←(C) B ← B-1 HL ← HL+1	① . ↕ X X 1 X	11 101 101 10 100 010	2	4	16	C naar A_0~A_7 B naar A_8~A_{15}
INIR	(HL)←(C) B ← B-1 HL ← HL+1 Herhaal tot B=0	. 1 X X 1 X	11 101 101 10 110 010	2 2	5 (Als B≠0) 4 (Als B=0)	21 16	C naar A_0~A_7 B naar A_8~A_{15}
IND	(HL)←(C) B ← B-1 HL ← HL-1	① . ↕ X X 1 X	11 101 101 10 101 010	2	4	16	C naar A_0~A_7 B naar A_8~A_{15}
INDR	(HL)←(C) B ← B-1 HL ← HL-1 Herhaal tot B=0	. 1 X X 1 X	11 101 101 10 111 010	2 2	5 (Als B≠0) 4 (Als B=0)	21 16	C naar A_0~A_7 B naar A_8~A_{15}
OUT (n),A	(n) ← A	. . . . . .	11 010 011 ← n →	2	3	11	n naar A_0~A_7 Acc naar A_8~A_{15}
OUT (C),r	(C) ← r	. . . . . .	11 101 101 01 r 001	2	3	12	C naar A_0~A_7 B naar A_8~A_{15}
OUTI	(C) ← (HL) B ← B-1 HL ← HL+1	① . ↕ X X 1 X	11 101 101 10 100 011	2	4	16	C naar A_0~A_7 B naar A_8~A_{15}
OTIR	(C) ← (HL) B ← B-1 HL ← HL+1 Herhaal tot B=0	. 1 X X 1 X	11 101 101 10 110 011	2 2	5 (Als B≠0) 4 (Als B=0)	21 16	C naar A_0~A_7 B naar A_8~A_{15}
OUTD	(C) ← (HL) B ← B-1 HL ← HL-1	① . ↕ X X 1 X	11 101 101 10 101 011	2	4	16	C naar A_0~A_7 B naar A_8~A_{15}
OTDR	(C) ← (HL) B ← B-1 HL ← HL-1 Herhaal tot B=0	. 1 X X 1 X	11 101 101 10 111 011	2 2	5 (Als B≠0) 4 (Als B=0)	21 16	C naar A_0~A_7 B naar A_8~A_{15}

Legenda:
① Als uitkomst van B-1 nul is wordt de Z-vlag 'gezet', anders wordt hij 'gereset'.

Vlagnotatie: . wordt niet beïnvloed, 0 = vlag reset, 1 = vlag set, X = vlag onbekend,
↕ = vlag wordt afhankelijk van de uitkomst van de bewerking beïnvloed.

Tabel C.12 - In- en uitvoergroep. Courtesy Zilog, Inc.

Appendix D – ASCII Tabel

b7 →					0	0	0	0	1	1	1	1
b6 →					0	0	1	1	0	0	1	1
b5 →					0	1	0	1	0	1	0	1
b4	b3	b2	b1	COL → / ↓ ROW	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	£ / #	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	A	LF	SUB	*	:	J	Z	j	z
1	0	1	1	B	VT	ESC	+	;	K	[	k	{
1	1	0	0	C	FF	FS	,	<	L	\	l	¦
1	1	0	1	D	CR	GS	-	=	M	]	m	}
1	1	1	0	E	SO	RS	.	>	N	^	n	~
1	1	1	1	F	SI	US	/	?	O	_	o	DEL

Een voorbeeld:
De ASCII code voor het teken K is binair 1001011, hexadecimaal 4B en decimaal 75.

Appendix E
Operatoren in Uitdrukkingen (Expressies)

De volgende operatoren mogen gebruikt worden in een operand, die als uitdrukking (expression) wordt opgegeven. De lijst is 'gesorteerd' op prioriteit van de operator bij de evaluatie van een expressie.

OPERATOR	FUNCTIE
+	unaire plus
-	unaire min
.NOT. or ¬	logische negatie
.RES.	resultaat
**	machtsverheffen
*	vermenigvuldiging
/	deling
.MOD.	modulo
.SHR.	logische shift rechts
.SHL.	logische shift links
+	optelling
-	aftrekking
.AND. of &	logische en
.OR. of ∧	logische of
.XOR.	logische exclusieve of
.EQ. of =	gelijk aan
.GT. of >	groter dan met teken
.LT. of <	kleiner dan met teken
.UGT.	groter dan zonder teken
.ULT.	kleiner dan zonder teken

Zie voor een toelichting p.164.

De RESultaat operator (.RES.) onderdrukt een eventuele overflow gedurende de evaluatie van een expressie. Indien hierbij overflow optreedt leidt dit dus niet tot een foutmelding tijdens het assembleren.

De MODulo operator (.MOD.) is gedefinieerd als

.MOD.B = A-B*(A/B)

waarbij A/B een integer deling is (dit is een deling met een geheel getal als uitkomst, zonder rest dus).

De SHift operatoren (.SHR. en .SHL.) krijgen twee argumenten mee. De eerste wordt zoveel bits 'verschoven' als door het tweede argument wordt aangegeven.

De vijf vergelijkende operatoren (.EQ., .GT., .LT., .UGT. en .ULT.) evalueren een uitdrukking tot TRUE als de vergelijking 'opgaat' en tot een logische FALSE (0) als de vergelijking niet opgaat.

Antwoorden van Opgaven uit de Tekst

De antwoorden staan op volgorde van opgavenummer, niet op volgorde van hoofdstuk! Hierdoor wordt vermeden dat men (doelbewust of per ongeluk) het antwoord op de volgende vraag reeds onder ogen krijgt. Dus eerst alle antwoorden van vragen, genummerd met .1, dan die met .2, .3, enzovoorts.

ANTWOORDEN VAN OPGAVEN GENUMMERD MET .1

1.1 Cassettebandjes en diskettes zijn gegevensdragers en géén in- en uitvoerapparatuur.

2.1 Omdat de waarde van n slechts één byte in beslag mag nemen.

3.1
```
; subroutine som van de enkelvoudige registers
;
SOMREG: LD   A,0
        ADD  A,B
        ADD  A,C
        ADD  A,D
        ADD  A,E
        ADD  A,H
        ADD  A,L
        RET
```

4.1
```
; programma voor een eindloze uitvoer van een *
;
        LD   A,'*'
WEER    CALL COUT
        JP   WEER
```

5.1

		A	S	Z
		-	-	-
LD	A,120	120	?	?
SUB	122	-2	1	0
LD	B,A	-2	1	0
SUB	B	0	0	1
ADD	A,70	70	0	0
NEG		-70	1	0

6.1 255 (11111111B)

7.1
```
        LD   A,'*'      wordt  4 keer uitgevoerd
        CALL COUT       wordt 24 keer uitgevoerd
        DEC  C          wordt  4 keer uitgevoerd
```

8.1 Ja

9.1 De Z-vlag zal op 1 gezet worden, want bit 1 van de accumulator bevat een 0.

10.1 De accumulator zal 53H en de carry vlag zal 1 bevatten.

11.1 Het resultaat van de logische operator XOR is 1 als de beide operanden verschillend zijn, anders is de uitkomst 0.

12.1

	A	Carry
	-	-----
RLA	01010110B	1
RLCA	10101100B	0
RRA	01010110B	0
RRCA	00101011B	0

13.1 -32768 tot en met +32767

14.1 Vervang LD B,10 door LD BC,aantal bytes
en DJNZ NEXBYT door DEC BC
LD A,B
CP 0
JP NZ,NEXBYT
LD A,C
CP 0
JP NZ,NEXBYT

15.1 01010111B

A.1 974H is equivalent met 2420.
101B is equivalent met 5.

ANTWOORDEN VAN OPGAVEN GENUMMERD MET .2

1.2 Respectievelijk de accumulator en de vlagregisters.
De accumulator bevat het resultaat. De vlagregisters bevatten informatie over de aard van het resultaat.

2.2
```
        LD   A,73
        ADD  A,55
        SUB  21
```

3.2 47H en +.

4.2 Een JP instructie neemt 3 bytes en een JR instructie neemt twee bytes in beslag.

5.2
```
i.      LD   A,X
ii.     SUB  10
iii.    JP   Z,GELIJK
iv.     LD   A,0
```

6.2
```
; programma voor het afdrukken van n sterretjes
;
        CALL CINEKO     ; voer cijfer in
        SUB  30H        ; zet code van cijfer om in de waarde
        LD   B,A
        LD   A,'*'
STER:   CALL COUT       ; druk * af
        DJNZ STER
        HALT
```

7.2
i. Impliciete adressering (accumulator wordt impliciet bedoeld).
ii. Register adressering (register D).
iii. Impliciete adressering (accumulator) en onmiddellijke adressering (de waarde 50).
iv. Register adressering (register A) en directe adressering (het adres 6352H).

8.2 CARRY

9.2
```
        BIT  0,A
        JR   Z,EVEN
ONEVEN: SET  7,B
        JR   GADOOR
EVEN:   RES   7,B
GADOOR: -
```

10.2

		B	C	Carry
		-	-	-----
LD	B,11	00001011B (+11)	?	?
SRA	B	00000101B (+5)	?	1
LD	C,-8	"	11111000B (-8)	1
SRA	C	"	11111100B (-4)	0

11.2

		A	S	Z	C
		-	-	-	-
LD	A,10110101B	10110101B	?	?	?
LD	C,11110000B	10110101B	?	?	?
AND	00011111B	00010101B	0	0	0
OR	C	11110101B	1	0	0
XOR	11001100B	00111001B	0	0	0
CPL		11000110B	0	0	0

12.2 De accumulator-roteerinstructies nemen elk een byte in beslag en ze zetten alleen de carry vlag. De register- en geheugenplaats-roteerinstructies nemen elk twee of vier bytes in beslag en zetten zowel de carry, de zero als de sign vlag.

13.2 RESMS zal 0668H bevatten en RESLS 0930H.

14.2
```
LD   HL,VAN+9    ; zet pointers op het einde
LD   DE,NAAR+9   ; van de blokken
LD   BC,10
LDDR
HALT
```

15.2 Een byte kan BCD-getallen van 00 tot en met 99 bevatten, binaire getallen zonder teken van 0 tot en met 255, dat wil zeggen meer dan de dubbele waarde van het grootste BCD-getal!

A.2 0 en 1.

ANTWOORDEN VAN OPGAVEN GENUMMERD MET .3

1.3 Decimaal 65536, hexadecimaal 10000.

2.3
```
        LD   A,56
        SUB  22
        LD   B,A
        ADD  A,B
        ADD  A,B
```

3.3 1008H

4.3
```
;  subroutine voor het invoeren en echoën van een teken
;
CINEKO: CALL CIN
        CALL COUT
        RET
```

5.3
```
;  programma voor het onderzoeken of B+C pos., neg. of nul is
;
        LD   A,B
        ADD  A,C        ; B + C
        JP   M,NEG
        JP   Z,NUL
        LD   A,'P'      ; positief
        CALL COUT
        JP   KLAAR
NEG:    LD   A,'N'      ; negatief
        CALL COUT
        JP   KLAAR
NUL:    LD   A,'Z'      ; nul
        CALL COUT
KLAAR:  -
```

6.3
```
;  programma voor het afdrukken van 9 t/m 1
;
        LD   A,39H      ; code van 9
CIJFER: CALL COUT
        DEC  A
        CP   30H
        JR   NZ,CIJFER
        HALT
```

7.3

```
                                  A       HL
                                  -       --
        LD    HL,N2               ?       1761H
        LD    A,(N1)             14H      1761H
        SUB   (HL)               37H      1761H
        LD    (VERS),A           37H      1761H
        LD    A,(N1)             14H      1761H
        ADD   A,(HL)             F1H      1761H
        LD    (SOM),A            F1H      1761H
        HALT                     F1H      1761H
```

8.3

```
        SCF                 ; zet carry vlag op 1
        CCF                 ; zet carry vlag andersom (dus nu nul)
```

9.3 161H

10.3

```
        SLA   A             ; 2 × N
        LD    B,A
        SLA   A             ; 2 × (2 × N)
        SLA   A             ; 2 × (2 × 2 × N)
        ADD   A,B           ; 2 × 2 × 2 × N + 2 × N
```

11.3

```
        OR    30H
```

12.3

```
        LD    A,B
        RRCA
        OR    C
        LD    (SEXLFT),A
```

13.3

```
        SCF                 ; reset carry vlag
        CCF                 ; op 0
        SUB   HL,BC
```

14.3

```
        LD    HL,VAN
        LD    DE,NAAR
        LD    BC,10
NEXBYT: LDI
        JP    PE,NEXBYT
        HALT
```

15.3

```
        00010111    BCD 17
      + 01101001    BCD 69
        --------
        10000000
          + 0110
        --------
        10000110    BCD 86
```

A.3 E8A5H = E×4096 + 8×256 + A×16 + 5×1
= 14×4096 + 8×256 + 10×16 + 5×1
= 57344 + 2048 + 160 + 5
= 59557

ANTWOORDEN VAN OPGAVEN GENUMMERD MET .4

1.4 Hexadecimaal B0

2.4

27	1BH
-27	E5H
-26	E6H

4.4 37H en 08H

5.4

```
        LD   A,(TELLER)
        CP   100
        JP   M,MINDER    ; TELLER < 100
        JP   Z,GELIJK    ; TELLER = 100
        JP   GROTER      ; TELLER > 100
```

6.4

		A	B	C	D	E	SP
		-	-	-	-	-	--
LD	A,0AH	0AH	?	?	?	?	?
LD	B,0BH	0AH	0BH	?	?	?	?
LD	C,0CH	0AH	0BH	0CH	?	?	?
LD	D,0DH	0AH	0BH	0CH	0DH	?	?
LD	E,0EH	0AH	0BH	0CH	0DH	0EH	?
LD	SP,16383	0AH	0BH	0CH	0DH	0EH	16383
PUSH	AF	0AH	0BH	0CH	0DH	0EH	16381
PUSH	BC	0AH	0BH	0CH	0DH	0EH	16379
PUSH	DE	0AH	0BH	0CH	0DH	0EH	16377
POP	BC	0AH	0DH	0EH	0DH	0EH	16379
POP	DE	0AH	0DH	0EH	0BH	0CH	16381

7.4

```
REGEL:  DEFM 'EERSTE REGEL'
        DEFB 0DH          ; CR code
        DEFB 0AH          ; LF code
        DEFM 'TWEEDE REGEL'
```

8.4

```
    10010101  (-107)
+   10010101  (-107)
-----------------------
[1] 00101010  (+42)
```

9.4 48H - de code voor het teken H.

11.4

```
QUATRE: MACRO
        SLA
        SLA
        ENDM
```

12.4 a. D8H
b. 2BH

13.4 Vervang de ADC OP-code door een SBC OP-code.

14.4
```
        LD   HL,HIER
        LD   DE,DAAR
        LD   BC,1000
NEXBYT  LDI
        LD   A,(HL)
        CP   0
        JP   NZ,NEXBYT
```

15.4
```
  10000010   BCD 82
- 01010110   BCD 56
----------
  00101100
    - 0110
----------
  00100110   BCD 26
```

A.4
```
  C7BAH          01101101B
- 9FF8H        + 01011110B
--------       ----------
  27C2H          11001011B
```

ANTWOORDEN VAN OPGAVEN GENUMMERD MET .5

1.5 10000001B

2.5 a. 35H
b. 79

7.5
```
        LD   HL,TEKST
WEER:   LD   A,(HL)
        CP   0
        JP   Z,HOMAAR
        CALL COUT
        INC  HL
        JP   WEER
HOMAAR: HALT
;
TEKST:  DEFM 'ABCDEFGHIJK'
        DEFB 0
```

11.5
```
BORP:   DEFL n          ; n = 1 voor beeldschermuitvoer
        -               ; n = 0 voor regeldrukkeruitvoer
        -
        COND BORP
AFDRUK: -               ; beeldschermuitvoersubroutine
        -
        RET
        ENDC
        -
        COND .NOT. BORP
AFDRUK: -               ; regeldrukkeruitvoersubroutine
        -
        RET
        ENDC
        -
```

12.5
```
                        Pariteit
        AND  OFEH           1
        SLA  A              0
        RLA                 0
```

14.5
```
        LD   HL,START+499
        LD   DE,START+599
        LD   BC,500
        LDDR
```

15.5

		A
		-
LD	A,43H	43H
LD	B,28H	43H
ADD	A,B	6BH
DAA		71H

A.5 62EH

ANTWOORDEN VAN OPGAVEN GENUMMERD MET .6

```
1.6          INC  B

12.6 CHKPAR: BIT  0,B
             JP   Z,EVTEST
             AND  0FFH
             JP   PO,OK
             JP   NOTOK
     EVTEST: AND  0FFH
             JP   PE,OK
     NOTOK:  LD   C,1
             JP   RETSUB
     OK:     LD   C,0
     RETSUB: RET
```

14.6 6, of 0 als het blok geen nul-waarde bevat.

A.6 FBH is equivalent met 251
A3B2H is equivalent met 41906
142 is equivalent met 8EH
9467 is equivalent met 24FBH

ANTWOORDEN VAN OPGAVEN GENUMMERD MET .7, .8 en .9

14.7 Vervang de instructies CPIR tot en met HALT door

```
NEXBYT: CPI
        JP   PO,FINI     ; einde blok?
        JR   NZ,NEXBYT
        LD   A,C         ; druk teller af
        ADD  A,30H
        CALL COUT
        LD   A,0         ; herstel A
        JR   NEXBYT
;
FINI:   HALT
```

A.7 9AB3H is equivalent met 1001101010110011B
110011101111B is equivalent met CEFH

A.8 1290 is equivalent met 50AH en 10100001010B
101110111101B is equivalent met BBDH en 3005

A.9 Getallen zonder teken van 0 tot en met 1111111111111111B (FFFFH of 65535) kunnen in twee bytes worden opgeslagen.

ANTWOORDEN VAN OPGAVEN GENUMMERD MET .10 en .11

A.10 -1 is hetzelfde als 11111111B
-2 is hetzelfde als 11111110B
-126 is hetzelfde als 10000010B
10000000B is hetzelfde als -128
10000001B is hetzelfde als -127

A.11

```
   11000100    -60
 + 01000110  + +70
[1] 00001010    +10

   11101001    -23
 + 11010010  + -46
[1] 10111011    -69

   01010101    +85
 - 01100000  - +96
[1] 11110101    -11

   00000101    +  5
 - 10000111  - -121
[1] 01111110    +126
```

Z80 ASSEMBLER Programmatuur voor de TRS-80, Apple, ZX81, ZX-Spectrum, Exidy Sorcerer, Newbrain, Osborne, Kaypro, Xerox 820, DEC Rainbow, VT 180 en 8" CP/M systemen

computer : TRS-80 model I of III
naam programma : INSTANT ASSEMBLER
leverancier : Mumford Micro Systems, U.S.A.
soort programma : editor, assembler, linking loader, debugger
medium : cassette of disk
vereist : TRS-80 model I of III met cassette of disk
bijzonderheden : Ideale ASSEMBLER en single-stepping DEBUGGER. Gebruikt slechts 8400 bytes zodat er in een 16K machine nog genoeg ruimte over is om assembly programma's van zo'n 2000 bytes te schrijven. Produceert relocatable modules die gelinkt kunnen worden met de bijgeleverde LINKING LOADER. Er kan rechtstreeks in het geheugen geassembleerd worden en het is mogelijk te switchen tussen Assembler en Debugger met behoud van source-code.

computer : TRS-80 model I of III
naam programma : DEMON
leverancier : Mumford Micro Systems, U.S.A.
soort programma : debugger, monitor, single stepper
medium : cassette (kan op disk worden gezet)
vereist : TRS-80 model I of III met cassette of disk
bijzonderheden : voor tape en disk systemen. Met dit programma is het mogelijk een machinetaal-programma instructie voor instructie door te lopen en daarbij het geheugenadres, de hexadecimale waarde, de Zilog-mnemonic en de registerinhoud te observeren. Zelfs programma's die ROM-routines gebruiken zijn te volgen. Tot de commando's behoren: step, disassemble, display en wijzig geheugen of CPU registers, voer een CALL instructie uit, zet breekpunten in RAM of ROM. Schrijf SYSTEM tapes en reloceer naar een willekeurig RAM adres. De uitvoer kan ook op de printer worden afgedrukt.

computer : Sinclair ZX81
naam programma : ZX AS
leverancier : Bug Byte Software, Engeland
soort programma : assembler
medium : cassette
vereist : ZX81 met 16K
bijzonderheden : Dit programma neemt 5K boven in het geheugen in beslag. Het te vertalen programma, dat uit alle standaard Zilog mnemonics mag bestaan, wordt in REM statements in het BASIC programma geschreven. Meer dan 1 opdracht per regel is toegestaan. ZX AS is een 2-pass assembler en kan samen met ZX DB gebruikt worden.

computer : Sinclair ZX81
naam programma : ZX DB
leverancier : Bug-Byte Software, Engeland
soort programma : disassembler en debugger
medium : cassette
vereist : ZX81 met 16K
bijzonderheden : Bevat een disassembler en mogelijkheden voor single-step, block search, transfer and fill, hex loader, register dispay en meer. Het programma wordt met behulp van simpele één-toets commando's via het toetsenbord bestuurd.

computer : Sinclair ZX Spectrum
naam programma : DEVPAC
leverancier : Hisoft, Engeland
soort programma : editor, assembler en monitor
medium : cassette
vereist : ZX Spectrum met 16 of 48 K
bijzonderheden : dit pakket bestaat uit de programma's GENS en MONS. GENS is een snelle 2-pass Z80-assembler met integrale editor. De assembler accepteert alle standaard ZILOG mnemonics en kent nog vele extra assembler commando's, o.a. voor conditional assembly. MONS is een relocatable monitor, single stepper, debugger.

computer : Apple met Z80 Softcard
naam programma : A.L.D.S.
leverancier : Microsoft Corporation, U.S.A.
soort programma : editor, assembler, debugger
medium : disk
vereist : Apple, Z80 Softcard, disk
bijzonderheden : Dit pakket bevat de volgende programma's: MS-Macro Assembler - een relocatable macro assembler die zowel Z80 als 8080 en 6502 opcodes accepteert, MS-LINK - een linking loader,

MS-CREF - een corss-reference programma, 6502 Debugger - een debugger voor 6502 assembly programma's en CPMXFER - een programma om CP/M-80 bestanden naar APPLE DOS over te zetten.

computer : CP/M systemen
formaat : 5.25" Osborne/Kaypro/Xerox820/DEC Rainbow/ VT180
8" standaard single density
naam programma : UVMAC
leverancier : The Software Toolworks, U.S.A.
soort programma : Z80 macro-assembler
medium : disk
vereist : CP/M computer
bijzonderheden : Accepteert volledige Z80 instructieset (Zilog mnemonics) en heeft mogelijkheden voor conditionele assembly, macro's en listing controle. Bevat tevens AS, een versie van de assembler die geen macro-mogelijkheden heeft, maar twee keer sneller werkt.

computer : Exidy Sorcerer
naam programma : ZETU
leverancier : System Software, Australië
soort programma : editor, assembler
medium : cassette
vereist : Exidy Sorcerer
bijzonderheden : Dit pakket bestaat uit een uitgebreide screen-editor en een 2-pass Assembler en biedt een beter alternatief voor het Development ROM PAC. De Assembler kent vele opties. Als er tijdens het assembleren een fout wordt ontdekt, wordt deze op het scherm afgedrukt en keert het programma terug naar de Edit-mode met de cursor in de foutieve regel. Source-, object- en link-files kunnen naar tape worden geschreven en weer worden ingelezen.

computer : Grundy NEWBRAIN
naam programma : DEVPAC
leverancier : Hisoft, Engeland
soort programma : editor, assembler en monitor
medium : cassette
vereist : NEWBRAIN
bijzonderheden : Dit pakket bestaat uit de programma's GENS en MONS. GENS is een snelle 2-pass Z80 assembler met integrale editor. De assembler accepteert alle standaard ZILOG mnemonics en kent nog vele extra assembler commando's, o.a. voor conditional assembly. MONS is een relocatable monitor, single stepper, debugger.

Index

Dit boek bevat een eerste, vrij volledige, inleiding in het leren programmeren in de assembleertaal van de Z-80 microprocessor. Programma's die in assembleertaal geschreven worden zijn veel 'sneller' dan programma's, geschreven in hogere programmeertalen als BASIC en Pascal. Bovendien kan in assembleertaal optimaal gebruik worden gemaakt van de mogelijkheden van de microprocessor.

Het boek is geschikt als studie- en leerboek bij diverse opleidingen, waarin het programmeren in assembleertaal aan de orde komt, maar zeer zeker ook voor de computerhobbyist die door zelfstudie een begin wil maken met het programmeren in assembleertaal. Al zeer snel is de lezer in staat zelf assembleertaalprogramma's te schrijven. De vele tekeningen, diagrammen en tabellen verduidelijken de tekst op tal van plaatsen. In de tekst is een groot aantal opgaven opgenomen, waarvan de uitwerkingen achter in het boek zijn opgenomen. Elk hoofdstuk eindigt met een programmeeropdracht.

De inhoud van dit boek is van toepassing op elk microcomputersysteem dat gebaseerd is op de Z-80 microprocessor. De minimaal vereiste configuratie is een Z-80 processor, een beeldscherm en een toetsenbord. Natuurlijk moet men ook beschikken over een assembler programma voor het vertalen van de assembleertaalinstructies naar machinecode.

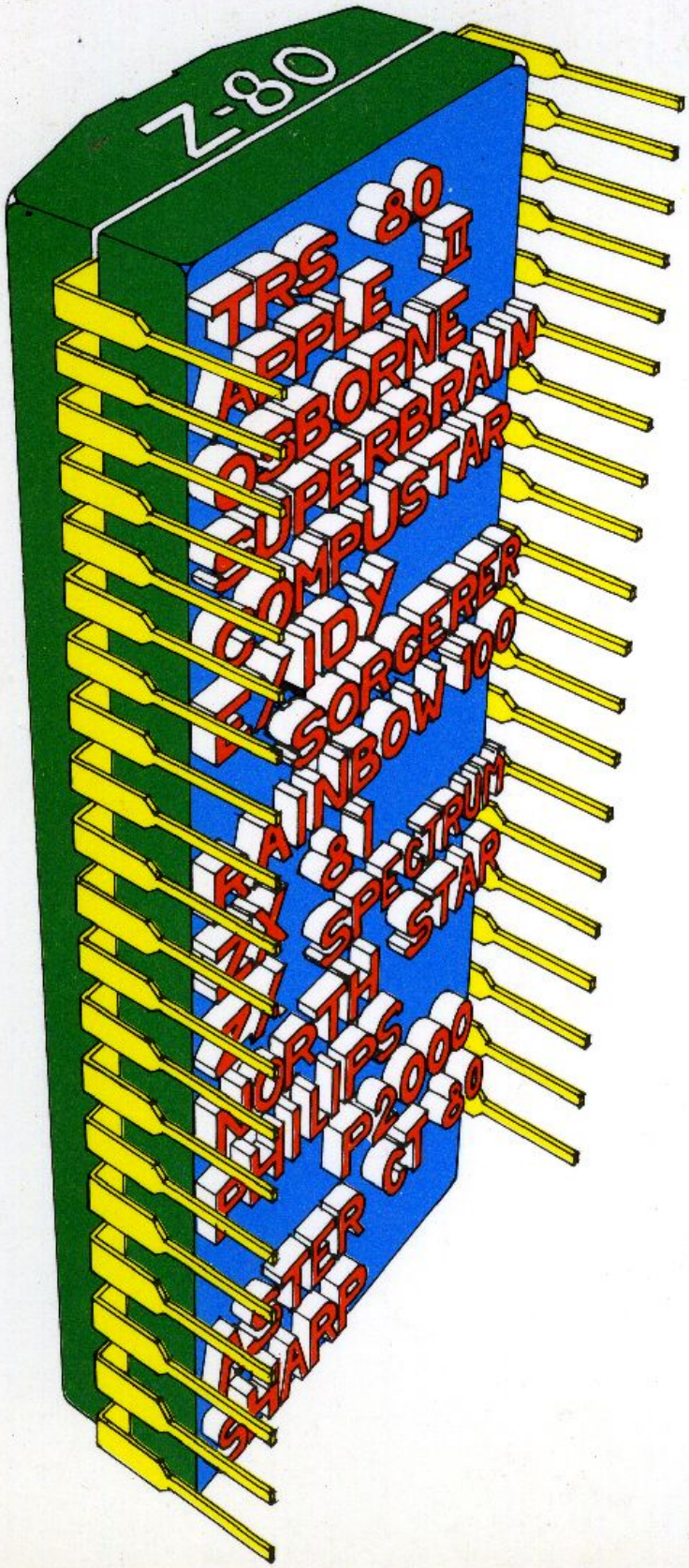
Op de Z-80 gebaseerde microcomputers zijn:

•Altos ACS 8000-2, 8000-6 t/m 8000-9 • Apple II (Z80 softcard)• Aster CT-80 Basf • Cormenco system one/DPU • Digital Rainbow 100 • Exidy Sorcerer • Heath/Zenith H-89,Z-89-FA • IDS-2009 • Intertec Compustar • Intertec Superbrain • Kaypro • Kontron PSI-80, PSI-80 DS2 • LNW 80 • Luxor ABC-80, ABC-800/Facit-6500 • Nascom • NEC PC-8000 • Newbrain • North Star • OKI IF-800 •Osborne • Philips P2000 T&M • RC Data Systems RC-700 Piccolo • Sharp MZ-80B, MZ-80K, PC-3201 • Shelton/SIG/NET • Sinclair ZX81, ZX Spectrum • Tandy TRS-80, model I,II,III • Televideo • Transam Tuscan • Xerox 820 • ZIlog MCZ-1/20, MCZ-2/20 • ZORBA

roger hutty **cursus Z-80 assembleertaal** ACADEMIC SERVICE

cursus Z-80 assembleertaal

roger hutty



ACADEMIC SERVICE

roger hutty cursus Z-80 assembleertaal ACADEMIC SERVICE