J.B.Vonk / E.J.J.Doppenberg

MACHINETAAL Z-80

Gestructureerd programmeren in theorie en praktijk

Kluwer Technische Boeken

Machinetaal
Z-80

J.B. Vonk/E.J.J. Doppenberg

Machinetaal Z-80

Gestructureerd programmeren in theorie en praktijk

Kluwer Technische Boeken B.V.
Deventer - Antwerpen

ISBN 90 201 2004 2
D/1987/0108/177
NUGI 434

1e druk 1987

Woord vooraf

De Z-80 mag dan een al wat oudere processor zijn, van verouderd is nog geen sprake. Dat bewijst de ontwikkeling van de markt, waarop nog steeds nieuwe microcomputers met als hart deze processor worden uitgebracht. Juist omdat de Z-80 al heel wat jaren meegaat, zijn ontwerpers er goed op thuis en is er veel software voor te krijgen. Een fabrikant kan hierdoor relatief goedkoop een microcomputer met een uitgebreid programmapakket leveren.
Natuurlijk heeft zo'n oude achtbitter ook nadelen, zoals de beperkte hoeveelheid adresseerbaar geheugen. Maar voor het zelf ontwikkelen van programma's, het gebruiken van een spreadsheet of tekstverwerker heeft lang niet iedereen geheugens van rond de megabyte nodig.
Dit boek is geschreven voor wie wil leren de Z-80 microprocessor zelf te programmeren met behulp van een microcomputer. Veel boeken over microprocessors leggen ofwel sterk de nadruk op de processor zelf, zonder deze als kern van een computer te beschouwen, of ze zijn toegespitst op een bepaalde computer. In het eerste geval gaat het om de behandeling van de instructieset en technische aangelegenheden, in het tweede geval om uit de computer te halen wat erin zit. Dit komt dan meestal neer op het creëren van flitsende schermen en muzikale effecten.
Bij het schrijven van dit boek stond ons het volgende voor ogen:

- Gestructureerd leren programmeren van de Z-80 met als basis enige bekendheid met een hogere programmeertaal als bijv. BASIC of Pascal.
- Het laten zien wat een microprocessor doet in het grotere geheel van de microcomputer.

Om dit te verwezenlijken, behandelen we onderwerpen als stringvergelijking, invoer/uitvoer, integer- en floating pointberekeningen, conversies en expressie-evaluatie. Zaken die niets nieuws toevoegen aan de microcomputer maar wel inzicht verschaffen in de werking ervan.
De programma's in dit boek zijn dus om van te leren. Waar we voor de alternatieven sneller of duidelijker stonden, is voor het laatste gekozen.
Bij het maken van dit boek ondervonden we veel steun van de firma E.C.D. b.v. uit

Delft die ons niet alleen met soft- en hardware terzijde stond maar ook met vakkennis en morele steuntjes in de rug. Voor hulp op softwaregebied bedanken we T. de Jong van Tedejo Produkties die juist dat programma waaraan behoefte was uit een la wist te toveren. Verder bedanken we de firma Zilog voor hun toestemming de beknopte instructieset in appendix A op te nemen. En dan zijn er natuurlijk nog velen die ons, soms zonder het zelf te weten, ertoe hebben aangespoord door te gaan. Zoals de vrouw in de nis langs de trap, de stadsarchitect, het meisje met de grote voeten en massa's anderen.

Dit boek is geschreven op een Acorn BBC computer met een 6502 second processor en een VIEW tekstverwerker. De programma's zijn ontwikkeld op een Acorn BBC met als second processor een Torch Z-80 en een Z-80 macro assembler van Digital Research.

Rotterdam, 1 juli 1986

J.B. Vonk
E.J.J. Doppenberg

Inhoudsopgave

1 Computer en microprocessor

1.1 Computertalen

Tot de hogere programmeertalen behoren onder andere BASIC, Pascal en BCPL. In deze en andere talen typt men op het toetsenbord van de computer een programma, bestaande uit zaken als variabelen, rekenkundige uitdrukkingen en tot de betreffende taal behorende woorden. Deze woorden geven in het Engels min of meer aan welk effect ze in het programma zullen hebben, zoals bijv. INPUT in BASIC.
Na probleemanalyse en programma-ontwerp volgen bij gebruik van BASIC meestal de volgende stadia:

intypen van het programma
uitvoeren van het programma
testen op programmeerfouten

Bij Pascal en BCPL komt daar, grof geschetst, nog een stadium bij:

intypen van het programma
compileren
uitvoeren van het programma
zoeken naar programmeerfouten

Bij het compileren zet een compiler de ingetypte programmatekst om in machinecode. Alleen het gecompileerde programma kan worden uitgevoerd. Is er een fout gevonden dan moet de programmeur deze in de getypte tekst herstellen en opnieuw compileren.
De machinecode, eindprodukt van het compileren, kan direct door de microprocessor in het hart van de computer worden verwerkt.
Om vanuit BASIC aan machinecode te komen, maakt deze taal gebruik van een interpreter. Tijdens het uitvoeren van het programma leest de interpreter het programma regel voor regel en start de bij BASIC-woorden en -uitdrukkingen behorende machinecodeprogramma's. PRINT "DAG" bijvoorbeeld start een machinecodeprogramma dat de letters tussen aanhalingstekens naar het beeldschermgeheugen brengt.

Eén en ander betekent wel dat er voor elk BASIC-woord een machinecodeprogramma in de computer aanwezig moet zijn. Het aantal woorden is dus beperkt. Het interpreteren kost tijd. Daarom is BASIC trager dan compileertalen. Voordeel van de BASIC-interpreter is het gemak in de omgang: een programma kan gestart, veranderd en meteen weer gestart worden.
Compileertalen hebben echter een bijkomend voordeel: doordat het compileren niet plaats vindt tijdens de uitvoer van het feitelijke programma, zodat de woorden uit de taal niet permanent in het geheugen hoeven staan, is er tijd en ruimte voor talen met een uitgebreide woordenschat. Pascal en BCPL hebben dan ook zeer veel mogelijkheden, vooral waar het gestructureerd programmeren betreft.
Bij microcomputers zet de compiler, vanwege het beperkte geheugen, het ingetypte programma soms om naar een zeer efficiënt met geheugenruimte omspringende code, die zich qua structuur ergens tussen hogere programmeertaal en machinecode bevindt. Men kan het programma dan laten 'draaien' met een speciaal voor deze code gemaakte interpreter.
Het ontwikkelen van programma's in machinecode gaat via dezelfde weg als bij compileertalen. Elke microprocessor beschikt over een eigen programmeertaal, de assembleertaal. In vergelijking met bijv. BASIC is deze zeer primitief. Is het programma in assembleertaal eenmaal ingetypt dan kan een assembler het, net als de compiler bij hogere programmeertalen, omzetten naar machinecode.
Samengevat:

BASIC:
De interpreter leest de BASIC-tekst en verwijst naar machinetaalprogramma's in het geheugen die de opdrachten uitvoeren.

Compileertalen als Pascal, BCPL en C:
De compiler leest de tekst van het programma en zet die om in machinetaal. Pas na het compileren kan het programma worden gestart.

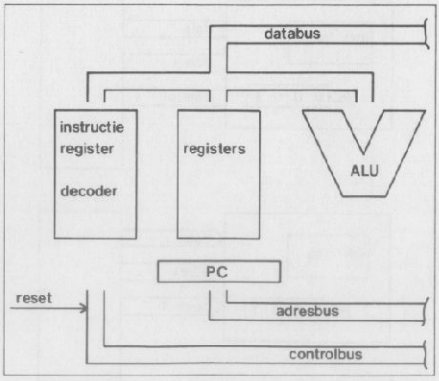
Assembleertaal:
De assembler leest de programmatekst en zet die om in machinetaal.

Interpreter, compiler en assembler zijn zelf meestal ook programma's in machinecode. Machinecode is het enige dat een microprocessor kan verwerken. Het bestaat uit getallen en die getallen zijn weer een voorstelling van de situatie op elektrisch gebied. Want al lijkt het soms alsof de computer BASIC-woorden begrijpt en daar met een foutmelding in het Engels op reageert, het is een elektrisch apparaat. Wie zich in de werking ervan verdiept, komt uiteindelijk terecht op het niveau van elektrische spanning en stroom.

1.2 Sequentieel afwerken van een programma

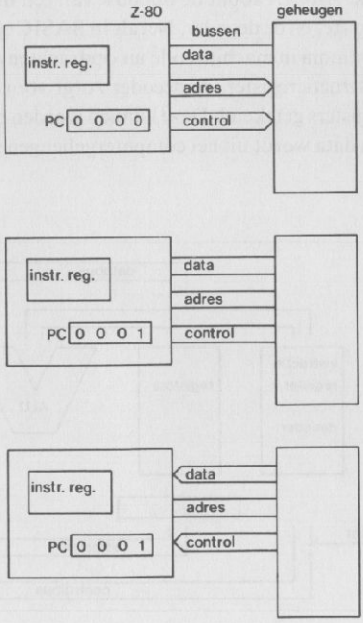
Het enige onderdeel in een computer dat een programma kan uitvoeren, is de microprocessor. Deze werkt niet alleen het interpreterprogramma voor BASIC of het in een andere hogere programmeertaal geschreven en daarna gecompileerde programma af, maar bestuurt ook vele van de overige chips in de computer, zoals die voor het opwekken van videosignalen en die voor de cassette- of disk-interface. De microprocessor regelt deze zaken door het uitvoeren van machinecodeprogramma's die de fabrikant in de computer heeft ingebouwd.

Programma's afwerken betekent in het algemeen een aantal opdrachten in de juiste volgorde uitvoeren. Het één voor één afwerken van instructies wordt sequentieel verloop genoemd. Een voorbeeld ervan is het regel na regel doorlopen van een BASIC-programma. Hoe de microprocessor dat kan, gaan we nu bekijken.
Hoewel microprocessors onderling nogal verschillen, zijn ze wat de principiële werking betreft gelijk. Afb. 1.1 toont de opbouw van een microprocessor. Links staat het instructieregister en de decoder. Net als in BASIC en andere computertalen bestaat een programma in machinecode uit opdrachten en data. De opdrachten komen in het instructieregister, de decoder zorgt voor de uitvoering ervan. Daarnaast zijn de registers getekend. Deze kunnen worden beschouwd als geheugentjes. Te bewerken data wordt uit het computergeheugen gehaald en in één van



Afb. 1.1 Schematisch overzicht van een microprocessor

de registers gezet. Bewerkte data gaat vanuit één van de registers naar het geheugen terug. Het bewerken van data vindt plaats in de ALU, de Arithmetic and Logical Unit ofwel de rekenkundige en logische eenheid.
Om instructies en data uit het geheugen te kunnen halen en andere chips in de computer te besturen, is een microprocessor voorzien van een flink aantal aansluitingen, naar hun functie in te delen bij één van de volgende drie groepen: de adresbus, de databus en de controlbus.
Het onderdeel dat een hoofdrol speelt bij het sequentieel verloop is de Program Counter, de programmateller, afgekort PC. Het geheugen van een computer is verdeeld in een groot aantal opeenvolgend genummerde vakjes en de PC wijst aan uit welk vakje de eerstvolgende instructie of data moet worden opgehaald.
Is de microprocessor eenmaal bezig de inhoud van een geheugenvakje op te halen dan verhoogt de PC zichzelf automatisch met één en wijst dus naar het volgende vakje. Op deze manier werkt de microprocessor instructies en data in opeenvolgende geheugenvakjes af. Om de PC bij het aanzetten van de computer een



Afb. 1.2 Het ophalen van de eerste instructie na een reset

bepaalde waarde te geven, namelijk die van het geheugenvakje waar het startprogramma begint, dient de reset-ingang. Een signaal op de reset-ingang van de Z-80 zet de PC op 0. Beginnend bij geheugenvakje 0 is in de computer een startprogramma ingebouwd. Dat programma zorgt er onder andere voor dat de microprocessor andere chips in het systeem juist instelt en eindigt meestal met vermelden van de firmanaam linksboven op het beeldscherm.
Afb. 1.2 laat in stappen zien wat er gebeurt na een reset. Links in de tekening bevindt zich de microprocessor, rechts het geheugen. Deze zijn met elkaar verbonden via de databus, de adresbus en de controlbus. De inhoud van PC is, zoals het hoort na een reset op de Z-80, gelijk aan nul.

Stap 1:
De PC wijst het nummer van het geheugenvakje aan waarin de eerste uit te voeren instructie staat. Het nummer van een geheugenvakje heet een adres.
De Z-80 zet de inhoud van de PC op de adresbus en geeft via één van de lijnen van de controlbus het geheugen een signaal de inhoud van het opgegeven adres te willen lezen. Behalve de inhoud van een adres lezen kan de microprocessor ook iets naar een adres schrijven.

Stap 2:
Het geheugen is bezig de inhoud van adres 0000 te zoeken. In de tussentijd verhoogt de Z-80 de inhoud van de PC met 1 tot 0001.

Stap 3:
Het geheugen heeft de inhoud van adres 0000 gevonden en op de databus gezet. Om dit te melden, stuurt het geheugen via de controlbus een signaal naar de Z-80. Deze leest via de databus de inhoud van adres 0000 in het instructieregister. Na deze derde stap voert de Z-80 de instructie uit. De inhoud van de PC is klaar voor het ophalen van de inhoud van geheugenlocatie 0001. Hierin kan bijv. data staan of een nieuwe instructie.
Met de adresbus selecteert een microprocessor een geheugenlocatie. Via de databus vindt uitwisseling van gegevens tussen microprocessor en geheugen plaats. Via de controlbus kunnen microprocessor en geheugen elkaar signalen geven om alles goed te laten verlopen.

1.3 Het geheugen

Iedere computer beschikt over een hoeveelheid geheugen in de vorm van chips.

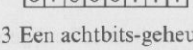
Een deel ervan is ROM, afkorting voor Read Only Memory, geheugen waaruit alleen kan worden gelezen. De inhoud ervan laat zich niet wijzigen. In ROM staat onder andere het startprogramma en in de meeste gevallen ook de BASIC-interpreter. Het overige geheugen bestaat uit RAM, Random Access Memory. Het is vrij toegankelijk geheugen. De microprocessor kan er gegevens uit lezen, maar er ook informatie in opslaan. In RAM staat een BASIC-programma, een tekst of een gegevensbestand. RAM in de computer is een vluchtig geheugen: de inhoud ervan verdwijnt als de spanning wordt uitgeschakeld. ROM en RAM in de computer zijn snel toegankelijke geheugens. Het kost heel weinig tijd om een gegeven uit het geheugen te halen. Hiernaast beschikt de computer over een veel trager achtergrondgeheugen in de vorm van een disk-drive of een cassetterecorder.

Alvorens nader in te gaan op de inhoud van het geheugen is het wenselijk een voorstelling te vormen omtrent de werking ervan. De eenvoudigste vorm van een elektrisch geheugen is de bekende 'stopknop' in de autobus. Drukt een passagier daarop dan gaat in de bus en bij de bestuurder een lamp branden. De schakeling onthoudt of er al dan niet iemand op de knop heeft gedrukt. Dit primitieve geheugen kent maar twee toestanden: ja, er heeft iemand op de knop gedrukt en nee, er heeft niemand op de knop gedrukt. Elektrisch gezien komen ja en nee overeen met wel of geen spanning op de lamp.

Een computergeheugen moet meer kunnen. De kleinste te onthouden eenheden zijn letters, cijfers en leestekens. Toch bestaat het computergeheugen uit cellen die net als de busschakeling maar twee toestanden kennen: ja en nee, wel spanning en geen spanning. Om aan de gestelde eisen te voldoen, wordt eenvoudigweg een groepje van deze cellen als eenheid beschouwd. In het geval van een op de Z-80 microprocessor gebaseerde computer zijn dat er acht. Een geheugenlocatie bestaat uit acht cellen. De Z-80 is een achtbits-processor (bit betekent stukje), wat wil zeggen dat de Z-80 het geheugen per acht bits (of cellen) tegelijk leest. De databus, verbinding tussen geheugen en de Z-80, bestaat dan ook uit acht lijnen.

1.4 De inhoud van het geheugen

Afb. 1.3 toont de inhoud van een achtbits-geheugenlocatie. De beide basistoestanden, spanning en geen spanning, zijn weergegeven met de cijfers 1 en 0. Bij een



Afb. 1.3 Een achtbits-geheugenlocatie

achtbits-processor heet een dergelijke eenheid een byte. Een eenheid van twee bits kan de volgende toestanden aannemen:

0 0
0 1
1 0
1 1

In totaal 2^2 ofwel vier verschillende mogelijkheden. Een eenheid van drie bits kent $2^3 = 8$ te onderscheiden toestanden:

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

Zou er nu een vierde bit bijkomen dan verdubbelt het aantal mogelijkheden. Het vierde bit kan immers 0 en 1 zijn. In beide gevallen kunnen de drie overige bits de acht hierboven geschetste toestanden aannemen. In totaal zijn er dan 16 mogelijkheden. In het kort: één bit kan twee toestanden aannemen, twee bits het dubbele ofwel 2^2, 3 bits daarvan het dubbele dus 2^3 en n bits 2^n.
Bij de eerder genoemde achtbits-eenheid, de byte, laten zich $2^8 = 256$ mogelijke toestanden onderscheiden. De inhoud van een geheugenlocatie varieert dan van 00000000 tot 11111111. Een dergelijke inhoud kan worden opgevat als een binair getal, dat wil zeggen een getal uit het binair of tweetallig stelsel. Gewoonlijk gebruikt iedereen het tientallig stelsel. Daarin bestaan tien verschillende cijfers en wel die van 0 t/m 9. Aantallen groter dan 9 worden als volgt weergegeven:
Eerst tellen tot 9 waarna er een 1 links van het getal komt en men weer bij 0, dus met 10, begint. Eenmaal bij 19 wordt de 1 in de linkerkolom verhoogd tot 2 en na 99 komt er een derde kolom bij om het getal 100 te krijgen.
Om te kunnen werken met binaire en de nog te introduceren hexadecimale getallen is het nodig goed te begrijpen wat een decimaal getal voorstelt. 625 bijvoorbeeld betekent:

$$\begin{aligned} 5 \times 10^0 = 5 \times 1 &= 5 \\ 2 \times 10^1 = 2 \times 10 &= 20 \\ 6 \times 10^2 = 6 \times 100 &= 600 \\ \hline & 625 \end{aligned} +$$

Het bovenstaande geldt ook voor het tweetallig stelsel. Dit kent echter maar twee getallen, namelijk 0 en 1. Bij het tellen komt er dus niet na 9 maar al na 1 een extra kolom bij.

binair	decimaal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7

Naar analogie met de decimale getallen betekent het binaire getal 110:

$$\begin{aligned} 0 \times 2^0 &= 0 \times 1 = 0 \\ 1 \times 2^1 &= 1 \times 2 = 2 \\ 1 \times 2^2 &= 1 \times 4 = \underline{4} + \\ & \qquad\quad 6 \end{aligned}$$

Het decimale of tientallige stelsel werkt met het grondtal 10, het binaire of tweetallige met het grondtal 2.
De inhoud van een achtbits geheugenlocatie is een binair getal. Dat getal kan een opdracht in machinecode zijn maar ook een getal, een cijfer of een letter. In het instructieboek van elke computer staat welke getalswaarde aan welke letter is toegekend. Een bekende standaard hiervoor is de ASCII-code. Daarin heeft de letter A de waarde 65. Staat de letter A in een geheugenlocatie dan is daarvan de binaire inhoud 0 1 0 0 0 0 0 1. Wat overeenkomt met:

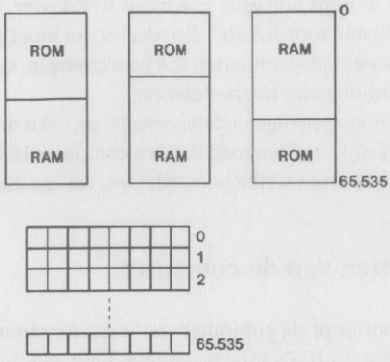
$$\begin{aligned} 1 \times 2^0 &= 1 \times 1 \;\; = \;\; 1 \\ 1 \times 2^6 &= 1 \times 64 = \underline{64} + \\ & \qquad\quad 65 \end{aligned}$$

Staat in een BASIC-programma het statement PRINT "DAG" dan bevindt het woord DAG zich in drie opeenvolgende geheugenlocaties:

0 1 0 0 0 1 0 0 = 68 ASCII-code voor D
0 1 0 0 0 0 0 1 = 65 ASCII-code voor A
0 1 0 0 0 1 1 1 = 71 ASCII-code voor G

1.5 De indeling van het geheugen

Het geheugen bestaat uit een grote hoeveelheid bytes die opeenvolgend zijn genummerd, te beginnen bij 0.



Afb. 1.4 De indeling van het geheugen

Afb. 1.4 laat mogelijke geheugenindelingen (zgn. memory maps) zien van een computer gebaseerd op een achtbits microprocessor. De meest rechtse kan niet van een computer met Z-80 processor zijn: adres 0 behoort tot het RAM-gedeelte van het geheugen en kan dus geen startprogramma bevatten. Het zou de memory map kunnen zijn van een computer met een 6502, een andere heel populaire processor. Deze zoekt na een reset het beginadres van het startprogramma in de adressen 65.532 en 65.533.

In totaal is er 64 Kb aan geheugen, wat overeen komt met 65.536 geheugenlocaties van één byte. Een Kb (afkorting van kilobyte) is gelijk aan 1024 bytes. Deze vreemde kilo is geen knieval voor het bekende "mag het ietsje meer zijn" maar hangt, zoals meteen zal blijken, samen met het tweetallig stelsel. Alle geheugenlocaties zijn genummerd, in dit voorbeeld van 0 tot 65.535. Om aan de inhoud van een geheugenlocatie te komen, moet het nummer daarvan aan het geheugen worden aangeboden. De Z-80 gebruikt voor het selecteren van geheugenlocaties een nummer van 16 bits, twee bytes dus. Maximaal kan de Z-80 daarmee $2^{16} = 65.536$ verschillende geheugenlocaties uit elkaar houden, lopend van 0 tot 65.535. De PC (program counter) uit het begin van dit hoofdstuk is een zestienbits-teller. De microprocessor kan elk bit ervan via de adresbus verbinden met het geheugen. De adresbus is een verbinding van 16 lijnen. Na het aanzetten van de

computer staat de PC op nul. De zestienbits-inhoud ervan is: 0000000000000000. Een Kb is het aantal te selecteren adressen met een adresbus van 10 lijnen: $2^{10} = 1024$.

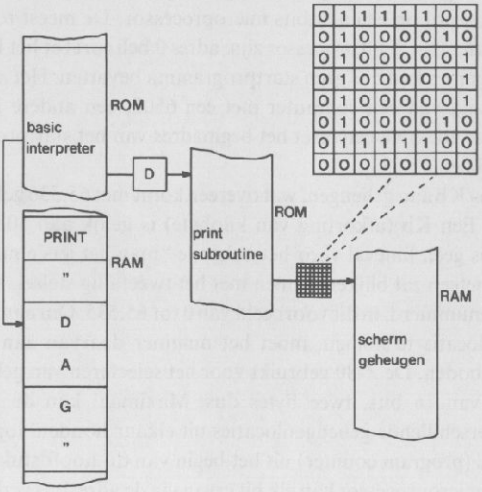
65.536 bytes is dus 65.536/1024 = 64 Kb.

De memory map links in afb. 1.4 laat zien dat de laagste 32 Kb (32.768 bytes) van het geheugen ROM is. Blijft nog eens 32 Kb aan RAM over. Deze hoeveelheid is niet volledig beschikbaar voor BASIC. Een deel ervan moet gereserveerd blijven voor bijvoorbeeld tussentijdse resultaten van berekeningen, systeemvariabelen en niet te vergeten de inhoud van het beeldscherm.

Al kan de Z-80 64 Kb aan geheugenlocaties selecteren, het is niet nodig dat deze 64 Kb er ook daadwerkelijk is. Evengoed kan een computer 16 Kb ROM en 16 Kb RAM hebben of om het even welke hoeveelheden, tot een maximum van 64 Kb.

1.6 Het aanzetten van de computer

Na inschakeling doorloopt de computer een startprogramma in de ROM dat begint op geheugenlocatie 0. Zichtbaar resultaat hiervan is in de meeste gevallen een merknaam, de taal, bijvoorbeeld BASIC en het aantal beschikbare bytes in



Afb. 1.5 Afwerking van de BASIC-opdracht PRINT "DAG"

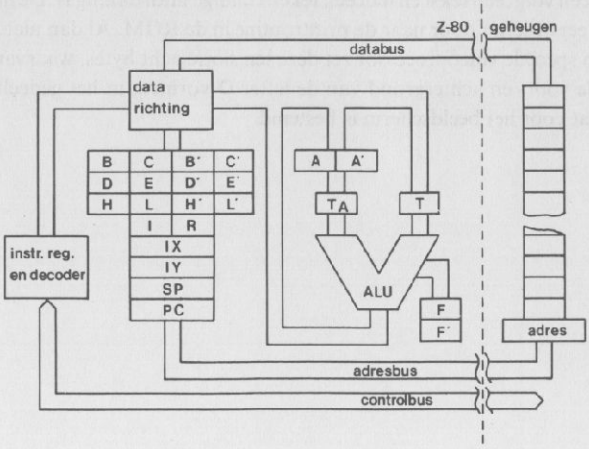
RAM, vermeld op het beeldscherm. Vervolgens begint de computer aan een programma dat op het toetsenbord ingetikte commando's op het beeldscherm zet en uitvoert en dat ingetikte BASIC-regels eveneens op het beeldscherm en bovendien in RAM zet. Een programma dus waarmee het mogelijk is een BASIC-programma te ontwikkelen. Na het commando RUN start het in de ROM aanwezige BASIC-interpreterprogramma. En al lijkt het alsof er niets anders gebeurt dan het uitvoeren van het BASIC-programma, de interpreter wordt regelmatig, bijv. iedere 10 milliseconden, onderbroken. De computer doorloopt dan een zeer kort programma in ROM om zaken van huishoudelijke aard te controleren zoals bijvoorbeeld nagaan of er een toets is ingedrukt.
Afb. 1.5 laat een moment zien waarop de interpreter, bezig met een BASIC-programma, de opdracht PRINT "DAG" aan het verwerken is. Wat betreft het woord PRINT, in de meeste microcomputers zullen niet de ASCII-waarden voor de letters P-R-I-N-T in achtereenvolgende adressen staan; de opdracht PRINT staat dan in de vorm van een code van één byte in een enkel adres. Bij sommige computers moeten gebruikers BASIC-woorden met een enkele toets intypen, de zogenaamde single-key entry. Andere computers eisen het voluit typen van BASIC-woorden In dit laatste geval wordt, om geheugenruimte te sparen, na het intypen het woord toch omgezet naar een code van één byte, token geheten. Bovendien hoeft de interpreter de BASIC-woorden nu niet letter voor letter af te tasten en kan dus sneller werken.
Na het tegenkomen van de aanhalingstekens achter PRINT weet de interpreter dat hetgeen volgt een tekst en niet een rekenkundige uitdrukking is. De interpreter geeft de eerste letter door naar de printroutine in de ROM. Al dan niet geholpen door een speciale videoprocessor zet deze ten slotte acht bytes, waarvan enen en nullen de voor- en achtergrond van de letter D vormen, in het gedeelte van de RAM dat voor het beeldscherm is bestemd.

2 Het programmeren van de microprocessor

2.1 De opbouw van de Z-80

Afb. 2.1 is een schematische weergave van de Z-80 microprocessor. De rechthoeken met de letters A, B, C, D, E, H en L stellen achtbits-registers voor. Net als in de eerder besproken geheugenlocaties kan daarin een achtbits-getal staan: van 00000000 tot en met 11111111. Dat komt in ons tientallig stelsel overeen met getallen van 0 tot en met 255. Weliswaar zijn er $2^8 = 256$ combinaties mogelijk maar 0 is ook een getal. De Z-80 heeft een dubbele set van deze registers: A', B' enz.

Om een microprocessor te kunnen programmeren, is voor elk type een taal ontwikkeld, de zogenaamde assembleertaal. De opdracht om register A te laden met getal 3 luidt in de assembleertaal van de Z-80: LD A,3. LD is een afkorting van LoaD (laad). In assembleertaal zijn opdrachten afkortingen van hetgeen ze



Afb. 2.1 Plattegrond van de Z-80 microprocessor

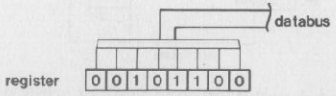
bewerkstelligen of doen daar sterk aan denken. Daarom worden ze mnemonics (geheugensteuntjes) genoemd.
Dezelfde laadopdracht werkt ook voor de andere registers: LD H,126 en LD E,210 laden respectievelijk het H-register met het getal 126 en het E-register met 210. Binair hebben deze registers dan de volgende inhoud:

A-register	00000101
H-register	01111110
E-register	11010010

Ook kunnen registers elkaars inhoud kopiëren. LD B,H kopieert de inhoud van H in B. Zowel in B als in H staat nu het getal 126. Bij zulke instructies is het eerstgenoemde register altijd de bestemming en het tweede de bron. Na LD B,H is de inhoud van de registers B en H:

H-register	01111110
B-register	01111110

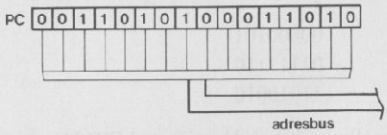
Het data-richtingsregister regelt het verkeer over de interne databus. Een uit het geheugen opgehaalde instructie gaat naar het instructieregister, data bijvoorbeeld naar accumulator A of register B. Het omgekeerde kan ook het geval zijn: de inhoud van een register gaat via de externe databusaansluiting naar het geheugen. Data kan, zoals juist is gezegd, ook van het ene register in het andere worden gekopieerd. In al deze gevallen legt het datarichtingsregister een elektrische verbinding tussen de acht bits van het register en de uit acht leidingen bestaande databus. Zie afb. 2.2. Een 1 of 0 in één van de bits komt overeen met wel of geen spanning.



Afb. 2.2 achtlijns-verbinding tussen register en databus

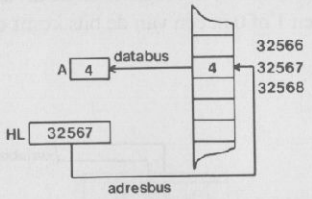
Instructies komen terecht in het instructieregister. De decoder wekt alle elektrische signalen op die nodig zijn om de instructie uit te voeren, zoals bijvoorbeeld de signalen voor de controlbus en die voor het besturen van het datarichtingsregister. De functie van het zestienbits-register PC is al ter sprake gekomen. Evenals de PC zijn ook de registers SP, IX en IY 16 bits groot. Voor de PC was deze grootte nodig

om een geheugenlocatie uit een totaal van 65536 te kunnen selecteren. Bij de andere drie genoemde registers dient de grootte van 16 bits hetzelfde doel. Van het register dat een adres selecteert, zijn de afzonderlijke bits verbonden met de 16 lijnen van de adresbus. Zie afb. 2.3.



Afb. 2.3 zestienlijns-verbinding tussen PC en adresbus

De achtbits-registers B en C, D en E, H en L laten zich aaneengekoppeld gebruiken als de zestienbits-registerparen BC, DE en HL. De inhoud hiervan kan, net als die van de PC, via de adresbus een adres selecteren. Voorbeeld van een instructie waarin dat gebeurt is LD A,(HL). De inhoud van HL is een zestienbits-getal. De Z-80 zet het getal op de 16 bits brede, ofwel 16 verbindingen tellende adresbus naar het geheugen (zie afb. 2.4). In ruil daarvoor zet het geheugen de inhoud van de met het getal op de adresbus overeenkomende geheugenlocatie op de databus. De Z-80 tenslotte verbindt de databus met register A. Net als bij LD B,H verandert de inhoud van de bron, de geheugenlocatie, niet.



Afb. 2.4 Laad A met de inhoud van het adres dat in HL staat

Het verplaatsen van getallen tussen registers en geheugen en registers onderling is op zich niet voldoende om een computer iets te laten doen. Er moet ook nog iets met de verplaatste getallen gebeuren. Daarvoor zorgt de V-vormige figuur in de tekening van de Z-80 processor, de ALU. Deze naam staat voor Arithmetic Logical Unit: de rekenkundige en logische eenheid waarin bewerkingen als optellen, aftrekken en vergelijken plaatsvinden. Direct naast de ALU in afb. 2.1 staat het vlagregister. Zes van de acht bits daarvan geven informatie omtrent de laatste

bewerking in de ALU. Bit 6 bijvoorbeeld, de zero-vlag (nulvlag) heeft de waarde 1 als de uitkomst van de laatste berekening in de ALU gelijk was aan nul. In het andere geval is de zero-vlag 0.
Om erachter te komen wat bit 6 is, moet men van rechts naar links tellen en bij 0 beginnen. Op deze manier krijgt elk bit het nummer waartoe het grondtal 2 verheven moet worden om de decimale waarde die dat bit voorstelt te krijgen.

inhoud byte:	1 0 0 1 0 1 0 0
bitnummers:	7 6 5 4 3 2 1 0
waarde:	$2^7+2^4+2^2=148$

Een aantal zaken aangaande het inwendige van de Z-80 is niet of summier aan de orde gekomen. De meeste laten zich pas goed doorgronden in het gebruik. Andere kunnen slechts tot hun recht komen als er meer omtrent machinetaal bekend is dan in het voorafgaande werd behandeld. Wat niets anders wil zeggen dan dat ze in de loop van de volgende hoofdstukken aan bod zullen komen.
Aangezien dit boek het programmeren in assembler op een bestaande computer behandelt, wordt niet of nauwelijks ingegaan op hardware aangelegenheden zoals de refresh van dynamische RAM of de afhandeling van interrupts. De registers I en R vervullen hiervoor speciale taken. Niet alleen is hiervoor nogal wat technische kennis vereist maar ook zijn de verschillen tussen computers onderling groot.

2.2 De vorm van het machinetaalprogramma

Geheugenlocaties bevatten niet meer dan een achtbits-getal. Het ligt daarom voor de hand dat machinetaalinstructies als LD (laad) en ADD (tel op) dezelfde vorm hebben. Het volgende voorbeeld is een simpel programma in assembleertaal. Ernaast staat het door de assembler in machinecode vertaalde programma bestaande uit getallen van 0 tot en met 255, eerst in de tweetallige en dan in de vertrouwde tientallige vorm. Het is deze reeks getallen die in opeenvolgende locaties van het geheugen staat. De spaties middenin de binaire getallen staan er alleen voor de overzichtelijkheid.

LD A,3	;laad register A met 3	0011 1110	62
		0000 0011	3
LD B,4	;laad register B met 4	0000 0110	6
		0000 0100	4
ADD A,B	;tel A en B op	1000 0000	128

De instructie ADD A,B (ADD is Engels voor optellen) telt de inhoud van registers

A en B bij elkaar op en zet het resultaat in register A. Daarvan is de inhoud na uitvoering van het programma niet langer 3 maar 7.
Een assembler biedt de gelegenheid een programma te schrijven in assembleertaal en het eventueel van commentaar te voorzien.

```
LD A,3     ;laad A met 3
LD B,4     ;laad B met 4
ADD A,B    ;tel A en B op
```

Wat een BASIC-programmeur allereerst opvalt, is het ontbreken van regelnummers. Het gebruik daarvan is echter typisch iets voor BASIC. Programma's in bijvoorbeeld C of Pascal doen het zonder.
Is een programma in assembleertaal gereed dan kan de assembler het vertalen. De assembler zet elke instructie om in het juiste getal en plaatst eventuele data als 3 en 4 er in de juiste volgorde bij. Het commentaar, mits voorafgegaan door het juiste teken, hier een puntkomma, maar dat kan per assembler verschillen, wordt overgeslagen. Het vertaalde programma ziet er als volgt uit:

geheugenlocatie X	62
geheugenlocatie X+1	3
geheugenlocatie X+2	6
geheugenlocatie X+3	4
geheugenlocatie X+4	128

De vraag is waar ergens in RAM het programma begint, ofwel wat de waarde van X is. Er zijn assemblers waarbij een assembler deel uitmaakt van de BASIC interpreter. De assembler plaatst het vertaalde programma dan bijvoorbeeld in een door een DIM-statement gereserveerd stukje geheugen. Veel assemblers zetten de machinetaal in een REM-statement aan het begin van het RAM-geheugen. Uiteindelijk doel is praktisch altijd de machinetaal in combinatie met BASIC te gebruiken. Een vaak aangeroepen subroutine van BASIC omzetten naar machinetaal versnelt een programma soms aanzienlijk. Aan de andere kant is het meestal onzinnig alles in machinetaal te willen doen. Een regel tekst naar het beeldscherm sturen is met een enkele PRINT-opdracht gebeurd. Het schrijven van een machinetaalprogramma dat hetzelfde doet kost heel wat meer tijd. Het is de vraag of dat in alle gevallen opweegt tegen de grotere snelheid waarmee de regel tekst op het scherm verschijnt.
Het is mogelijk zonder assembler in machinetaal te schrijven. De programmeur zet dan met de hand het assembleertaalprogramma om in de machinecode en plaatst de serie getallen, (62,3,6,4 en 128 uit bovenstaand voorbeeld) met POKE statements in een REM-statement in het geheugen.

Het REM-statement moet de eerste regel van het programma zijn aangezien latere regels door wijzigingen in het programma van plaats kunnen veranderen. Het voorbeeldprogramma bestaat uit 5 getallen en zou in een BASIC REM-statement van 5 tekens kunnen:

```
1 REM ABCDE
```

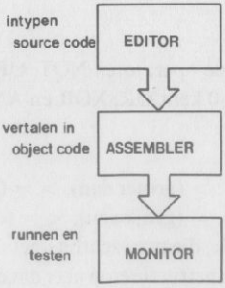
Is de geheugenlocatie waarin de A staat bekend, bijv. nummer 16387, dan zetten de volgende commando's het machinecode programma op de goede plaats:

```
POKE 16387,62
POKE 16388,3
POKE 16389,6
POKE 16390,4
POKE 16391,128
```

De getallen 62, 3 enz. overschrijven daarbij de ASCII-code voor de letters A, B enz. die oorspronkelijk in de geheugenlocaties stonden.
Op deze manier machinecodeprogramma's maken, is tijdrovend en het intypen van reeksen nummers is niet alleen vervelend, ook de kans op het maken van fouten is groot. Een assembler doet het werk probleemloos en is in staat bepaalde fouten te detecteren. Bovendien heeft een assembler faciliteiten die het programmeren in machinetaal vergemakkelijken.

2.3 De assembler

Een losse assembler, d.w.z. één die niet deel uitmaakt van de BASIC-interpreter, bestaat in elk geval uit een editor, de eigenlijke assembler en een monitor.
Met behulp van de editor, zie afb. 2.5, kan het programma in assembleertaal



Afb. 2.5 Onderdelen van het assemblerpakket

worden ingetypt. Meestal biedt de editor zulke voorzieningen als het wissen van regels, of het gemakkelijk veranderen en invoegen daarvan. De editor is niets anders dan een simpele tekstverwerker. Bij ontbreken ervan in een assemblerpakket kan vaak een gewone tekstverwerker worden gebruikt.
Het in assembleertaal geschreven programma heet de sourcecode (source = bron). Zoals gezegd vertaalt de assembler het sourcecodeprogramma in getallen: de object code.
Met een monitor kan men het programma uitvoeren maar ook de inhoud van geheugenlocaties bekijken, wat van pas komt bij het onvermijdelijke foutzoeken, het debuggen (ontluizen) van het programma.

2.4 Typen CPU-instructies

In boeken en tijdschriften duidt men een microprocessor soms aan met CPU, afkorting van Central Processing Unit. De vertaling hiervan is Centrale Verwerkings Eenheid. In Nederlandse lectuur ziet men wel eens de afkorting hiervan: CVE.
De mnemonics, de woorden van de assembleertaal, zijn voor iedere microprocessor anders. Maar in het licht van hetgeen ze bewerkstelligen is de volgende indeling vrij algemeen geldig.

Rekenkundige instructies
Het aantal hiervan is veel kleiner dan in BASIC, waar behalve optellen en aftrekken onder andere vermenigvuldigen, delen, worteltrekken, logaritmische en goniometrische bewerkingen mogelijk zijn. De Z-80 komt, net als de meeste huidige processors niet verder dan optellen, aftrekken en het vergroten of verkleinen van een achtbits-getal met 1.

Logische instructies
Hieronder vallen in BASIC de operatoren NOT, OR, XOR (ook wel met EOR aangeduid) en AND. De Z-80 kent OR, XOR en AND.

Voorwaardelijke instructies
In het algemeen kent BASIC: > (groter dan), > = (groter of gelijk), < (kleiner dan), < = (kleiner of gelijk), = (gelijk aan), < > (ongelijk aan). De Z-80 heeft een voorwaardelijke instructie, die twee achtbits-getallen met elkaar vergelijkt. In feite komt uitvoering van die instructie erop neer dat de Z-80 die twee getallen van elkaar aftrekt. De inhoud van het vlagregister wijst dan uit of de getallen gelijk

waren (in welk geval de uitkomst 0 is en de zero-vlag 1) of niet en welke in dat laatste geval groter was.

Dataverplaatsende instructies
Iets waarmee men bij het werken in BASIC meestal niets te maken heeft. Een voorbeeld is de al behandelde laadinstructie LD. Deze verplaatst data van het geheugen naar een register in de Z-80 en andersom of van het ene Z-80 register naar het andere. Ook is het mogelijk grote hoeveelheden bytes in het geheugen van plaats te laten veranderen.
Verplaatsing van data heeft niet alleen betrekking op processor en geheugen. Bij het maken van een BASIC-programma bijvoorbeeld leest de Z-80 een achtbits-getal van het toetsenbord, de ASCII-code voor een letter, cijfer of leesteken waarvan de toets is ingedrukt en zet dat zowel in het geheugen als op het beeldscherm. Voor dit soort acties dienen input- en output-instructies.

Spronginstructies
In BASIC-programma's komen instructie als GOTO en GOSUB veelvuldig voor. Ze doorbreken het regel na regel afwerken van het programma met een sprong. Machinecodeprogramma's doorlopen dankzij de PC achtereenvolgende geheugenlocaties. Ook hier veranderen spronginstructies het sequentiële verloop. GOTO is te vergelijken met de Z-80 mnemonic JUMP (Engels voor sprong) en GOSUB, het aanroepen van een subroutine, met CALL. En zoals BASIC terugkeert van een subroutine met RETURN doet een Z-80 machinetaalprogramma dat na een RET-instructie.
Wat er precies gebeurt door zo'n instructie is in machinetaal veel gemakkelijker in te zien dan in BASIC. Spronginstructies in machinetaal veranderen de inhoud van de PC. Het simpele optelprogramma begon in geheugenlocatie 16387 en eindigde in 16391. Moet de Z-80 daarna verder gaan met een programmaonderdeel dat begint in geheugenlocatie 19124 dan luidt de spronginstructie JUMP 19124. De Z-80 laadt het getal 19124 in de PC. Het eerstvolgende byte dat de Z-80 nu uit het geheugen haalt, de inhoud van geheugenlocatie 19124, wordt in het instructie-register gezet. Het programma wordt vanaf dat punt voortgezet.

Schuiven, roteren en bitmanipulatie
Bitmanipulatie-opdrachten bieden de gelegenheid een afzonderlijk bit in een achtbits-register of -geheugenlocatie te testen, met andere woorden te kijken of de waarde 1 dan wel 0 is, of te veranderen, dus 1 of 0 te maken.
Schuif- of roteeropdrachten verplaatsen alle bits in een byte één plaats naar links of naar rechts. In onderstaand voorbeeld vindt een verschuiving naar links plaats.

ervoor	0011	1010	= 58
erna	0111	0100	= 116

De verschuiving naar links komt rekenkundig overeen met vermenigvuldiging met 2. Die naar rechts met deling door twee. Er zijn nogal wat manieren om te verschuiven of te roteren. De verschillen komen tot uiting in hetgeen er gebeurt met het bit dat er door de verschuiving of rotatie uitvalt (bit 7 in het voorbeeld) en met wat er op de vrijkomende plaats (bit 0 in het voorbeeld) komt.

Controle-instructies
Deze instructies hebben betrekking op de hardware-organisatie.

2.5 Hexadecimale getallen

Bij het programmeren in machinetaal is een monitorprogramma een nuttig hulpmiddel. Een monitor produceert lijsten met nummers van geheugenlocaties en daarbij hun inhoud. Werkend in machinetaal is het heel plezierig moeiteloos te kunnen zien wat in welke geheugenlocatie staat.
Een losse assembler bevat vaak een monitor die de inhoud van het geheugen in hexadecimale getallen geeft. In publicaties van en over machinetaalprogramma's staan getallen vaak in hexadecimale notatie. Hoewel deze vorm nooit zo vertrouwd wordt als de gewone decimale is het toch van belang er enigszins mee te kunnen werken.
Het hexadecimale getallenstelsel werkt met het grondtal 16. Zoals bij de uitleg van het binaire stelsel ter sprake kwam, in het gewone decimale stelsel tellen we van 0 tot en met 9 en gaan dan verder met 10. In het binaire stelsel, met als grondtal twee, gebeurde dat van 0 tot en met 1 waarop dan 10 volgde. In analogie daarmee moet in het hexadecimale stelsel eerst op de een of andere manier tot vijftien geteld kunnen worden voordat naast het eerste cijfer een 1 komt. En dat gaat als volgt:

decimaal:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15...16
hexadecimaal:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F...10

Het getal 12 in het decimale stelsel betekent:

$2 \times 10^0 = 2$
$1 \times 10^1 = 10$

Hetzelfde getal in het hexadecimale stelsel:

$2 \times 16^0 = 2$
$1 \times 16^1 = 16$

Wat neerkomt op 18. In het hexadecimale stelsel komen getallen voor als 1A ($16+10=26$) of CD ($12\times16+13=205$). Hoewel het in het begin raar lijkt om bijvoorbeeld FA als getal te zien, went het in de praktijk vrij snel.
De keuze om hexadecimale getallen in plaats van decimale te gebruiken, werd gemaakt om een bepaalde reden. Werken met binaire getallen is, door de lange rijen enen en nullen, nogal lastig. Ze omzetten naar decimale getallen neemt dat probleem weg maar van een dergelijk getal is niet eenvoudig in te zien welk bit in het oorspronkelijke binaire nummer nul was en welk één. Bij hexadecimale getallen is dat simpel. Een hexadecimaal cijfer kan een waarde hebben van 0 t/m 15. Hetzelfde geldt voor een groepje van 4 bits.

binair	hexadecimaal
0000	0
0001	1
0010	2
:	:
1110	E
1111	F

Eerder verdeelden we een achtbits-getal in twee groepen van 4. Elke groep laat zich omzetten in een hexadecimaal cijfer en omgekeerd. Na enige oefening is dat makkelijk. Van het binaire getal 0011 1110 bijvoorbeeld heeft de eerste groep de hexadecimale waarde 3 en de tweede groep de hexadecimale waarde E. De waarde van het binaire getal 0011 1110 komt overeen met het hexadecimale getal 3E.
Dat het rekenkundig klopt, blijkt uit het volgende. Een binair getal heeft de waarde:

$2^7\times$ bit 7 $+2^6\times$ bit 6 $+2^5\times$ bit 5 $+2^4\times$ bit $4+2^3\times$ bit 3 $+2^2\times$ bit 2 $+$ $2^1\times$ bit 1 $+2^0\times$ bit 0

Waarin bit 0 t/m bit 7 de waarden 0 of 1 hebben. De uitdrukking kan ook als volgt worden geschreven:

$2^4\times(2^3\times$ bit 7 $+2^2\times$ bit 6 $+2^1\times$ bit 5 $+2^0\times$ bit $4)+$ $(2^3\times$ bit 3 $+2^2\times$ bit 2 $+2^1\times$ bit 1 $+2^0\times$ bit 0)

Beide groepen van 4 bits kunnen nu een waarde hebben van 0 t/m 15 en omgezet worden in een hexadecimaal cijfer.

$2^4\times$ (waarde groep 1) $+$ (waarde groep 2) $=$

$16\times$ (hexadecimaal cijfer) $+$ (hexadecimaal cijfer)

En dit is ook exact de waarde die een hexadecimaal getal van 2 cijfers heeft:
$16\times$ (hoogste cijfer) $+$ laagste cijfer

Een telvoorbeeld ten slotte illustreert het omzetten nog eens:

binair	hexadecimaal
0000 0000	00
0000 0001	01
0000 0010	02
0000 0011	03
0000 1001	09
0000 1010	0A
0000 1011	0B
0000 1110	0E
0000 1111	0F
0001 0000	10
0001 0001	11
1111 1110	FE
1111 1111	FF

De maximumwaarde van een byte is FF ofwel $15 \times 16^1 + 15 \times 16^0 = 255$.
Het omzetten van een zestienbits-getal, zoals het nummer van een geheugenlocatie, naar hexadecimaal gaat op dezelfde manier:

```
1111 1110 0001 1010 = FE1A
```

De Z-80 kan maximaal 65536 geheugenlocaties selecteren met nummers van 0 tot en met 65535. De binaire waarde van de PC loopt daarbij van: 0000 0000 0000 0000 tot en met 1111 1111 1111 1111.
Hexadecimaal is dat van 0000 tot en met FFFF.

Het naast elkaar voorkomen van verschillende talstelsels leidt tot verwarring. Het getal 11 is mogelijk een binaire voorstelling van 3, een hexadecimale voorstelling van 17 of gewoon het decimale getal 11. Onderscheid wordt gewoonlijk gemaakt door het getal een teken mee te geven dat aanduidt om welk stelsel het gaat. Het meest simpele is het toevoegen van een H voor hexadecimaal en B voor binair en alles zonder teken als decimaal te beschouwen. De eerder genoemde mogelijkheden van getal 11 zijn nu te onderscheiden: 11B, 11H en 11.
De mogelijkheid verschillende getallenstelsels in één programma te gebruiken en de manier om ze te onderscheiden verschillen per assembler. Raadpleeg hiervoor de bij uw assembler geleverde documentatie.

3 Optellen en aftrekken

3.1 De achtbits-optelling

```
510 LET TERM1=3
520 LET TERM2=4
530 LET SOM=TERM1+TERM2
540 RETURN
```

De bovenstaande simpele BASIC-subroutine voert een optelling uit, daarbij gebruik makend van drie variabelen. We zullen eerst proberen iets dergelijks te doen in machinecode.
In het vorige hoofdstuk kwam een eenvoudig optelprogramma voor. Ingetypt met de editor zou het er, in de vorm van een subroutine, als volgt uit kunnen zien:

Programma H3P1A Subroutine achtbits-optelling

```
ORG     8000H

LD      A,3     ;laadt register A met 3
LD      B,4     ;laadt register B met 4
ADD     A,B     ;tel inhoud A en B op
RET             ;keer terug

END
```

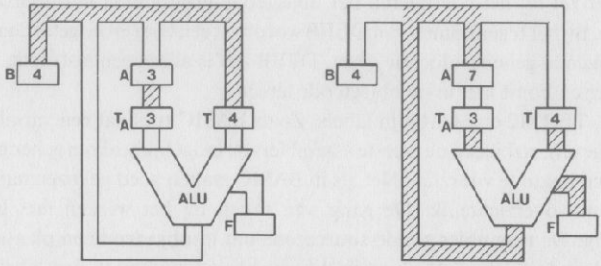
Het programma telt de inhoud van register A op bij die van register B. De instructie ADD zet het resultaat van de optelling in A. Bij alle achtbits rekenkundige instructies van de Z-80 is register A zowel bron, d.w.z één van de twee getallen staat in A, als bestemming van het eindresultaat.
De meeste assemblers geven na het assembleren een overzicht als het volgende:

Programma H3P1 Assembler-listing achtbits-optelling

```
                        ORG     8000H

8000'  3E 03            LD      A,3     ;laadt register A met 3
8002'  06 04            LD      B,4     ;laadt register B met 4
8004'  80               ADD     A,B     ;tel inhoud A en B op
8005'  C9               RET             ;keer terug

                        END
```

De eerste kolom geeft, hexadecimaal, de adressen waarin de code geplaatst is en de tweede de object code. Meestal zet de assembler de code uit zichzelf op de meest geschikte plaats. Soms moet of mag de plaats waar de code komt, worden aangewezen. Hiervoor dient dan het commando ORG *nn* waarin *nn* het adres is waarop de code moet beginnen. ORG hoort niet tot de assembleertaal maar tot de assembler-directives (instructies), commando's waarmee men de assembler opdrachten geeft. Ze vergemakkelijken het schrijven van een programma aanzienlijk. Een ander assembler-directive is END. Het geeft de assembler te kennen dat de sourcecode is afgelopen.
De eerste kolom geeft niet alle adressen maar alleen die waarop een in object code vertaalde instructie begint. De instructie LD B,4 neemt twee geheugenplaatsen in beslag, één voor de eigenlijke instructie en één voor het getal dat in register B moet komen te staan. In de eerste, 8002H, staat de code om het B-register te laden met het getal in de geheugenlocatie die meteen hierna komt: 06H. In de tweede, 8003H, staat dit getal: 4.
Wil een subroutine in machinetaal vanuit BASIC aan te roepen zijn, dan moet de assembler de mogelijkheid hebben de machinecode in een door een BASIC-opdracht als DIM of REM vrijgemaakt stuk geheugen te zetten. Kan dat niet dan moet BASIC of het besturingssysteem de gelegenheid bieden om een deel van het geheugen ontoegankelijk te maken voor BASIC. In geen geval mag de machinecode zomaar ergens neergezet worden omdat BASIC haar dan overschrijft met een programma of variabelen.
Aanroep van de subroutine vanuit BASIC gebeurt met een CALL- of USR(USeR)-functie. In beide gevallen moet in de een of andere vorm het startadres van de subroutine worden meegegeven, bijv. USR (8000H).
Waar een mix van BASIC en machinetaal is toegestaan, is het vaak mogelijk het resultaat van het machinetaalprogramma, zoals de uitkomst van onze simpele berekening, terug te geven aan het BASIC-programma.
Een onafhankelijk machinetaalprogramma kan, in de testfase, worden gestart vanuit de monitor/debugger. Met RET of een eventueel ander commando springt men terug in de monitor en kan de inhoud van adressen en registers worden bekeken.
In alle gevallen betekent aanroep van de subroutine dat het startadres ervan in de Program Counter komt. Hierna plaatst de Z-80 de inhoud van de PC op de adresbus, zet de inhoud van geheugenlocatie 8000H, 3EH, in het instructieregister. Allereerst verhoogt de Z-80 de inhoud van de PC met één en voert dan de instructie uit door weer de PC op de adresbus te zetten. De inhoud van adres 8001H gaat nu niet naar het instructieregister maar, dankzij het werk van het data-richtingsregister, naar register A, de accumulator.

Op dezelfde manier gaat de inhoud van adres 8003H naar register B. De optel-instructie ADD A,B zet de inhoud van accumulator A in de tijdelijke accumulator TA en tegelijk die van register B in het tijdelijke register T. Zie afb. 3.1. De reden hiervoor komt dadelijk aan bod. Vervolgens telt de ALU de inhoud van TA en T bij elkaar op en zet ze in de accumulator A. Deze manoeuvre maakt de noodzaak van de tijdelijke registers duidelijk. Zouden ze er niet zijn dan moest de ALU de inhoud van A en B optellen. Aangezien de som meteen weer in A komt, verandert de inhoud van A bij wijze van spreken tijdens het optellen waardoor de som verandert enz.



Afb. 3.1 Optelling van de inhoud van twee registers

Is het programma ten einde dan zouden we, als de monitor de mogelijkheid biedt, eens kunnen kijken naar de inhoud van de registers. In A moet dan het getal 7 staan. Met sommige monitors kan men een programma stap voor stap (single step) doorlopen. Tijdens elke stap voert de Z-80 één instructie uit waarna men het effect ervan op de inhoud van de registers kan bekijken.

3.2 Labels

Een programma als het hiervoor behandelde doet niet meer dan steeds weer 3 en 4 bij elkaar optellen. Om het wat universeler te maken, doen we het volgende.

Programma H3P2A achtbits-optelling met gebruik van labels

```
LD      A,(TERM1)       ;laadt A met eerste term
LD      B,A             ;laadt B met A
LD      A,(TERM2)       ;laadt A met tweede term
ADD     A,B             ;tel A en B op
LD      (SOM),A         ;zet som in SOM
```

```
        RET
TERM1:  DEFB    03
TERM2:  DEFB    04
SOM:    DEFB    00

        END
```

Het ORG-directive is hier niet gebruikt. De assembler waarmee de voor dit boek ontwikkelde programma's zijn geassembleerd, genereert in zo'n geval objectcode die begint op adres 0.
DEFB is een assembler-directive en staat voor define byte (bepaal byte). De assembler zet bij het compileren het objectcode-programma in opeenvolgende adressen. Bij het tegenkomen van DEFB wordt het één byte grote getal daarachter in de volgende geheugenlocatie gezet. DEFB zelf is alleen een instructie voor de assembler en komt niet in de objectcode terecht.
TERM1, TERM2 en SOM zijn labels. Zoals BASIC toestaat een variabele een naam te geven, zo bieden de meeste assemblers de mogelijkheid een geheugenlocatie van een naam te voorzien. Net als in BASIC maken goed gekozen namen een programma overzichtelijk. De gang van zaken bij het werken met labels is eenvoudig. De assembler zet de sourcecode om in objectcode en plaatst die in opeenvolgende geheugenlocaties, in dit geval beginnend bij 0. Een label aan de linkerkant van de sourcecode krijgt als waarde het geheugenadres waar de assembler de eerstvolgende objectcode zal opbergen. Voorwaarde voor het mogen gebruiken van labels is dat de assembler van het "two pass" type is, dus tweemaal langs de sourcecode gaat. Dat dit nodig is, blijkt uit het volgende. De eerste maal dat de assembler TERM1 tegenkomt, is in de instructie LD A,(TERM1). Het adres waarbij TERM1 hoort, ligt dan nog niet vast; dat gebeurt pas na RET als TERM1 links van de sourcecode staat. Wanneer de assembler voor de tweede maal langs de sourcecode gaat, is het adres van TERM1 wel bekend en wordt op de juiste plaats in de objectcode gezet.

Programma H3P2 Assembler-listing van de achtbits-optelling

```
0000'  3A 000C'          LD      A,(TERM1)   ;laadt A met eerste term
0003'  47                LD      B,A                 ;laadt B met A
0004'  3A 000D'          LD      A,(TERM2)   ;laadt A met tweede term
0007'  80                ADD     A,B                 ;tel A en B op
0008'  32 000E'          LD      (SOM),A             ;zet som in SOM
000B'  C9                RET
000C'  03        TERM1:  DEFB    03
000D'  04        TERM2:  DEFB    04
000E'  00        SOM:    DEFB    00

                         END
```

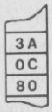
Van het geassembleerde programma hierboven bekijken we eerst het effect van DEFB. De drie bytes na de laatste sourcecode-instructie, RET, zijn gevuld met de waarden na de opeenvolgende DEFB's en wel 03, 04 en 00.
Het adres van het label TERM1 is blijkbaar 000CH. Hetzelfde adres is te vinden in de eerste instructie LD A,(TERM1). 3A is de code voor de instructie en daarachter staat het adres waar de waarde van term 1 te vinden is, namelijk 000CH. De assembler heeft voor TERM1 de juiste waarde ingevuld: LD A,(000CH).
De hier op adres 0 beginnende code wordt niet in het geheugen gezet. Op adres 0 begint namelijk bij Z-80-systemen een opstartprogramma. De hier gebruikte assembler haalt de sourcecode van schijf en schrijft de gemaakte objectcode weer naar schijf weg. Het is dus niet mogelijk de objectcode te runnen. Deze vreemd aandoende gang van zaken heeft de volgende reden. Programma's voor dit boek zijn ontwikkeld op een computer met een CP/M-achtig besturingssysteem. Daarin beginnen onafhankelijk draaiende programma's, geassembleerde machinetaal of gecompileerde hogere taal, op adres 100H. Assemblers en compilers voor dit systeem genereren allereerst code beginnend op adres 0. Pas een tweede stap, het zgn. linken, verplaatst de code naar adres 100H. Het heet dan dat de assembler of compiler relocatable (verplaatsbare) code genereert.
Dit omslachtig lijkend proces biedt grote voordelen. De linker kan namelijk niet alleen de code verzetten naar 100H maar naar elk gewenst adres. Dankzij deze mogelijkheid kan men programma's in onderdelen ontwikkelen, bijv. hoofdprogramma en subroutines, waarna de linker alles netjes 'aan elkaar plakt'. Eenmaal ontwikkelde subroutines laten zich ook gebruiken in andere programma's, door ze daarmee te linken. Op deze manier kan men library's opbouwen, bibliotheken van steeds weer te gebruiken subroutines.
We voegen nu het directive ORG 8000H weer toe en bekijken het optelprogramma opnieuw.

Programma H3P3 Listing van het optelprogramma geassembleerd

```
                                ORG     8000H

8000'  3A 800C'                 LD      A,(TERM1)   ;laadt A met eerste term
8003'  47                       LD      B,A                 ;laadt B met A
8004'  3A 800D'                 LD      A,(TERM2)   ;laadt A met tweede term
8007'  80                       ADD     A,B                 ;tel A en B op
8008'  32 800E'                 LD      (SOM),A             ;zet som in SOM
800B'  C9                       RET
800C'  03               TERM1:  DEFB    03
800D'  04               TERM2:  DEFB    04
800E'  00               SOM:    DEFB    00

                                END
```

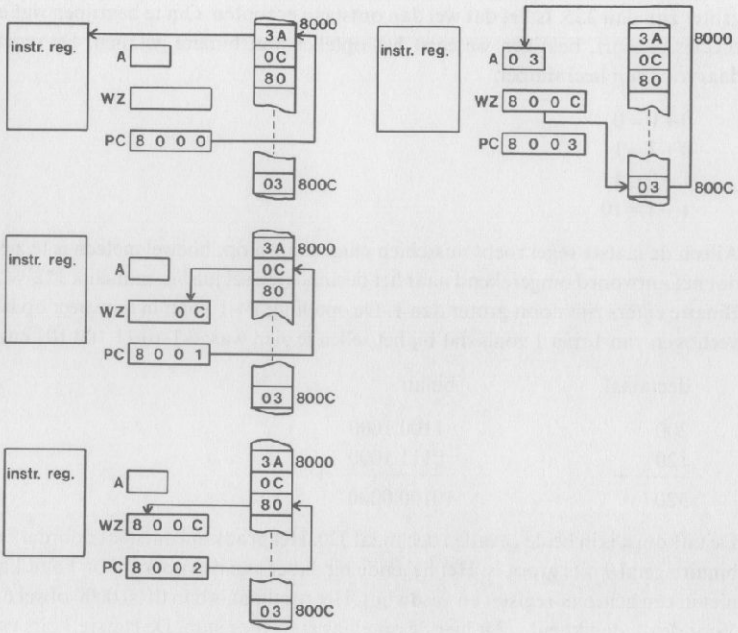
Tot dusver is alleen een laadinstructie als LD A,3 behandeld. Afb. 3.2 laat zien hoe deze instructie in het geheugen staat. Allereerst de code voor de instructie (3E), de zogenaamde operatiecode of opcode die de instructie kenmerkt. Meteen daarna komt de data, het getal dat de Z-80 in A moet laden.
In afb. 3.3 staat de instructie LD A,(800CH). Voor de uitvoering ervan gebruikt de Z-80 een niet voor de programmeur toegankelijk register dat we hier met WZ zullen aanduiden. Net als HL en PC is het een zestienbits-register.

Afb. 3.2 De instructie LD A,3 in het geheugen



Afb. 3.3 De instructie LD A,(800CH) in het geheugen

De opcode van de instructie is 3A. Deze gaat naar het instructieregister, wat het volgende bewerkstelligt. De inhoud van het eerstvolgende adres (0C), zie afb. 3.4, gaat naar register Z, die van het daarop volgende adres naar register W. De Z-80 zet de zestienbits-inhoud van WZ op de adresbus en plaatst de inhoud van deze locatie in A. Aangezien een geheugenlocatie maar acht bits kan bevatten, zijn zestienbits-adressen of -getallen verdeeld in 2 stukken van één byte groot. Bij laden vanuit of opslaan in het geheugen komt het laagste byte, hier 0C, voorop. Dat wil dus zeggen: in het laagste adres.
Het programma laadt de inhoud van TERM1 in A en zet de inhoud van A vervolgens in B. Dat moet op deze manier, want de Z-80 kent geen instructie zoals LD B,(TERM2), dat is alleen met A mogelijk. Beide getallen moeten dus via A binnengehaald worden. Het omgekeerde gebeurt met LD (SOM),A. De Z-80 zet de inhoud van A in de geheugenlocatie met het label SOM. De andere manier om de overige registers te laden met de inhoud van een adres is het adres in HL zetten, gevolgd door de instructie LD *r*, (HL). Hierin is *r* één van de achtbits-registers A, B, C, D, E, H of L.
Met de monitor kan men in de adressen horend bij TERM1 en TERM2 (800CH en 800DH) andere waarden voor getal 1 en 2 zetten. Of plaats anders, indien mogelijk, vanuit BASIC de op te tellen getallen in TERM1 en TERM2 met POKE 32780,4 en POKE 32781,3. (32780 is de decimale waarde van 800CH) Start daarna het programma met USR(32768) en vraag het resultaat op met PRINT PEEK 32782. Dat laatste is, decimaal, het adres van SOM. Bovengenoemde adressen zijn voor het op 8000H beginnende programma. Andere startadressen vereisen natuurlijk aanpassing. Heeft een computer de mogelijkheid het argument



Afb. 3.4 De uitvoering van de instructie LD A,(800CH)

van PEEK of POKE hexadecimaal op te geven dan kan men volstaan met POKE 800C,3 enz.

Dat wat we nu met de hand doen, TERM1 een waarde geven, wordt door hogere programmeertalen als BASIC en Pascal automatisch gedaan. Een BASIC-interpreter slaat de namen van variabelen op in een lijst. Op elke naam volgt het adres waarin de waarde van de variabele staat. Dat in zulke talen de waarde van een variabele groter kan zijn dan 255 komt doordat er voor een getal meer dan één byte wordt gebruikt. In latere hoofdstukken zullen we dieper ingaan op de manier waarop zoiets gebeurt.

3.3 Het gebruik van de carry

Is de waarde van TERM1 bijvoorbeeld 200 en van TERM2 120 dan is de uitkomst niet 320 maar 64. Het getal in een achtbitsgeheugenlocatie kan namelijk niet

groter zijn dan 255. Is het dat wel dan ontstaan er fouten. Om te begrijpen wat er precies gebeurt, bekijken we eerst het optellen van binaire getallen. De regels daarvoor zijn heel simpel:

$0+0=0$
$0+1=1$
$1+0=1$
$1+1=10$

Alleen de laatste regel roept misschien enige vragen op, hoewel meteen is te zien dat het antwoord omgerekend naar het decimale stelsel juist is, namelijk $1*2^1=2$. Binaire cijfers zijn nooit groter dan 1. De optelling $1+1$ komt in feite neer op het verhogen van 1 met 1 zoals dat bij het tellen te zien was: 0 1-10 11 100 101 enz.

decimaal		binair	
200		1100 1000	
120	+	0111 1000	+
320		10100 0000	

De uitkomst is in beide gevallen decimaal 320. Het probleem ontstaat doordat het binaire getal 9 bits groot is. Het negende bit dat er aan de voorkant bij komt kan niet in een achtbits-register en verdwijnt. Het resultaat is dan 0100 0000 ofwel 64. Voor de duidelijkheid volgt hier de optelling stap voor stap. De laatste 3 bits van beide getallen zijn 0. De optelling daarvan ligt voor de hand.

	1100	1
	0111	1
carry	1	
	0000	+

Optelling van de beide bits 4 geeft $1+1=10$. Net als in het decimale stelsel is de verwerking hiervan: 0 opschrijven, 1 onthouden. Deze 1 noemt men naar Engels voorbeeld carry: overdracht. De carry moet bij de beide bits 5 worden opgeteld.

oude carry	1	
	1100	
	0111	
carry	1	
	00000	+

Beide bits 5 plus carry geven weer 10. Er is opnieuw een carry ontstaan.

```
oude carry     1
              110
              011
     carry    1
              ------ +
               00 0000
```

De nu ontstane situatie is nieuw. De beide bits 7 zijn beide 1 en daarbij komt dan nog de carry: 1 + 1 + 1 = 11. 1 opschrijven, 1 onthouden.

```
oude carry     1
              11
              01
     carry    1
              ------ +
             100 0000
```

```
oude carry     1
               1
               0
     carry     1
              ------- +
              0100 0000
```

Deze laatste carry kan niet in het achtbits-getal worden gezet. Met de carry erbij, als negende bit, is de uitkomst correct: 10100 0000 = 320. De inhoud van accumulator A is na de optelling: 0100 0000 = 64.

Er is een manier om deze fout te detecteren. Zodra zich in een optelling een carry naar een negende bit voordoet, set de ALU het carry-bit van het vlagregister F, d.w.z. het carry-bit van het vlagregister wordt 1. Is er geen carry naar een negende bit dan reset de ALU het carry-bit van F; het bit wordt nul.

Het vlagregister F is een achtbits-register. In zes bits daarvan slaat de ALU informatie aangaande de uitkomst van de laatste berekening op. We noemden al het zero-bit. Is de uitkomst van een berekening 0 dan wordt het zero-bit geset, ofwel 1, anders gereset, dus 0.

Om een fout als de behandelde te kunnen detecteren, wijzigen we het optelprogramma tot het volgende.

In het programma staat een nog niet behandelde instructie: JP C FOUT. Het is een afkorting voor JumP on Carry naar FOUT: spring als het carry-bit is geset naar label FOUT.

Programma H3P4 Optelling met foutdetectie

```
                                ORG     8000H

8000'  3A 8016'                 LD      A,(TERM1)   ;term 1 in A
8003'  47                       LD      B,A                     ;term 1 in B
8004'  3A 8017'                 LD      A,(TERM2)   ;term 2 in A
8007'  80                       ADD     A,B                     ;som in A
8008'  32 8018'                 LD      (SOM),A                 ; naar SOM
800B'  3E 01                    LD      A,1                     ;in A 1
800D'  DA 8012'                 JP      C,FOUT                  ;carry door optelling ?
8010'  3E 00                    LD      A,0                     ;nee, in A 0
8012'  32 8019'         FOUT:   LD      (VLAG),A    ;waarde A in VLAG
8015'  C9                       RET

8016'  03               TERM1:  DEFB    03
8017'  04               TERM2:  DEFB    04
8018'  00               SOM:    DEFB    00
8019'  00               VLAG:   DEFB    00

                                END
```

Een gewone jump-instructie is te vergelijken met GOTO in BASIC. Door gebruik van GOTO, gevolgd door een regelnummer wordt het achtereenvolgens van programmaregels doorbroken. Het programma gaat verder bij de opgegeven regel. De jump-instructie, JP, doorbreekt het achtereenvolgens afwerken van adressen. Het effect van JP, gevolgd door een adres of een label is dat de Z-80 het nieuwe adres in de PC zet en vanaf dat punt verder gaat met het programma. Is er, zoals hier, een label gebruikt dan zet de assembler voor het label het juiste adres in de plaats.
Jumps kunnen ook voorwaardelijk zijn. Dat is het geval in het programma. De sprong wordt alleen gemaakt als het carry-bit geset is. De Z-80 laadt dan adres 8012H in de PC.
Is er geen carry dan krijgt A de waarde 1, de sprong wordt niet gemaakt en A krijgt vervolgens de waarde 0. Deze waarde komt in het byte aangeduid met VLAG. Is de carry geset dan krijgt A de waarde 1 en wordt de sprong naar FOUT uitgevoerd. Nu wordt getal 1 in het met VLAG aangeduide byte geladen. Na elke optelling geeft VLAG (adres 8019H) aan of het resultaat geldig is (0) of niet (1).

3.4 Tijd en ruimte

Het uiteindelijke optelprogramma neemt 25 geheugenplaatsen in beslag. De instructies LD A,(TERM1) LD A,(TERM2) enz. gebruiken telkens drie bytes in

het geheugen. Aangezien TERM1, TERM2, SOM en VLAG opeenvolgende adressen hebben en ook opeenvolgend worden gebruikt, zou het economisch zijn het adres van TERM1 in HL te zetten. De inhoud van HL verhogen met één geeft het adres van TERM2, nogmaals verhogen dat van SOM enz. Het verhogen gebeurt met de instructie INC HL. INCrement betekent vergroten.

Programma H3P5 Verbetering optelprogramma eerste fase

```
                              ORG     8000H

8000'  21 8013'               LD      HL,TERM1    ;adres TERM1 in HL
8003'  7E                     LD      A,(HL)              ;TERM1 in A
8004'  23                     INC     HL                  ;adres TERM2 in HL
8005'  46                     LD      B,(HL)              ;TERM2 in B
8006'  80                     ADD     A,B                 ;TERM1 + TERM2 in A
8007'  23                     INC     HL                  ;adres SOM in HL
8008'  77                     LD      (HL),A              ;som naar SOM
8009'  23                     INC     HL                  ;adres VLAG in HL
800A'  3E 01                  LD      A,1                 ;in A 1
800C'  DA 8011'               JP      C,FOUT              ;was er een carry ?
800F'  3E 00                  LD      A,0                 ;nee, A = 0
8011'  77              FOUT:  LD      (HL),A              ;0 of 1 naar VLAG
8012'  C9                    RET

8013'  03              TERM1: DEFB    03
8014'  04              TERM2: DEFB    04
8015'  00              SOM:   DEFB    00
8016'  00              VLAG:  DEFB    00

                              END
```

Het is in feite niet nodig register B te gebruiken. Er bestaat een instructie ADD A,(HL). Deze telt de inhoud van accumulator A op bij die van de geheugenlocatie waarvan het adres in HL staat. Het resultaat komt in A.
Ook de foutroutine is te vereenvoudigen. Als in A eenmaal het getal 1 staat, kan in plaats van het laden met 0 de inhoud van A worden verlaagd met de instructie DEC A. Deze doet het omgekeerde van de INC-instructie. DEC A vermindert de inhoud van A met 1 zodat deze na de instructie 0 is. DECrement betekent verlagen. De instructie LD A,0 gebruikt twee adressen tegen DEC A één.

Programma H3P6 Verbetering optelprogramma tweede fase

```
                              ORG     8000H

8000'  21 8011'               LD      HL,TERM1    ;adres TERM1 in HL
8003'  7E                     LD      A,(HL)              ;TERM1 in A
8004'  23                     INC     HL                  ;adres TERM2 in HL
```

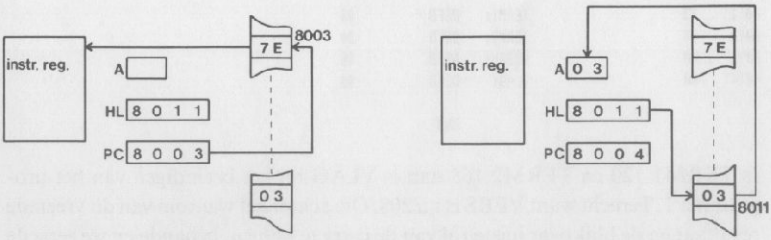
```
8005'  86                    ADD     A,(HL)          ;TERM1 + TERM2 in A
8006'  23                    INC     HL              ;adres SOM in HL
8007'  77                    LD      (HL),A          ;som naar SOM
8008'  23                    INC     HL              ;adres VLAG in HL
8009'  3E 01                 LD      A,1             ;in A 1
800B'  DA 800F'              JP      C,FOUT          ;was er een carry ?
800E'  3D                    DEC     A               ;in A 0
800F'  77            FOUT:   LD      (HL),A          ;0 of 1 naar VLAG
8010'  C9                    RET

8011'  03            TERM1:  DEFB    03
8012'  04            TERM2:  DEFB    04
8013'  00            SOM:    DEFB    00
8014'  00            VLAG:   DEFB    00

                     END
```

Het oorspronkelijke volledige optelprogramma nam 25 bytes in beslag. Dit laatste voorbeeld levert met 20 bytes een besparing op van 20% aan geheugenruimte. Het is mogelijk het programma nog kleiner te maken, maar de daarvoor benodigde kennis komt pas later aan de orde.

Zoals we nu hebben gezien, kan hetzelfde resultaat op verschillende manieren worden bereikt, eenvoudigweg door gebruik te maken van andere instructies. Welke oplossing de voorkeur geniet, hangt af van de eisen die de programmeur stelt. Zulke eisen kunnen velerlei zijn: overzichtelijkheid bijvoorbeeld. Voor wie programma's schrijft waarmee anderen moeten werken, kan dit een rol spelen.

Veel voorkomende criteria zijn die van compactheid en snelheid. In het eerste geval gaat het om programma's die zo min mogelijk geheugenruimte in beslag nemen. In het andere geval is de snelheid van het programma van groot belang. Deze criteria kunnen elkaar uitsluiten. De snelste programma's zijn niet altijd het zuinigst met geheugenruimte en de meest compacte vaak niet het snelst. Daartussen kiezen is een kwestie van afwegen.

Wat de snelheid betreft, daaraan wordt in hoofdstuk 4 uitgebreid aandacht besteed. Hier zij slechts opgemerkt dat iedere instructie een bepaalde tijd in beslag neemt, de één meer dan de ander. Dit heeft voornamelijk te maken met het aantal malen dat het geheugen moet worden aangesproken om de instructie op te halen en uit te voeren. LD A,(TERM1) spreekt het geheugen viermaal aan. Zie afb. 3.4. Eénmaal voor de code die de instructie kenmerkt, de zogenaamde operatiecode, kortweg opcode genoemd (3A). Vervolgens nog driemaal voor de data. Eerst moet het adres van TERM1 naar het speciale WZ-register. Deze 16 bits moeten in twee stukken van acht via de databus naar WZ. Vervolgens komt de inhoud van WZ, nu het adres van TERM1 op de adresbus en gaat TERM1 van het geheugen naar accumulator A.



Afb. 3.5 Uitvoering van de instructie LD A,(HL)

De instructie LD A,(HL) daarentegen spreekt het geheugen maar tweemaal aan. Zie afb 3.5. Eénmaal voor de opcode (7E). Daarna komt de inhoud van HL op de adresbus en gaat TERM1 naar A.
De instructie LD A,(TERM1) kost dan ook bijna tweemaal zoveel tijd als LD A,(HL) namelijk 13 tegen 7 tijdseenheden.

3.5 De achtbits-aftrekking

Het programma voor de aftrekking is identiek met dat voor het optellen met dien verstande dat men in het laatste optelprogramma ADD A,(HL) moet vervangen door SUB (HL). SUB staat voor subtract (aftrekken). Is TERM1 nu 168 en TERM2 120 dan is VRSCH (van verschil dat op de plaats van SOM komt) zoals te verwachten was 48 en VLAG is 0 om aan te geven dat het resultaat geldig is.

Programma H3P7 De achtbits-aftrekking

```
                        ORG     8000H

8000'  21 8011'         LD      HL,TERM1    ;adres TERM1 in HL
8003'  7E               LD      A,(HL)              ;TERM1 in A
8004'  23               INC     HL                  ;adres TERM2 in HL
8005'  96               SUB     (HL)                ;TERM1 - TERM2 in A
8006'  23               INC     HL                  ;adres VRSCH in HL
8007'  77               LD      (HL),A              ;verschil naar VRSCH
8008'  23               INC     HL                  ;adres VLAG in HL
8009'  3E 01            LD      A,1                 ;in A 1
800B'  DA 800F'         JP      C,FOUT              ;was er een carry ?
800E'  3D               DEC     A                   ;in A 0
800F'  77               FOUT:   LD      (HL),A      ;0 of 1 naar VLAG
8010'  C9               RET
```

```
8011'  03          TERM1:  DEFB    03
8012'  04          TERM2:  DEFB    04
8013'  00          VRSCH:  DEFB    00
8014'  00          VLAG:   DEFB    00

                           END
```

Is TERM1 120 en TERM2 168 dan is VLAG bij het beëindigen van het programma 1. Terecht want VERS is nu 208. Om achter het waarom van dit vreemde resultaat en de blijkbaar juiste rol van de carry te komen, behandelen we eerst de regels voor binair aftrekken.

$0-0=0$
$1-0=1$
$1-1=0$

De aftrekking 168-120 verloopt als volgt:

```
1010 1000          168
0111 1000        - 120
-------------
     0000
```

Het cijfer voor cijfer aftrekken van de laagste 4 bits is eenvoudig. Dan echter doet zich de situatie 0-1 voor. De te volgen handelwijze is dezelfde als die voor decimale getallen.

```
63
 7
--  -
```

Aangezien 3 – 7 niet gaat, "leent" men van de 6 een 1. Eerder is al opgemerkt dat van rechts naar links gaand in een getal de betekenis van een cijfer steeds groter wordt. Voor elke gedane stap moet men het met het grondtal van het stelsel vermenigvuldigen. Ten opzichte van 3 heeft 6 de betekenis 60. Het "lenen" brengt ons tot het volgende:

```
  5 (13)
  7
-----  -
  6
```

De handelwijze voor binaire getallen is eender.

```
1010               100(10)
0111             - 011  1       -
-------------      ---------
     0000               1 0000
```

Links is een 1 "geleend". Dit lenen duidt men aan met het Engelse woord voor lenen: borrow. De binaire waarde daarvan, één kolom naar rechts in het getal, is 10. (Hetgeen decimaal neerkomt op 2: de geleende 1 vermenigvuldigd met het grondtal van het stelsel, 2.) Voor het aftrekken van de volgende twee bits is ook een borrow nodig maar in dit geval staat de te lenen 1 twee kolommen verder. We geven het lenen in fasen weer. Eén kolom naar rechts verplaatst, heeft de geleende 1 de waarde 10, ofwel 1 + 1. Van deze twee enen gaat er één naar de kolom rechts en heeft daar de waarde 10.

```
100          0(10)0          01 (10)
011          0 1 1           01  1
-----   -    -----   -       -----   -
 1 0000      1 0000          00  1 1 0000
```

De uitkomst is 0011 0000, decimaal 48.
Het verwisselen van TERM1 en TERM2 geeft in eerste instantie weinig problemen bij het aftrekken.

```
0111 1000 120
1010 1000 168  -
--------------
 101 0000
```

Bij het aftrekken van de laatste twee bits ontstaat een borrow, alleen is het de vraag waar de te lenen 1 vandaan moet komen. Er is namelijk aan de linkerkant geen kolom meer. Dat er een dergelijke borrow is opgetreden, signaleert de Z-80 door de carry-vlag te setten (1 te maken).

```
(10)111 1000      120
 1 010  1000  -    168
---------------------
 1 101  0000      208
```

De carry-vlag is geset:

- Als er bij optellen een carry naar het niet bestaande negende bit had gemoeten en het resultaat dus groter was dan 255.

- Als er bij aftrekken een borrow van het niet bestaande negende bit had moeten komen en het resultaat dus negatief was.

4 Lusstructuren

Wat de mogelijkheid tot het maken van lusstructuren betreft, verschillen BASIC-interpreters nogal van elkaar. Gemeen hebben ze in elk geval de FOR-NEXT-lus. Deze ziet er als volgt uit.

```
100 FOR GETAL = 2 TO 50 STEP 2
110 PRINT GETAL;'' kwadraat = '';GETAL*GETAL
120 NEXT GETAL
```

Het programma geeft de kwadraten van de even getallen in de reeks 2 t/m 50. Aan het begin van de lus staat het aantal herhalingen vast. Dat is niet het geval bij de minder algemeen voorkomende REPEAT-UNTIL-(ook wel DO-UNTIL) en WHILE-WEND-lussen.

```
100 REPEAT (of DO)
110 INPUT ''Wilt u stoppen '';ANTWOORD$
120 UNTIL ANTWOORD$=''JA'' OR ANTWOORD$=''NEE''
```

De opdracht(en) tussen REPEAT en UNTIL worden herhaald totdat aan de voorwaarde na UNTIL is voldaan. Dat kan na één keer zijn, maar ook na dertig keer, als de gebruiker bijv. niet JA of NEE intypt maar iets anders. In elk geval werkt het programma de opdrachten tussen REPEAT (herhaal) en UNTIL (totdat) minstens éénmaal af. Bij de WHILE-WEND-lus is dat niet zo.

```
100 INPUT ''Welk getal '';GE
110 WHILE GE > 0
120 PRINT ''Wortel = '';SQR(GE)
130 INPUT ''Welk getal '';GE
140 WEND
```

Indien de waarde van GE voldoet aan de voorwaarde na WHILE (indien) voert het programma de BASIC-opdrachten tot aan WEND (ga) uit en keert dan terug naar het begin van de lus om de voorwaarde opnieuw te testen. Zodra GE niet aan de voorwaarde voldoet, springt het programma naar de eerste BASIC-opdracht na WEND. Als GE positief is, geeft het bovenstaande programma de vierkantswortel van GE.
Is het allereerst ingevoerde getal negatief of nul dan doorloopt het programma de lus geen enkele maal.

In elk van de drie lussen is er sprake van een test. Bij de FOR-NEXT-lus is dat misschien moeilijk in te zien. Bedenk dat bij FOR TELLER = BEGIN TO EINDE STEP INTERVAL de instructie NEXT TELLER de waarde van TELLER vergroot met de waarde van INTERVAL. Daarna wordt getest of TELLER al groter dan EINDE is. Is dat niet het geval dan doorloopt het programma de lus nog eens.
Voor een wat eenvoudiger BASIC laat een eventueel ontbrekende WHILE-WEND-lus zich namaken met een test en twee GOTO's.

```
100 INPUT ''Welk getal '';GE
110 IF GE <= 0 THEN GOTO 140
120 PRINT ''Wortel = '';SQR(GE)
130 GOTO 100
140 END
```

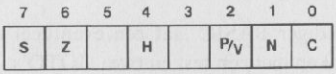
In feite kan men alle lussen imiteren met tests en GOTO's. Ze krijgen dan ongeveer dezelfde vorm als soortgelijke lussen in machinecode.
Het rekenprogramma in het vorige hoofdstuk maakte gebruik van de voorwaardelijke sprong JP C,label. Met behulp hiervan krijgt een WHILE-WEND-achtige lus in machinecode de volgende structuur.

```
WHILE:    LD A,10      ;in A 10
          SUB B        ;trek B af van A
          JP C,WEND    ;is B groter dan A ?
          :            ;nee, doorloop lus
          :
          routine
          :
          :
          JP WHILE     ;naar begin lus
WEND:     RET
```

Het programma laadt register A met 10 en trekt daar vervolgens de inhoud van register B van af. Is deze groter dan 10 dan zal door het aftrekken het carry-bit worden gezet. In dat geval springt het programma meteen naar het einde bij het label WEND. Is B kleiner dan of gelijk aan 10, dan is de carry door SUB B gereset. Het programma doorloopt de routine en keert voor WEND terug naar de test bij WHILE. Om ooit uit de lus te kunnen raken, moet de routine de inhoud van register B veranderen. Wat precies die routine is of wat de inhoud van B voorstelt, doet voor het principe niet ter zake.
We zullen nu eerst onderzoeken welke tests met de Z-80 mogelijk zijn en hoe daarop met voorwaardelijke instructies zoals JP C kan worden gereageerd.

4.1 Het vlagregister

Het vlagregister is acht bits groot. De Z-80 gebruikt er daarvan zes. Hiervan zijn er vier toegankelijk voor de programmeur. Zie afb. 4.1



Afb. 4.1 Het Z-80 vlagregister

C = Carry-vlag
N = Add/Subtract-vlag (optellen/aftrekken)
P/V = Parity/Overflow-vlag (pariteit/overflow)
H = Half-carry-vlag
Z = Zero-vlag (nul)
S = Sign-vlag (teken)

De Add/Subtract- en de Half-carry-vlag worden door de Z-80 zelf gebruikt bij het corrigeren van BCD(Binary Coded Decimal)- getallen in de accumulator na een berekening.
Sign- en Overflow-vlag hebben te maken met tweecomplementsnotatie en komen in hoofdstuk 7 aan de orde.
Parity en Overflow gebruiken hetzelfde bit van het vlagregister. Lang niet alle instructies beïnvloeden de vlaggen en zij die dat wel doen, beïnvloeden niet alle vlaggen (zie ook het vlagoverzicht in appendix B). Sommige instructies gebruiken bit 2 van het vlagregister om aan te geven of er al dan niet overflow is opgetreden, andere instructies gebruiken dit bit om een even of oneven pariteit te signaleren. Onder laatstgenoemde instructies bevinden zich o.a. de logische instructie AND, OR en XOR. De term pariteit heeft betrekking op het aantal nullen en enen in een binair getal, in het geval van de Z-80 een achtbits-getal ofwel een byte. Bij even pariteit (Parity Even) is er een even aantal enen in het getal: 2, 4, 6 of 8. Is na één van bovengenoemde instructies een even aantal bits in de accumulator 1 dan zet de Z-80 het pariteitsbit op één. Bij oneven pariteit (Parity Odd) reset de Z-80 de Parity-vlag.
Over blijven de Zero- en Carry-vlaggen. Na onder andere rekenkundige en logische instructies set de Z-80 de Zero-vlag als het resultaat van de instructie 0 is. De rol van de Carry-vlag, het aangeven of er bij een berekening al dan niet een carry of borrow is opgetreden, is al behandeld in het vorige hoofdstuk.
De volgende vlaggen zijn toegankelijk voor de programmeur: Carry, Zero, Parity en Sign. De toestand van deze vlaggen kan bijvoorbeeld worden gebruikt bij voorwaardelijke spronginstructies.

JP	C	spring als er een carry is
JP	NC	spring als er geen carry is
JP	Z	spring als de zero-vlag geset is
JP	NZ	spring als de zero-vlag 0 is
JP	PO	pariteit oneven of overflow
JP	PE	pariteit even of geen overflow
JP	P	teken positief
JP	M	teken negatief

4.2 Testinstructies

Kenmerk van een testinstructie is dat deze de elementen die getest worden niet beïnvloedt, maar wel in de vlaggen het resultaat aangeeft. De Z-80 kent er een paar. De eerste die we zullen behandelen, is CP: ComPare (vergelijk). CP 23 bijvoorbeeld trekt 23 af van de accumulator zonder het resultaat in A terug te zetten. De inhoud van A verandert dus niet. Is het resultaat 0 dan set de Z-80 de Zero-vlag; is het resultaat niet 0 dan reset de Z-80 de Zero-vlag. Is de inhoud van A kleiner dan 23 dan treedt er een borrow op en set de Z-80 de Carry-vlag. Is de inhoud gelijk aan of groter dan 23 dan is er geen borrow en reset de Z-80 de Carry-vlag.
De volledige vorm van de instructie is CP *s*, waarin *s* kan zijn: *r*, *n*, (HL), (IX + d), (IY + d).

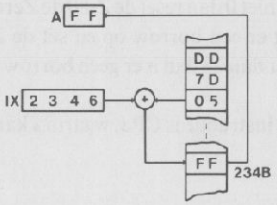
r:
Staat voor één van de achtbits-registers A, B, C, D, E, H of L. De instructie luidt dan CP B of CP E en de inhoud van het aangegeven register wordt, met de hierboven omschreven gevolgen, van de accumulator afgetrokken. Ook CP A is mogelijk. De uitkomst hiervan is natuurlijk altijd 0, dus is na CP A de Zero-vlag geset en de Carry-vlag gereset.
De instructie is snel en neemt maar één geheugenplaats in beslag. Bijv. CP D: BA (hexadecimaal)

n:
Staat voor onmiddellijke data. Deze moet in de instructie worden opgegeven, zoals in CP 23. Elk getal tussen 0 en 255 is mogelijk. De instructie gebruikt 2 geheugenplaatsen, 1 voor de opcode en 1 voor de data. De Z-80 moet het geheugen bij deze instructie tweemaal aanspreken; de uitvoering kost dan ook bijna tweemaal zoveel tijd als CP B. Formaat van CP 16 in het geheugen:

byte 1 FE opcode
byte 2 10 data (= 16 decimaal)

(HL):
Wijst naar de inhoud van de geheugenplaats waarvan het adres in HL staat. De instructie CP (HL) is maar één byte groot (BE), maar neemt evenveel tijd in beslag als de vorige instructie, omdat de Z-80 zowel de opcode als de inhoud van (HL) moet ophalen.

(IX+d) en (IY + *d*):
Dit zijn de beide indexregisters van de Z-80. In IX of IY staat een adres waar bij de offset of verplaatsing *d*, een in de instructie op te geven achtbits-getal, wordt opgeteld. Staat in IX bijv. adres 2346H dan laadt LD A,(IX + 5) de accumulator met de inhoud van adres 234BH. Zie afb. 4.2. CP (IX + *d*) vergelijkt de accumulator met de inhoud van het berekende adres IX + *d*. Instructies met de indexregisters zijn traag. CP (IX + *d*) kost bijna vier keer zoveel tijd als CP *r*.



Afb. 4.2 Geïndexeerd laden van de accumulator

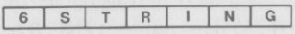
In alle genoemde gevallen stelt *s* een achtbits-getal voor, direct, in een register of in een geheugenlocatie. De CP-instructie trekt *s* af van de accumulatorinhoud, daarbij beide getallen onveranderd latend. Het resultaat van de aftrekking wordt genegeerd maar beïnvloedt wel de vlaggen. De volgende situaties zijn mogelijk.

s gelijk aan A: Zero-vlag 1, Carry-vlag 0
s groter dan A: Zero-vlag 0, Carry-vlag 1
s kleiner dan A: Zero-vlag 0, Carry-vlag 0

Hieruit laten zich combinaties samenstellen. *s* is gelijk aan of kleiner dan A als de Carry-vlag 0 is en *s* is gelijk aan of groter dan A als de Zero-vlag of de Carry-vlag 1 is.

4.3 De lengte van een string

Een via het toetsenbord ingetypte string, zoals bijvoorbeeld na INPUT A$ in een BASIC-programma, komt inclusief het carriage return-teken, dat er door het drukken op de RETURN-toets aan werd toegevoegd, tijdelijk in een buffer. Vanaf daar gaat de string naar de variabelenruimte. Een mogelijk formaat voor een string in de variabelenruimte van BASIC staat in afb. 4.3. In het eerste byte staat het aantal tekens van de string. Vandaar dat een string bij deze organisatiemethode niet meer dan 255 tekens mag bevatten. Na het eerste byte komen in de opeenvolgende adressen de ASCII-tekens voor de karakters waaruit de string bestaat.



Afb. 4.3 Het formaat van een string

Programma H4P1 berekent de lengte van de ingevoerde string in de buffer exclusief de voor het formaat als in afbeelding 4.3 verder niet nodige carriage return: in ASCII-code 0DH.

Programma H4P1 Bepaling van de stringlengte

```
        LD      HL,BUFFER       ;adres buffer in HL
        LD      B,0             ;teller B op 0
        LD      A,0DH           ;carriage return
WHILE:  CP      (HL)            ;in HL carr. return ?
        JP      Z,WEND          ;ja, einde lus
        INC     HL              ;nee, adres volgende teken
        INC     B               ;verhoog teller
        JP      WHILE           ;volgende teken
WEND:   LD      A,B             ;aantal tekens in A
        LD      (TOTAAL),A      ; en naar kop string
        RET

TOTAAL: DEFB    00
BUFFER: DEFB    48H             ;ASCII voor H
        DEFB    41H             ;ASCII voor A
        DEFB    4CH             ;ASCII voor L
        DEFB    4CH             ;ASCII voor l
        DEFB    4FH             ;ASCII voor O
        DEFB    0DH             ;ASCII carriage return

        END
```

De lus is van een WHILE-WEND-achtige structuur. De beslissing de lus al dan niet te doorlopen valt aan het begin ervan.

In HL staat het adres van het eerste teken in de string. Register B dient als teller en wordt daarom aan het begin op 0 gezet. In A staat het carriage return-teken.
Aan het begin van de lus, bij WHILE, kijkt CP (HL) of het teken in adres (HL) een carriage return is. Zoniet dan is de Zero-vlag niet geset en doorloopt het programma de lus, daarbij teller B en HL met 1 verhogend. In HL staat dan het adres van het volgende stringteken.
Is het teken in (HL) een carriage return dan set de instructie CP (HL) de Zero-vlag. Het programma springt naar WEND en verhoogt teller B niet. In B staat het aantal stringtekens. Instructies vanaf WEND zetten dit aantal in A en vervolgens aan de kop van de string zodat deze het juiste formaat krijgt om overgebracht te worden naar de variabelen-ruimte in het geheugen.
Het programma is niet in staat alert te reageren als de string groter is dan 255 tekens. Is de inhoud van B 255 dan resulteert INC B tot B = 0 en vervolgens tot B = 1 enz.

4.4 Blokvergelijkingsinstructies

Het doorzoeken van aantallen bytes op de aanwezigheid van een bepaald teken komt vaak voor. Steeds weer moet dan een programma tellers en adressen bijhouden. De Z-80 beschikt over instructies die dat automatisch doen. Het zijn de blokvergelijkingsinstructies CPI, CPD, CPIR en CPDR.

4.4.1 CPI ComPare Increment (vergelijk en verhoog)

De instructie veronderstelt in HL het beginadres van de te doorzoeken serie bytes, in ons geval dat van de string, in BC het maximum aantal te doorzoeken bytes en in A het te vinden teken. Op de teller na is de situatie als in het vorige programma. De instructie CPI doet nu het volgende.

- Een CP (HL)-instructie die de Zero-vlag set of reset.
- Een INC HL-instructie, een verhoging van HL dus.
- Een DEC BC-instructie. Indien BC daardoor 0 wordt en het maximum aantal bytes dus is doorzocht dan reset de instructie de P/V-vlag. Is BC niet 0 dan set de instructie de P/V-vlag. Let op: dit is anders dan bij de Zero-vlag. De Z-80 gebruikt de P/V-vlag voor deze instructie; het 0 worden van BC heeft met pariteit of overflow niets te maken. Het maakt echter niet uit waarvoor deze vlag wordt gebruikt; de enige beschikbare mnemonics zijn PO en PE.

Voor BC = 0 is de P/V-vlag gereset: dit komt overeen met Parity Odd (PO).
Voor BC niet 0 is de P/V-vlag geset: dit komt overeen met Parity Even (PE).

Gebruik makend van de CPI-instructie wijzigen we het vorige programma als volgt.

Programma H4P2 Bepaling van de stringlengte met CPI-instructie

```
        LD      HL,BUFFER       ;adres buffer in HL
        LD      A,ØDH           ;carriage return
        LD      B,Ø1H           ;in BC 256
        LD      C,Ø
WHILE:  CPI                     ;CP(HL), INC HL, DEC BC
        JP      PO,WEND         ;BC=Ø ? ja, einde lus
        JP      NZ,WHILE        ;A=(HL) ? nee, opnieuw
WEND:   LD      A,255
        SUB     C               ;in A aantal tekens
        LD      (TOTAAL),A      ; naar kop string
        RET

TOTAAL: DEFB    ØØ
BUFFER: DEFB    48H             ;ASCII voor H
        DEFB    41H             ;ASCII voor A
        DEFB    4CH             ;ASCII voor L
        DEFB    4CH             ;ASCII voor l
        DEFB    4FH             ;ASCII voor O
        DEFB    ØDH             ;ASCII carriage return

        END
```

Het programma is, op het werk met de teller na, vrij simpel. Of A gelijk is aan (HL) of niet, de CPI-instructie verlaagt altijd de inhoud van BC. Meteen na de eerste uitvoering van CPI is BC 255. Dat wil zeggen B = 0 en C = 255. Voor het aantal tekens hebben we dan alleen met C te maken.
Wordt een carriage return gevonden dan staat in C: 256 − (aantal tekens + 1) = 255 − aantal tekens. Na de vergelijking laden we A met 255 en trekken daar de inhoud van C vanaf. In A staat dan: 255 − (255 − aantal tekens) = aantal tekens. Is er geen carriage return gevonden dan doorloopt het programma de lus 256 maal voordat BC 0 is en de instructie de P/V-vlag set. De inhoud van C is dan 0 en TOTAAL krijgt de waarde van de maximaal toegestane stringlengte: 255.

4.4.2 CPIR ComPare Increment Repeat (vergelijk, verhoog, herhaal)

Dit is een CPI-instructie die zichzelf herhaalt totdat A gelijk is aan (HL) of BC = 0. Achtereenvolgens voert de instructie het volgende uit.

- Een CP (HL)-instructie die de Zero-vlag set of reset.

- Een INC HL-instructie.
- Een DEC BC-instructie die de P/V-vlag set of reset.
- Een herhaalinstructie. Is de P/V-vlag 1, dus BC niet gelijk aan 0, en is de Zero-vlag 0, dus A niet gelijk aan (HL), dan verlaagt de instructie de PC met 2. Aangezien de instructie zelf 2 bytes in beslag neemt wijst de PC hierna weer naar de eerste byte van CPIR. De instructie wordt dus opnieuw geladen, zij het met andere waarden in HL en BC.

Deze instructie toegepast in het telprogramma levert het volgende op.

programma H4P3 Bepaling van de stringlengte met CPIR-instructie

```
        LD      HL,BUFFER       ;adres buffer in HL
        LD      A,0DH           ;carriage return
        LD      BC,0100H        ;in BC 256
        CPIR
        LD      A,255
        SUB     C               ;in A aantal tekens
        LD      (TOTAAL),A      ; naar kop string
        RET

TOTAAL: DEFB    00
BUFFER: DEFB    48H             ;ASCII voor H
        DEFB    41H             ;ASCII voor A
        DEFB    4CH             ;ASCII voor L
        DEFB    4CH             ;ASCII voor l
        DEFB    4FH             ;ASCII voor O
        DEFB    0DH             ;ASCII carriage return

        END
```

De CPIR-instructie is vrij snel. Uitvoering ervan kost 21 T(tijdseenheden) per keer als BC ongelijk 0 en A ongelijk (HL) is. De laatste uitvoering van de instructie, als BC = 0 of A = (HL), kost 16 tijdseenheden. De lus in het oorspronkelijke programma nam aan tijd:

CP (HL)	7 T
JP Z	10 T
INC HL	6 T
INC B	4 T
JP	10 T
	37 T +

Daar staat tegenover dat zichzelf herhalende instructies zoals CPIR iets meer aan

voorbereidende en verwerkende instructies kosten. Zo moet BC worden geladen i.p.v. alleen B en is er één instructie meer nodig om aan het aantal tekens te komen. Bedenk echter dat een snellere lus bij elke doorloop voordeel oplevert.

4.4.3 CPD ComPare Decrement (vergelijk, verlaag)

Gelijk aan CPI behalve dat de instructie HL verlaagt i.p.v. verhoogt.

4.4.4 CPDR ComPare Decrement Repeat (vergelijk, verlaag, herhaal)

Gelijk aan CPIR behalve dat de instructie net als bij CPD HL verlaagt i.p.v. verhoogt.

CPI en CPIR doorzoeken het geheugen van laag naar hoog, CPD en CDPR van hoog naar laag. Of de inhoud van A nu gelijk is of niet aan die van geheugenlocatie (HL), de blokvergelijkingsinstructies verhogen of verlagen HL altijd. Is bij een CPIR A=(HL) dan wijst HL na de instructie naar het byte volgend op het gezochte. Na een CPDR naar het voorafgaande.

In tegenstelling tot het vorige programma is registerpaar BC in eenmaal geladen met 256 in plaats van in 2 fasen. Evenals de achtbits-registers kunnen de zestienbits-registers van de Z-80 worden geladen met een bepaalde waarde. In feite gebeurde dat al met LD HL,BUFFER.

4.5 Het vergelijken van strings

Met het volgende programma willen we twee strings met elkaar vergelijken. Net als in BASIC's IF A$=B$ gaat het erom te constateren of de strings aan elkaar gelijk zijn of niet.
In het algemeen is een ingetypte string, zoals die uit het vorige voorbeeld, van de buffer naar de ruimte voor variabelen in het geheugen gebracht. In een hogere programmeertaal, zoals BASIC, zal de naam van de string in een lijst met variabelen staan. De naam van elke variabele in die lijst wordt gevolgd door het eerste adres van de opeenvolgende geheugenplaatsen waarin de waarde van de variabele staat. We veronderstellen de adressen van beide te vergelijken strings bekend. In het eerste byte van elke string staat het aantal karakters.

Programma H4P4 Het vergelijken van strings

```
        LD      HL,STRING1      ;startadres string 1
        LD      DE,STRING2      ;startadres string 2
        LD      B,(HL)          ;aantal tekens string 1 plus
        INC     B               ; byte voor aantal in teller B
WHILE:  LD      A,(DE)          ;teken uit string 2
        CP      (HL)            ;gelijk aan dat in string 1 ?
        JP      NZ,WEND         ;nee, einde lus
        INC     HL              ;volgende teken in string 1
        INC     DE              ;volgende teken in string 2
        DJNZ    WHILE           ;vergelijk opnieuw
WEND:   LD      A,B             ;indien strings gelijk B=0
        LD      (RESULT),A      ; anders B ongelijk 0
        RET

RESULT: DEFB    00
STRING1:
        DEFB    03              ;aantal tekens
        DEFB    44H             ;ASCII voor D
        DEFB    41H             ;ASCII voor A
        DEFB    47H             ;ASCII voor G
STRING2:
        DEFB    03
        DEFB    44H
        DEFB    41H
        DEFB    47H

        END
```

De startadressen van de strings gaan naar HL en DE. In teller B komt het aantal tekens van string 1. Vervolgens wordt B met 1 verhoogd omdat het programma ook de eerste bytes, waarin de aantallen tekens staan, met elkaar vergelijkt. De lus test de opeenvolgende bytes van beide strings tot er twee niet gelijk zijn of teller B nul is. Zijn de eerste bytes van de strings, waarin het aantal tekens staat, niet gelijk dan zijn de strings niet gelijk. Een nog onbekende instructie is DJNZ: Decrement Jump Non Zero (verlaag, spring indien niet nul). De instructie werkt alleen in combinatie met register B als teller en doet het volgende:

- De inhoud van B met 1 verlagen. Deze verlaging beïnvloedt in tegenstelling tot een gewone DEC-instructie bij een achtbits-register de vlaggen niet.
- Een sprong uitvoeren als B ongelijk is aan 0. Dit is geen gewone sprong als bij JP NZ maar een relatieve sprong: JR (Jump Relative).

JP	JR
C3 opcode	18 opcode
q lage byte adres	e verplaatsing
p hoge byte adres	

Bij JP staat na de opcode in de instructie het adres waarnaar het programma moet springen. Aangezien het een zestienbits-adres is, neemt het 2 bytes in beslag. Uitvoering van JP laadt adres pq in de PC.
In de instructie voor de relatieve sprong staat geen adres maar een verplaatsing van één byte. Uitvoering van JR telt verplaatsing e op bij de waarde van de PC. Kan een JP-instructie door het gehele geheugen springen, het bereik van JR is beperkt: 127 bytes vooruit en 128 terug. Tussen de labels WHILE en WEND in het voorbeeld mogen niet meer dan 128 bytes liggen.
Relatieve sprongen bestaan ook buiten de DJNZ-instructie en hebben als mnemonic JR. Bij gebruik van labels berekent de assembler de verplaatsing e automatisch en waarschuwt als de sprong buiten het bereik van $+127$ en -128 ligt. Relatieve sprongen kunnen ook voorwaardelijk zijn maar er zijn minder mogelijkheden dan bij de gewone sprong: JR Z, JR NZ, JR C, JR NC. Voordeel van JR is dat de instructie één byte minder geheugenruimte vergt. Voor onvoorwaardelijke sprongen is JR trager dan JP. Bij voorwaardelijke sprongen is JR trager als de sprong gemaakt wordt, anders sneller.

Is in het voorbeeld het programma bij WEND aangekomen dan is B nul als de strings gelijk zijn. Dan immers zijn alle tekens met elkaar vergeleken. Is B niet nul dan is naar WEND gesprongen omdat 2 tekens ongelijk waren. De instructies na WEND zetten de waarde van B in RESULT. Als RESULT gelijk is aan nul zijn de strings gelijk.

4.6 De FOR-NEXT-achtige lus

Bij deze lus ligt het aantal herhalingen vast. De structuur ervan is simpel.

```
        LD B,AANTAL   ;B is teller
NEXT:   te herhalen routine
        :
        :
        DJNZ NEXT
```

Probleem in dit geval is dat het aantal herhalingen niet groter mag zijn dan 255, de maximale inhoud van B. Met een kleine kunstgreep zijn 256 herhalingen mogelijk. Indien men B laadt met 0 zal de eerste uitvoering van DJNZ de inhoud van B eerst 'verlagen' tot 255 voordat gekeken wordt of B al dan niet nul is. 0-1 levert namelijk 255 op.
Moet het aantal herhalingen toch groter zijn dan is een dubbele lus de oplossing.

```
        LD C,TELLER   ;herhalingen van lus
LUS:    LD B,AANTAL   ;herhalingen in lus
NEXT:   te herhalen routine
:
:
        DJNZ NEXT
        DEC C         ;verlaag C
        JP NZ,LUS     ;C=0?    nee,dan    lus
                      opnieuw
```

De lus zelf is gelijk aan die in het vorige voorbeeld en wordt herhaald tot B nul is. DEC C verlaagt vervolgens de inhoud van C. Als deze niet nul is, springt het programma naar lus en laadt teller B opnieuw. Het totaal aantal herhalingen is TELLER*AANTAL, maximaal 256*256.
In dit geval is een registerpaar als teller minder goed bruikbaar omdat hierbij een DEC-opdracht de vlaggen niet beïnvloedt. De bytes van het registerpaar moeten dan afzonderlijk op de waarde nul worden onderzocht.

4.7 Uitvoering van een LEFT$-functie

De BASIC-opdracht B$ = LEFT$(A$,3) creëert een substring B$ bestaande uit de eerste 3 tekens van A$. Zijn de startadressen van A$ en B$ bekend dan komt uitvoering van de opdracht neer op het kopiëren van 3 tekens. Dit kan gebeuren met een FOR-NEXT achtige lus. Het aantal te kopiëren tekens staat in AANTAL.

Programma H4P5 De LEFT$-functie

```
;Zet de startadressen van string en substring in HL en DE
        LD      HL,STRING
        LD      DE,SUBS
;Zet het aantal tekens voor de substring in het eerste
;byte daarvan en in teller B
        LD      A,(AANTAL)
        LD      (DE),A
        LD      B,A
;Adressen van de volgende karakters in HL en DE
NEXT:   INC     HL
        INC     DE
;Breng een karakter over van string naar substring
        LD      A,(HL)
        LD      (DE),A
;Net zolang tot teller B=0
        DJNZ    NEXT
        RET
```

```
AANTAL: DEFB    03              ;aantal tekens substring
STRING: DEFB    08              ;aantal tekens string
        DEFB    'D'
        DEFB    'A'
        DEFB    'G'
        DEFB    'E'
        DEFB    'R'
        DEFB    'A'
        DEFB    'A'
        DEFB    'D'
SUBS:   DEFS    4

        END
```

Er is meteen een nieuw assembler-directive gebruikt: DEFS (DEFine Storage) reserveert tijdens het assembleren het erachter opgegeven aantal bytes, in dit geval 4. Daarvan is er één voor het aantal tekens in de substring; de overige drie zijn voor de tekens zelf. De inhoud van STRING is nu niet opgegeven in ASCII-code maar gewoon door de tussen haakjes geplaatste letters. De assembler zet daarvoor in de plaats de juiste code. Het benutten van deze mogelijkheid maakt het programma universeler: het is nu ook bruikbaar als de computer een andere dan de ASCII-code hanteert.

4.8 Blokverplaatsingsinstructies

Net als bij de CP-instructie het geval was, zijn er laadinstructies die adressen in registers verhogen of verlagen en zichzelf eventueel herhalen. Het zijn LDI, LDIR, LDD, LDDR.

4.8.1 LDI LoaD Increment (laad, verhoog)

De instructie veronderstelt het adres waar een databyte vandaan komt, de bron, in HL en het adres waar het databyte heen moet, de bestemming, in DE. Het aantal te verplaatsen bytes staat in BC.
De instructie doet het volgende:

- Verplaatst zonder gebruik van registers een byte van (HL) naar (DE).
- Verhoogt zowel HL als DE met 1.
- Verlaagt BC, reset de P/V-vlag als BC=0 en set de P/V-vlag als BC niet 0 is.

4.8.2 LDIR LoaD Increment Repeat (laad, verhoog, herhaal)

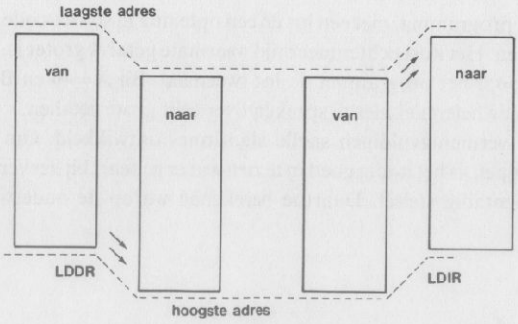
De instructie voert LDI uit totdat BC = 0. De herhaling vindt plaats door, als BC niet nul is na de verlaging, van de PC twee af te trekken. Aangezien de instructie zelf twee bytes in beslag neemt wijst in dat geval de PC weer naar het eerste byte daarvan. De maximale inhoud van BC is 65.535. Is BC aan het begin van de instructie 0 dan kunnen 65.536 bytes ofwel een volle 64 Kb worden verplaatst. De vraag is in dit laatste geval echter: waarheen? De Z-80 kan zonder extra voorzieningen namelijk niet meer dan 64 Kb aan geheugen adresseren.
Gebruik makend van LDIR krijgen we het volgende programma voor de LEFT$-functie.

Programma H4P6 LEFT$-functie met LDIR

```
;Constante:
LEFT$   EQU     3
;Zet de startadressen van string en substring in HL en DE
        LD      HL,STRING
        LD      DE,SUBS
;Zet het aantal tekens voor de substring in het eerste
;byte daarvan en in teller BC
        LD      A,LEFT$
        LD      (DE),A
        LD      C,A
        LD      B,0
;Adressen van de volgende karakters in HL en DE
        INC     HL
        INC     DE
;Breng het gewenste aantal karakters over
        LDIR
        RET

STRING: DEFB    08
        DEFM    'DAGERAAD'
SUBS:   DEFS    LEFT$ + 1

        END
```

Om het aantal over te brengen tekens makkelijk te kunnen veranderen, is een constante geïntroduceerd, namelijk LEFT$. EQU staat voor EQUate, maak gelijk aan. De constante LEFT$ zelf neemt in de objectcode geen ruimte in beslag maar overal waar de assembler het woord LEFT$ tegenkomt zet deze de waarde van LEFT$, 3 dus, daarvoor in de plaats.
Het aantal over te brengen tekens is 3 en de lengte van de substring is 3 plus 1 byte voor het aantal tekens. Bij praktisch alle assemblers mogen zulke rekenkundige uitdrukkingen worden opgegeven. Bedenk wel dat de berekening plaatsvindt

tijdens het assembleren en niet tijdens het runnen van een programma. Het is dus niet mogelijk langs deze weg iets bij de inhoud van een register op te tellen. Voor het overbrengen van een ander aantal tekens moet men dus LEFT$ veranderen en opnieuw assembleren.
DEFM staat voor DEFine Memory. De assembler plaatst de codes voor de letters in de tussen haakjes staande string in opeenvolgende bytes. Het effect is hetzelfde als dat van de serie DEFB-directives in het vorige programma.
Net als bij CPIR zijn er bij de blokverplaatsingen vaak wat meer voorbereidende instructies nodig dan bij gebruik van een gewone lus. Dit wordt ruimschoots gecompenseerd doordat de lus zelf, LDIR, zeer kort en zeer snel is.
De LDD- en LDDR-instructies doen hetzelfde als LDI en LDIR maar verlagen na elke verplaatsing HL en DE. LDDR staat voor LoaD Decrement Repeat: laad, verlaag, herhaal. In ons voorbeeld zouden bij gebruik van LDDR de startadressen in HL en DE STRING + 3 respectievelijk SUBS + 3 moeten zijn. Gebruik van LDDR ligt voor de hand bij het uitvoeren van een RIGHT$-functie waarbij, beginnend met het laatste teken van STRING, bytes verplaatst worden.
Merk op dat het in de programma's H4P5 en H4P6 ook mogelijk was geweest HL meteen te laden met STRING + 1 en de INC HL-instructie weg te laten. Deze asymmetrische behandeling van beide strings zou de leesbaarheid van de programma's echter niet ten goede zijn gekomen.
LDIR en LDDR zijn de aangewezen instructies voor het verplaatsen van hoeveelheden aaneengesloten bytes, ofwel blokken, in het geheugen. Vaak maakt het niet uit of men daarbij begint op het laagste adres van het blok en dan LDIR gebruikt of op het hoogste met LDDR. Indien echter bron en bestemming elkaar overlappen, door bijvoorbeeld een blok van één kilobyte een afstand van 10 bytes te verplaatsen, kan bij een verkeerde keuze het blok zichzelf overschrijven.
Zie afb. 4.4.



Afb. 4.4 Blokverplaatsing met LDDR en LDIR

5 Vermenigvuldigen en machtsverheffen

5.1 Het principe van de vermenigvuldiging

Tot nu toe zijn alleen programma's besproken voor optellen en aftrekken, rekenkundige bewerkingen waarvoor de Z-80 instructies heeft. Vermenigvuldigen of worteltrekken kan de Z-80 niet. In BASIC zijn zulke bewerkingen wel mogelijk. De interpreter beschikt over programma's die bijvoorbeeld een vermenigvuldiging uitvoeren door een reeks Z-80-instructies af te werken.
Het algoritme voor het maken van een vermenigvuldiging kan bijvoorbeeld een herhaald aantal malen optellen zijn: $3 \times 5 = 5 + 5 + 5$. Als BASIC geen vermenigvuldiging kende, zou men die kunnen maken met een FOR-NEXT-lus en een optelling. Het volgende programma rekent A × B uit.

```
:
:
60 C=0
70 FOR N=1 TO A
80 C=C+B
90 NEXT N
:
:
```

Een dergelijk programma, met een lus en een optelling in source code, zou ook op de Z-80 werken. Het kost echter meer tijd naarmate getal A groter is. Bij $A = 2$ en $B = 40$ doorloopt het programma de lus tweemaal. Bij $A = 40$ en $B = 2$ veertigmaal. Om maar helemaal niet te spreken over echt grote getallen.
Er zijn voor vermenigvuldigen snelle algoritmes ontwikkeld. Om de werking ervan te begrijpen, is het nodig goed in te zien wat er gebeurt bij het vermenigvuldigen in het tientallig stelsel. Daartoe berekenen we op de ouderwetse manier 25×57.

$$\begin{array}{r} 57 \\ 25 \\ \hline 285^{5} \end{array} \times$$

Voor de eerste stap van de vermenigvuldiging, $5 \times 57 = 285$, maken we gebruik van de distributieve eigenschap: $25 \times 57 = (20 + 5) \times 57 = 20 \times 57 + 5 \times 57$. 5×57 hebben we al uitgerekend. Om 20×57 uit te rekenen, splitsen we 20 in 2×10 en berekenen we 2×57.

$$\begin{array}{r} 57 \\ 25 \\ \hline 285 \\ 114 \end{array} \times$$

$20 \times 57 = 2 \times 57 \times 10$.

De factor 10 verwerken we door de uitkomst van 2×57 een plaats naar links te schuiven, wat in het tientallig stelsel hetzelfde betekent als met 10 vermenigvuldigen.
Binair vermenigvuldigen gaat op dezelfde manier. Daarbij gelden de volgende simpele rekenregels:

$0 \times 0 = 0$
$1 \times 0 = 0$
$1 \times 1 = 1$

Om te beginnen een eenvoudige berekening: 3×5

$$\begin{array}{r} 101 \\ 11 \\ \hline 101 \end{array} \times$$

Volgens de distributieve regel is 11×101 hetzelfde als $10 \times 101 + 1 \times 101$. Dezelfde werkwijze gebruikend als bij het tientallig stelsel vermenigvuldigen we eerst het meest rechtse cijfer van 11 met 101.

$$\begin{array}{r} 101 \\ 11 \\ \hline 101 \\ 101 \\ \hline 1111 \end{array} \begin{array}{l} \times \\ \\ \\ + \\ \end{array}$$

Daarna komt 10×101. Evenals bij het tientallig stelsel betekent vermenigvuldigen met het grondtal alle cijfers een positie naar links verschuiven. Het binaire getal 101 heeft de waarde $1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2$. Vermenigvuldiging met het grondtal 2, binair 10, geeft: $1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3$ wat overeenkomt met 1010. Uit het voorbeeld volgt hoe simpel de vermenigvuldiging voor binaire getallen is:

vermenigvuldigen met 0 levert 0 op;
vermenigvuldigen met 1 heeft 1 als resultaat;
elke nul die achter bovengenoemde 1 staat, heeft als gevolg dat het resultaat één positie naar links wordt geschoven.

In de programma's passen we de algemene schuif-en-tel-op-tactiek toe. Wat dat inhoudt, laat het volgende voorbeeld zien. We vermenigvuldigen daarin 17 met 13.

```
factor1   0001 0001          =17
factor2   0000 1101      ×   =13
          ---------
          0001 0001
       00 0100 01
      000 1000 1
     ---------------     +
produkt   1101 1101          =221
```

In plaats van alle deelprodukten in een rijtje onder elkaar te zetten en aan het einde op te tellen, zoals hierboven, zullen we in de programma's elk deelprodukt meteen optellen bij het resultaat. Dit resultaat moet dus aan het begin van de vermenigvuldiging 0 zijn. Het waarom van zo'n tactiek ligt voor de hand: het is bijzonder onvoordelig alle deelprodukten eerst in het geheugen op te slaan. Toepassen van de omschreven werkwijze leidt tot het volgende:

factor1 0001 0001 factor2 0000 1101 produkt 0000 0000
bit 0 van factor2 = 1; tel op en schuif factor1 naar links
factor1 0010 0010 factor2 0000 1101 produkt 0001 0001
bit 1 van factor2 = 0; schuif factor1 naar links
factor1 0100 0100 factor2 0000 1101 produkt 0001 0001
bit 3 van factor2 = 1; tel op en schuif factor1 naar links
factor1 1000 1000 factor2 0000 1101 produkt 0101 0101
bit 4 van factor2 = 1; tel op en schuif factor1 naar links
factor1 0001 0000 factor2 0000 1101 produkt 1101 1101

De overige bits van factor2 zijn 0. Er vindt dus geen optelling meer plaats. Bovendien is factor1 nu zover naar links geschoven dat het niet meer in acht bits past. Is bit 5 van factor2 ook 1, dan is de uitkomst niet correct en wel omdat deze groter is dan 255.
Voor het naar links schuiven van factor1 is er een speciale schuifinstructie. Het onderzoeken van de bits van factor2 doen we met een instructie voor het schuiven naar rechts waarbij de bits één voor één in de Carry komen. Is de Carry 0 dan slaan we de optelling van factor1 en resultaat over met een voorwaardelijke sprong: JP NC.
Merk op dat na de eerste ronde bit 0 van het produkt niet meer verandert, na de tweede ronde bit 1 niet meer enz.

5.2 Programma's voor vermenigvuldigen

Programma H5P1 achtbits-vermenigvuldiging

```
;       Vermenigvuldiging
;
;FACT1 * FACT2 = PROD
;
;Zet factor 1 in register D en factor 2 in E
        LD      HL,FACT1        ;adres FACT1
        LD      D,(HL)
        INC     HL              ;adres FACT2
        LD      E,(HL)
;Het product komt in A. Daarom moet A aan het
;begin 0 zijn. Register B dient als teller.
        XOR     A
        LD      B,8
;Schuif het laagste bit van factor 2 naar de
;Carry. Is de Carry 1 tel dan FACT1 op bij het
;product in A. Schuif dan FACT1 een bit naar
;links.
VLUS:   SRL     E               ;schuif FACT2 rechts
        JP      NC,VEIND        ;Carry 1 ?
        ADD     A,D             ;ja, tel op
VEIND:  SLA     D               ;schuif FACT1 links
        DJNZ    VLUS            ;alle bits gedaan ?
        LD      (PROD),A        ;ja, berg product op
        RET

FACT1:  DEFB    09
FACT2:  DEFB    11
PROD:   DEFB    00

        END
```

Bij de hier gevolgde methode doorloopt het programma voor een achtbits-vermenigvuldiging de lus achtmaal. Later zullen we zien dat bij een vermenigvuldiging van 32-bits-getallen de lus 32 maal wordt doorlopen.
Allereerst komen de waarden voor FACT1 en FACT2 in de registers D en E. Aangezien het produkt FACT1 × FACT2 in de accumulator wordt opgebouwd, moet de inhoud hiervan aan het begin 0 zijn. LD A,0 doet dat, maar kost tweemaal zoveel aan tijd en geheugenruimte als XOR A. XOR staat voor eXclusive OR. De algemene notatie is XOR *s* waarbij *s* net als in CP het volgende kan zijn.

r: ofwel één van de achtbits-registers bijv. XOR H
n: onmiddellijke data in de instructie bijv. XOR 28
(HL): de inhoud van het adres in HL, XOR (HL)
(IX + *d*) en (IY + *d*) bijv. XOR (IY + 6)

De XOR-functie voert een exclusieve OR uit tussen de inhoud van de accumulator en *s* en zet het resultaat in de accumulator. XOR behoort tot de logische functies evenals AND en OR. Deze vergelijken twee binaire getallen bit voor bit en geven het volgende resultaat.

A	0101 1110
s	0100 1010
A AND *s*	0100 1010

Bij de AND(EN)-functie is een bit van het resultaat 1 als beide overeenkomstige bits in A en *s* 1 zijn. De OR (OF)-functie maakt een bit 1 als het overeenkomstige bit in A of dat in *s* 1 is of beide 1 zijn.

A	0101 1110
s	0100 1010
A OR *s*	0101 1110

De XOR (EXCLUSIEVE OR) doet hetzelfde als de OR-functie maar niet als beide overeenkomstige bits 1 zijn.

A	0101 1110
s	0100 1010
A XOR *s*	0001 0100 XOR A

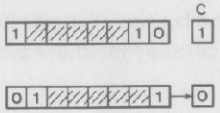
geeft een exclusieve OR van A met zichzelf. Is een bit in A 0 dan blijft het 0. Is het 1 dan zijn beide overeenkomstige bits één en is het bit in het resultaat 0.

A	0101 1110
A	0101 1110
AXOR A	0000 0000

We zullen nu eerst de vermenigvuldiging stap voor stap uitwerken door te laten zien wat er bij elke doorloop van de lus gebeurt. De beginsituatie is als volgt.

A	0000 0000
E	0000 1011
D	0000 1001

De eerste instructie in de lus is SRL E. De mnemonic SRL staat voor Shift Right Logical, een logische verschuiving naar rechts. SRL E schuift alle bits in E één plaats naar rechts. Bit 0 komt in de Carry en het vrijkomende bit zeven wordt opgevuld met 0. Zie afb. 5.1.



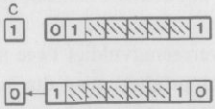
Afb. 5.1 Een logische verschuiving naar rechts

A	0000 0000
E	0000 0101
Carry = 1	
D	0000 1001

Is er geen Carry dan springt het programma naar label VEIND. In het andere geval telt ADD D de inhoud van D op bij die van A. Aangezien er een Carry is, vindt een optelling plaats.

A	0000 1001
E	0000 0101
D	0000 1001

Of SRL E nu wel of niet een Carry veroorzaakte, het programma werkt de instructies na VEIND af. SLA D, de volledige vorm is SLA *m*, schuift de inhoud van D één plaats naar links ofwel vermenigvuldigt de inhoud van D met het grondtal, precies hetgeen we in het voorbeeld van de vermenigvuldiging deden. Zie afb. 5.2. *m* kan zijn: *r*, (HL), (IX + *d*) en (IY + *d*).



Afb. 5.2 Een rekenkundige verschuiving naar rechts

A 0000 1001
E 0000 0101
D 0001 0010

SLA staat voor Shift Left Arithmetic, een rekenkundige verschuiving naar links. In het vrijkomende bit 0 komt een 0 te staan en het eruit vallende bit 7 gaat naar de carry. In dit geval doen we daar niets mee. Rekenkundig gezien komt een verschuiving naar rechts zoals in SRL neer op deling door het grondtal, dus door 2, en een verschuiving naar links op vermenigvuldiging met het grondtal.
DJNZ VLUS ten slotte verlaagt teller B en springt naar label VLUS.
Na de vierde doorloop van de lus verandert er niets meer aan het produkt. We geven nog even de situatie aan het einde van elke lus. C voor Carry geeft aan of er al dan niet is opgeteld, C correspondeert dus met de inhoud van de Carry na SRL E.

1	A = 0000 1001	E = 0000 0101	D = 0001 0010 C = 1
2	A = 0001 1011	E = 0000 0010	D = 0010 0100 C = 1
3	A = 0001 1011	E = 0000 0001	D = 0100 1000 C = 0
4	A = 0110 0011	E = 0000 0000	D = 1001 0000 C = 1
5	A = 0110 0011	E = 0000 0000	D = 0010 0000 C = 0

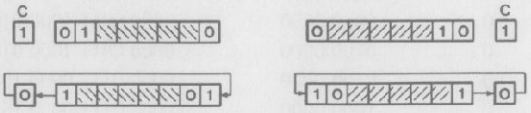
We zien dat na de vijfde doorloop van de lus FACT2 niet meer in register D past. Zou er in E nu nog een 1 staan dan is het resultaat niet correct. Zoals eerder gezegd, praktisch gezien betekent het dat het antwoord nooit groter mag zijn dan 255. Voor het rekenen met grotere getallen, bijv. 16 bits, kan men de registerparen gebruiken. De maximum mogelijke waarde van het resultaat is dan 65.535. Verderop in dit boek komen berekeningen met 32 bits-getallen aan de orde. Maximale waarde daarbij is 4.294.967.295. Hierbij en bij de later behandelde floating point-berekeningen worden twee registerparen aaneengeschakeld om een getal te kunnen bevatten.
Van de achtbits-microprocessors is de Z-80 rijk gezegend met interne registers. Vergeleken daarbij komt bijv. de 6502 er met z'n drie achtbits-registers nogal bekaaid af. Een simpele vermenigvuldiging kan dan al niet meer binnen de processor zelf worden uitgevoerd. Bewerkingen als verschuiven hebben dan de inhoud van een geheugenlocatie als operand en niet de inhoud van een register. Gezegd moet echter worden dat de 6502 zulke operaties veel sneller uitvoert dan de Z-80.
Het volgende programma vermenigvuldigt twee achtbits-getallen en staat een zestienbits-resultaat toe. Het produkt mag dus maximaal 65.535 zijn. Ruimschoots voldoende want het grootst mogelijke produkt van twee achtbits-getallen is $255 \times 255 = 65.025$.

Programma H5P2 8 × 8-bits-vermenigvuldiging met zestienbits-resultaat

```
;       Vermenigvuldiging
;
;8 bits * 8 bits met een 16 bits product

        LD      HL,FACT1        ;adres FACT1
        LD      C,(HL)          ;FACT1 in C
        INC     HL              ;adres FACT2
        LD      E,(HL)
        LD      D,0             ;FACT2 in DE
        LD      HL,0            ;maak product 0
        LD      B,8             ;bitteller
VLUS:   SLA     L               ;schuif product links
        RL      H
        SLA     C               ;schuif FACT1 links
        JP      NC,GNOPT        ;geen carry, geen optelling
        ADD     HL,DE           ;som product en FACT2
GNOPT:  DJNZ    VLUS            ;alle bits gedaan ?
        LD      (PROD),HL       ;berg product op
        RET

FACT1:  DEFB    200
FACT2:  DEFB    150
PROD:   DEFW    00

        END
```

Ook in dit programma enkele nieuwigheden. Allereerst het assembler-directive DEFW (DEFine Word) dat twee bytes reserveert voor het produkt. ADD HL,DE telt de inhoud van de twee zestienbits-registerparen op en zet het resultaat in HL. De algemene vorm is ADD HL,ss waarbij ss kan zijn: BC, DE, HL en SP. Er is ook een ADC HL,ss (ADd with Carry) en SBC HL,ss (SuBtract with Carry). Bij al deze instructies staat vooraf in HL één van de operanden en na uitvoering ervan het resultaat. HL fungeert voor deze zestienbits rekenkundige instructies als accumulator.

Een onbekende instructie is RL, Rotate Left ofwel roteer links. Bij de schuifinstructies ging één bit naar de Carry en werd het vrijkomende bit opgevuld met 0. Roteerinstructies schuiven een bit naar de Carry en tegelijk de oorspronkelijke inhoud van de Carry naar het vrijkomende bit. Zie afb. 5.3



Afb. 5.3 Roteer links

De volledige notatie is RL *m*, waarin *m* kan zijn:
r: één van de achtbits-registers bijv. RL A of RL L
(HL): roteert het byte waarvan het adres in HL staat
(IX + *d*) en (IY + *d*)

Er is, zoals afb. 5.3 laat zien, ook een instructie RR *m* (Rotate Right of roteer rechts). Voor de inhoud van de accumulator zijn er twee snelle roteerinstructies RRA en RLA. Hier doet zich het vreemde feit voor dat er twee instructies bestaan om de accumulator te roteren, RLA en RL A, RRA en RR A. De historische achtergrond hiervan zal in hoofdstuk acht aan de orde komen.
Het totale effect van SLA L en RL H is dat de inhoud van HL één bit naar links wordt geschoven.

	H	C	L
	0000 0000	?	1101 0000
na SLA L	0000 0000	1	1010 0000
na RL H	0000 0001	0	1010 0000

Hetzelfde kan sneller met ADD HL,HL. De inhoud van HL bij zichzelf optellen is hetzelfde als met 2 vermenigvuldigen ofwel één bit naar links schuiven.
En dan het programma zelf. Factor 1 schuift bit voor bit naar links de carry in. Het produkt schuift in elke doorloop van de lus eveneens naar links. Is een uit factor 1 naar de carry geschoven bit 1 dan wordt factor 2 opgeteld bij het produkt. De eerste keer dat het produkt naar links schuift, verandert er niets omdat het produkt dan nog 0 is. We doorlopen de lus nu achtmaal. De inhoud van DE is 0000 0000 1001 0110 = 150 decimaal.

	carry	C	HL
HL links	?	1100 1000	0000 0000 0000 0000
C links	1	1001 0000	0000 0000 0000 0000
HL + DE	0	1001 0000	0000 0000 1001 0110
HL links	0	10010 000	0000 0001 0010 1100
C links	1	0010 0000	0000 0001 0010 1100
HL + DE	0	0010 0000	0000 0001 1100 0010
HL links	0	0010 0000	0000 0011 1000 0100
C links	0	0100 0000	0000 0011 1000 0100
HL links	0	0100 0000	0000 0111 0000 1000
C links	0	1000 0000	0000 0111 0000 1000
HL links	0	1000 0000	0000 1110 0001 0000

C links	1	0000 0000	0000 1110 0001 0000
HL+DE	0	0000 0000	0000 1110 1010 0110

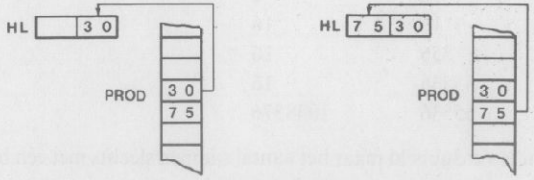
Na deze 5 lussen is C nul. Er vindt geen optelling meer plaats. Wel wordt HL nog driemaal naar links geschoven en is dan: 0111 0101 0011 0000 ofwel 7530 hex. Dat is $3 \times 16 + 5 \times 16^2 + 7 \times 16^3 = 30.000$.
Het vorige programma schoof factor 1 naar rechts en factor 2 naar links. Het huidige schuift factor 1 en het produkt naar links bij het doorlopen van de lus. Dat lijkt niet te kloppen maar in feite komt het op hetzelfde neer.

```
factor 2      125
factor 1       56   x
```

We kunnen, zoals in het eerste programma, 6×125 berekenen en dan voor de volgende vermenigvuldiging 125 naar links schuiven zodat we 5×1250 kunnen berekenen. Of, zoals in programma 2, we berekenen 5×125, schuiven het produkt naar links zodat het resultaat dat van 5×1250 wordt, en berekenen vervolgens 6×125.
Bij uittesten onder een monitor staat het zestienbits-resultaat in PROD met het laagste byte eerst. Dus:

```
PROD:   30 ;laagste byte produkt
        75 ;hoogste byte produkt
```

Zestienbits-laadacties tussen registerpaar en geheugen bergen eerst de laagste acht bits op in het aangegeven adres, in dit geval dat van label PROD. In de volgende geheugenlocatie, PROD+1, komen de hoogste acht bits. Omgekeerd, het laden van een registerpaar, bijv. met LD HL,(PROD), zet het byte in adres PROD in L en dat in PROD+1 vervolgens in H.



Afb. 5.4 16 bits laden vanuit het geheugen

5.3 Een algoritme voor machtsverheffen

Net zoals vermenigvuldigen in principe mogelijk is door herhaaldelijk optellen, kan men machtsverheffen door herhaaldelijk vermenigvuldigen. Immers, $4^3 = 4 \times 4 \times 4$. Voor heel kleine getallen is zoiets nog te doen.
Bij het vermenigvuldigen met het schuif-en-tel-op-algoritme is het produkt van twee achtbits-getallen in acht stappen bereikt. Later zullen we zien dat het produkt van 32-bits-getallen zich vormt in 32 stappen. Herhaaldelijk optellen zou in dit laatste geval tienduizenden stappen kunnen eisen, bijv. bij 47.000×58.000, waarvan het antwoord (2.726.000.000) binnen 32 bits (max. 4.294.967.295) blijft.

Het is dan ook niet verwonderlijk dat we ook voor het machtsverheffen een speciaal algoritme hanteren. We berekenen $Z = K^n$ als volgt. Bij aanvang is $Z = 1$.

HERHAAL:

als n even is dan $n = n/2$ en $K = K \times K$
als n oneven is dan $n = n\text{-}1$ en $Z = Z \times K$

TOTDAT $n = 0$

Als voorbeeld wordt 4^5 met gebruikmaking van het algoritme uitgerekend. Dus $n = 5$, $K = 4$ en $Z = 1$.

1e maal: n is oneven dus $n = n\text{-}1 = 4$ en $Z = 1 \times 4 = 4$
2e maal: n is even dus $n = n/2 = 2$ en $K = 4 \times 4 = 16$
3e maal n is even dus $n = n/2 = 1$ en $K = 16 \times 16 = 256$
4e maal: n is oneven dus $n = n\text{-}1 = 0$ en $Z = 4 \times 256 = 1024$

De berekening is gemaakt in 4 stappen. Dat lijkt weinig spectaculair maar we doen nu hetzelfde voor 4^{10} en geven alleen n, Z en K na elke ronde.

n	K	Z
5	16	1
4	16	16
2	256	16
1	65536	16
0	65536	1048576

De exponent is verdubbeld maar het aantal stappen slechts met één toegenomen. Ook bij een veel grotere n neemt het aantal stappen nauwelijks toe. Bij een startwaarde van bijvoorbeeld $n = 120$ doorloopt n in achtereenvolgende stappen de waarden: 60, 30, 15, 14, 7, 6, 3, 2, 1 en 0. Dat betekent 10 vermenigvuldigingen. Het resultaat, ook bij een kleine K, zou een enorm getal zijn. 2^{120} bijvoorbeeld

heeft 16 bytes nodig (121 bits in feite). Decimaal is de uitkomst een getal van 36 cijfers.

5.4 Programma machtsverheffen

Het programma volgt nauwkeurig het algoritme en gebruikt voor het vermenigvuldigen de routine uit het eerste voorbeeld, zij het enigszins gewijzigd.

Programma H5P3 Machtsverheffen voor achtbits-getallen

```
;       Machtsverheffen
;
        LD      HL,EXPON        ;adres exponent
        LD      C,(HL)          ;exponent n in C
        INC     HL              ;adres grondtal K
        LD      D,(HL)          ;grondtal in D
        LD      E,1             ;in E resultaat Z=1
MLUS:   SRL     C               ;is n even ?
        JP      NC,EVEN         ;ja
        LD      H,D             ;nee, Z=Z*k
        LD      L,E             ;K in H; Z in L
        CALL    VERM
        LD      E,A             ;nieuwe Z in E
        XOR     A               ;A=0
        CP      C               ;n=0 ?
        JP      Z,MEIND         ;ja, naar einde
EVEN:   LD      H,D             ;K=K*K
        LD      L,D             ;K in H en L
        CALL    VERM
        LD      D,A             ;nieuwe K in D
        JP      MLUS            ;opnieuw
MEIND:  LD      A,E
        LD      (MACHT),A       ;berg resultaat op
        RET

EXPON:  DEFB    03
GROND:  DEFB    06
MACHT:  DEFB    00

;Subroutine vermenigvuldiging
;Aanroep met te vermenigvuldigen getallen in H en L.
;Resultaat bij terugkeer in A.
VERM:   LD      B,8             ;bitteller
        XOR     A               ;maak resultaat 0
VLUS:   SRL     H               ;schuif factor 1 rechts
        JP      NC,VEIND        ;is er een Carry
        ADD     A,L             ;ja, tel op
```

```
VEIND:  SLA     L               ;schuif factor 2 links
        DJNZ    VLUS
        RET

        END
```

We bespreken nu het gedeelte van het programma dat beslist welke van de vermenigvuldigingen, K × K of Z × K, moet worden uitgevoerd. Na SRL C is de Carry 1 als *n* oneven is. In een oneven getal is het laagste bit 1 en in een even getal is dit bit 0. SRL C deelt in feite *n* door 2. Dat is juist, zolang *n* even is. Is *n* oneven dan zijn er twee mogelijkheden. Ofwel *n* was voor de verschuiving gelijk aan 1 en is dus erna gelijk aan 0, wat betekent dat het machtsverheffen is gedaan. In het andere geval was *n* voor de verschuiving groter dan 1. Wat er dan eigenlijk moet gebeuren is het ongedaan maken van de deling door 2 en het verminderen van *n* met 1. Echter, een oneven *n* is na vermindering met 1 altijd even, dus moet de vermenigvuldiging K × K volgen.

n:	0000 1011	
na SRL C	0000 0101	Carry is 1

De deling door 2, de verschuiving dus, ongedaan maken en *n* met 1 verminderen levert op:

n:	0000 1010

Zouden we opnieuw testen of *n* even is, dan wordt *n*:

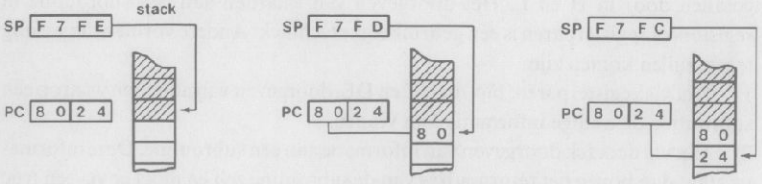
0000 0101

Deling door twee is nu precies wat er volgens het algoritme moet gebeuren. De waarde van *n* is nu gelijk aan die na de test waarbij *n* oneven was. In plaats van in dat geval de onterechte deling te herstellen, laten we als *n* niet 0 is altijd de vermenigvuldiging voor *n* = even volgen.
Een nog onbekende instructie staat in de opdracht CALL VERM. De instructie CALL met daarachter een adres of label heeft in principe dezelfde uitwerking als GOSUB in BASIC. Bij gebruik van GOSUB springt de interpreter naar het erachter opgegeven regelnummer waar een subroutine moet beginnen. Het einde van de subroutine is aangegeven met RETURN. Is de interpreter daar aangeland dan vindt een sprong terug plaats, naar de instructie volgend op GOSUB.
CALL (roep aan) en RET (van RETurn, keer terug) hebben in de assembleertaal van de Z-80 dezelfde uitwerking. CALL VERM springt naar de subroutine VERM. Bij het tegenkomen van de instructie RET wordt het programma voortgezet met de instructie na de CALL.

De precieze werking van een CALL-instructie is heel simpel. Na CALL staat het startadres van de subroutine of een label dat de assembler vertaalt in een adres. Tijdens het runnen van het programma vervangt de Z-80 de waarde van de PC door het adres na de CALL. Tot zover is alles precies eender als bij een JUMP-instructie. Het verschil met de JUMP-instructie is dat de Z-80 de oorspronkelijke inhoud van de PC moet bewaren. Uitvoering van de RET-instructie is niets anders dan deze oorspronkelijke inhoud weer in de PC terugzetten.
Deze waarde van de PC is het adres van de eerste instructie na de CALL.

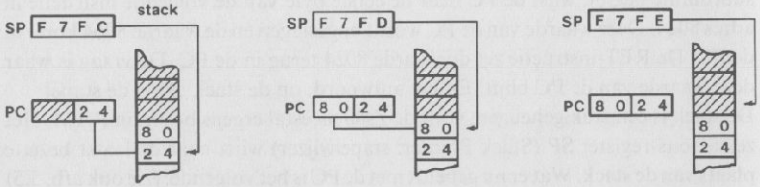
adres	inhoud	PC na ophalen byte
8021	CD (call)	8022
8022	64 (adres van)	8023
8023	80 (subroutine)	8024
8024		

Na het ophalen van de instructie CALL 8064, waarbij 8064 het adres is waarop de subroutine begint, wijst de PC naar de eerste byte van de volgende instructie in adres 8024. Deze waarde van de PC wordt opgeslagen en de waarde 8064 komt in de PC. De RET-instructie zet de waarde 8024 terug in de PC. De vraag is waar deze waarde van de PC blijft. En het antwoord: op de stack ofwel de stapel.
De stack is een stuk geheugen, voor de Z-80 meestal ergens boven in RAM. Het zestienbits-register SP (Stack Pointer: stapelwijzer) wijst naar de laatst bezette plaats van de stack. Wat er nu gebeurt met de PC is het volgende: (zie ook afb. 5.5) de Z-80 verlaagt de inhoud van SP en zet het msb (most significant byte, het belangrijkste en hoogste byte) van de PC in het adres waarnaar SP wijst. Vervolgens verlaagt de Z-80 de inhoud van SP nogmaals en zet het lsb (least significant byte, het minst belangrijke dus laagste byte) van de PC in het adres waarnaar SP nu wijst. Net als aan het begin van de instructie wijst SP naar de laatst bezette plaats van de stapel.



Afb. 5.5 Inhoud van de PC naar de stack

De stapel groeit van boven naar beneden. Naarmate er meer op de stapel komt, gaat de Stack Pointer naar steeds lagere adressen. De meeste assemblers zorgen ervoor dat de stapel op de juiste plaats begint. Het is mogelijk het begin van de stapel zelf vast te stellen. LD SP,HL bijvoorbeeld kopieert de inhoud van HL in SP. De oude waarde van SP moet voor het einde van het programma worden teruggezet en de oorspronkelijke stapel met return-adressen die de computer zelf nodig heeft mag niet zijn overschreven.
De stapel kan ook slinken. Door de RET-instructie zet de Z-80 de inhoud van het adres waarnaar de SP wijst in het lagere-orde-deel van de PC. De Z-80 verhoogt SP en zet de inhoud van het adres waarnaar SP nu wijst in het hogere-orde-deel van de de PC. Ten slotte verhoogt de Z-80 de inhoud van SP nogmaals zodat die weer naar de laatst bezette plaats van de stapel wijst. Zie afb. 5.6. PC en SP hebben na de subroutine dezelfde waarde als ervoor. Dezelfde gang van zaken vindt plaats bij de aanroep vanuit BASIC. Vandaar dat alle zo aangeroepen routines moeten eindigen met RET.



Afb. 5.6 Stack naar PC

Nu zou de vraag over kunnen blijven waar de Z-80 het in de CALL opgegeven adres laat voordat PC naar de stack gaat. De Z-80 slaat dit adres op in het al eerder genoemde WZ-register.
Bij het aanroepen van de subroutine met CALL geven we de te vermenigvuldigen getallen door in H en L. Het doorgeven van waarden aan een subroutine in registers of registerparen is een gebruikelijke techniek. Andere vormen die we nog tegen zullen komen zijn:
1) Het via registerparen, bijv. in HL en DE, doorgeven van adressen waarop een subroutine de nodige informatie kan vinden.
2) Het via de stack doorgeven van informatie aan een subroutine. Deze informatie staat dan boven het return-adres van de subroutine zelf en moet er via een truc worden afgehaald. De informatie kan bestaan uit data of adressen.

6 Delen en worteltrekken

6.1 Algemeen

De meeste assemblers staan labelnamen toe met een lengte van maximaal zes tekens. Het eerste teken moet een letter zijn en wat de eisen aan de overige tekens betreft, deze verschillen nogal per assembler. Soms zijn kleine letters toegestaan of leestekens. Indien de source code van een programma ook op andere assemblers moet zijn te gebruiken, is het verstandig voor labelnamen alleen hoofdletters en cijfers te gebruiken.
De leesbaarheid van een programma neemt aanzienlijk toe als labelnamen, net als mnemonics, duiden op hun functie. Vermijd namen die identiek zijn met mnemonics en namen die registers of registerparen aanduiden, zoals bijv. ADD, NZ of DE. In een geval als dit laatste zal LD A,(DE) registerpaar DE als adres gebruiken en niet het label.
Alle programma's in dit boek zijn voorzien van commentaar dat de werking verduidelijkt. Althans, dat hopen de auteurs. Bij het uitproberen van een programma hoeft men het commentaar uiteraard niet in te typen. Wél is het noodzakelijk zelf gemaakte programma's van uitleg te voorzien. Dat lijkt direct na het ontwerpen ervan overbodig, aangezien dan alles volmaakt duidelijk is. Een paar maanden later echter is men de werking ervan meestal helemaal vergeten. Moet dan bijvoorbeeld een eerder ontworpen subroutine in een programma worden gebruikt, dan kost het veel tijd om uit te zoeken hoe het ook al weer zat: met welke parameters en hoe het programma de subroutine moet aanroepen, hoe de subroutine het resultaat teruggeeft en welke restricties de subroutine stelt aan bijv. getallen.
Het commentaar dient de werking van het programma uiteen te zetten. Hoewel het hier ter verduidelijking wel eens gebeurt, is het niet nodig uit te leggen wat een instructie doet. Dit laatste behalve wanneer men om wille van snelheid en beknoptheid trucs gebruikt als XOR A of SUB A in plaats van LD A,0 of ADD A,A voor SLA A.

6.2 Het principe van de deling

Net als bij de vermenigvuldiging is het voor een goed begrip van de binaire deling nodig ervan doordrongen te zijn hoe de bewerking decimaal plaatsvindt. We voeren daartoe allereerst een staartdeling uit met als deler 21 en als deeltal 347.

```
21 / 347 \
```

Het proces verloopt als volgt. Neem het eerste cijfer van het deeltal en kijk of het deelbaar is door 21. Dat is niet het geval: 3 is niet deelbaar door 21. Neem dan de eerste twee cijfers van het deeltal en probeer het opnieuw.

```
21 / 347  \ 01
    21
    --  -
    13
```

34 is wél deelbaar door 21. Het gaat éénmaal en de rest is 13. De volgende stap is het derde cijfer achter de rest plaatsen en te kijken of het zo ontstane getal, 137, deelbaar is door 21.

```
21 / 347 \   016
    21
    --   -
    137
    126
    ---  -
     11
```

Dat gaat zesmaal. Van de deling is het quotiënt 16 en de rest 11. Net zo goed als vermenigvuldigen kan plaatsvinden door herhaald optellen, is het mogelijk te delen door herhaald aftrekken. Tot de rest kleiner is dan de deler kan men 21 zestienmaal aftrekken van 347. Nadeel van zulke methoden is het grote aantal handelingen dat nodig is om tot een antwoord te komen.
Een deling met binaire getallen gaat niet anders dan die met decimale getallen. Opnieuw voeren we een staartdeling uit met als deler het binaire getal 101 (=5 decimaal) en als deeltal 100110 (=38).

```
101 / 100110 \
```

Het eerste cijfer van het deeltal is kleiner dan 101, de eerste twee en de eerste drie ook. Pas van de eerste vier cijfers van het deeltal kan 101 worden afgetrokken.

```
101 / 100110 \0001
      101   -
      ---
      100
```

Voeg nu het vijfde getal van het deeltal aan de rest toe en probeer het opnieuw. Dan het zesde.

```
101 / 100110 \000111
      101   -
      ---
      1001
       101  -
       ---
       1000
        101 -
        ---
          11
```

Het quotiënt is 111 (= 7 decimaal) en de rest is 11 (= 3 decimaal). Niet schokkend maar wel correct.
Het algoritme om een deling met een microprocessor uit te voeren, volgt exact de staartdeling van hierboven. In principe werkt het als volgt. Schuif het deeltal 100110 naar links een andere locatie in (een geheugenplaats of een register) net zolang tot de deler (101) ervan kan worden afgetrokken.

	locatie	deeltal
start	000000	100110
na 1 ×	000001	001100
na 2 ×	000010	011000
na 3 ×	000100	110000
na 4 ×	001001	100000

Locatie geeft hier een geheugenadres of een register aan. Nadat vier cijfers van het deeltal naar de locatie zijn geschoven, is het mogelijk geworden om de deler (101) af te trekken. We hebben dan precies dezelfde situatie als in de staartdeling. Na aftrekking van 101 blijft in locatie 100 over. De vraag is hoe we nu aangeven dat er een aftrekking heeft plaatsgevonden, met andere woorden hoe we het quotiënt opbouwen. Het eenvoudigst gaat dat door het deeltal met één te verhogen.

	locatie	deeltal
na aftrekking	000100	100001
na 5 ×	001001	000010

Opnieuw is aftrekking mogelijk. Over blijft weer 100 en het deeltal wordt met één verhoogd. Aangezien het deeltal zes cijfers telde, moeten er zes verschuivingen plaatsvinden om het geheel naar de locatie te brengen.

	locatie	deeltal
na aftrekking	000100	000011
na 6×	001000	000110

Na de laatste aftrekking en verhoging van het deeltal:

locatie	deeltal
000011	000111

In de locatie staat de rest en het deeltal is vervangen door het quotiënt. Let, bij bestudering van de gang van zaken, vooral op de manier waarop het quotiënt zich vormt.

6.3 Programma's voor deling

Het eerste programma volgt rechttoe rechtaan de hierboven besproken werkwijze. Het deeltal komt in register L en schuift dan bit voor bit links register H in. Na elke verschuiving wordt gekeken of het mogelijk is de deler van de inhoud van H af te trekken.
Het verschuiven van het deeltal van register L naar register H gebeurt niet door schuiven en roteren maar met een zestienbits rekenkundige instructie ADD HL,HL. De inhoud van H is bij aanvang 0. ADD HL,HL telt de inhoud van HL op bij de inhoud van HL en zet het resultaat daarvan in HL. Het effect is hetzelfde als tweemaal de inhoud van HL ofwel een verschuiving daarvan naar links.
Het is, zoals in het vorige hoofdstuk al is gezegd, bij de Z-80 in beperkte mate mogelijk rekenkundige instructies uit te voeren met de zestienbits-registers. HL speelt daarbij dezelfde rol als de accumulator bij het achtbits-rekenen: voor de bewerking is de inhoud ervan één van de operanden en erna staat het resultaat erin. De volledige vorm is ADD HL,ss waarbij voor ss kan worden ingevuld BC, DE, HL en SP. De zestienbits rekenkundige instructies dienen vaak om adressen uit te rekenen. Staat bijv. het basisadres van een array in DE en de offset in HL dan geeft ADD HL,DE het adres van het element. Stel het basisadres van ARRAY = 2000H en we willen element 25 laden, dus ARRAY(25):

basis ARRAY in DE	= 2000H	
element 25: in HL	= 0019H	+
na ADD HL,DE in HL	2019H	

Voorgaande geldt als elk element maar één byte in beslag neemt. Is elk element twee bytes groot dan moet ADD HL,HL eerst de inhoud van HL met twee vermenigvuldigen voor de optelling plaatsvindt.
Zestienbits-aftrekken is eveneens mogelijk evenals rekenkundige bewerkingen met de indexregisters IX en IY.

Programma H6P1 achtbits-deling

```
;Deling 8 bits / 8 bits
;
;Het programma gebruikt de registers A, B, C, H en L

        LD      A,(DEELT)
        LD      L,A             ;deeltal in HL
        LD      H,0
        LD      A,(DELER)
        LD      C,A             ;deler in C
        LD      B,8             ;bitteller
DLUS:   ADD     HL,HL           ;schuif deeltal naar H
        LD      A,H             ;kan deler worden
        SUB     C               ; afgetrokken ?
        JP      C,DEIND         ;nee, naar einde lus
        LD      H,A             ;ja, rest in H
        INC     L               ;verhoog quotient
DEIND:  DJNZ    DLUS            ;alle bits gedaan ?
        LD      A,L             ;ja, berg quotient op
        LD      (QUOT),A
        LD      A,H             ;evenals de rest
        LD      (REST),A
        RET

DEELT:  DEFB    171
DELER:  DEFB    12
QUOT:   DEFB    00
REST:   DEFB    00

        END
```

Het deeltal komt in het laagste byte van HL, de deler in register C. Als teller dient, zoals gebruikelijk, B. We doorlopen eenmaal de gehele lus. DEELT is bij aanvang 171 en DELER is 12. ADD HL,HL geeft tweemaal de inhoud van HL, wat hetzelfde is als een verschuiving naar links.

HL = 0000 0000 1010 1011 (171 decimaal)
na ADD HL,HL 0000 0001 0101 0110

LD A,H plaatst de inhoud van H in de accumulator. SUB C trekt vervolgens de deler af van de inhoud van A. Ontstaat hierbij een carry dan is de inhoud van H kleiner dan de deler. Er kan niet worden afgetrokken en het programma springt naar het einde van de lus DEIND.
Is er geen carry dan is de inhoud van H groter dan de deler en moet er een aftrekking plaatsvinden. Het resultaat daarvan staat al in de accumulator. LD H,A plaatst dit in H.
INC L ten slotte zet een 1 in het laagste bit van het quotiënt.
We doorlopen nu de lus achtmaal en geven daarbij de inhoud van HL binair weer. Als een aftrekking mogelijk is, zullen we deze ook uitvoeren.

1e	HL =	0000	0001	0101	0110
2e		0000	0010	1010	1100
3e		0000	0101	0101	1000
4e		0000	1010	1011	0000
5e		0001	0101	0110	0000

Aftrekking van de deler, 0000 1100, is nu mogelijk.

0001	0101	inhoud H
0000	1100	– deler
0000	1001	

Aan het einde van de lus is de inhoud van HL:

5e	HL =	0000	1001	0110	0001
6e		0001	0010	1100	0010

Aftrekking is mogelijk.

0001	0010	inhoud H
0000	1100	– deler
0000	0110	

6e	HL =	0000	0110	1100	0011
7e		0000	1101	1000	0110

Aftrekking is mogelijk.

```
          0000  1101   inhoud HL
          0000  1100   deler
          ---------  -
          0000  0001

7e   HL = 0000  0001  1000  0111
8e        0000  0011  0000  1110
```

Het resultaat staat nu in L: 0000 1110, decimaal 14. De rest is te vinden in H: 0000 0011, decimaal 3.
Deling door 0 is niet toegestaan. Het quotiënt zou in dat geval FF hexadecimaal (binair 1111 1111) zijn, aangezien aftrekking steeds mogelijk is. Het is echter niet moeilijk een foutroutine toe te voegen die het geval deler = 0 detecteert en een foutmelding geeft. Het programma komt er dan zo uit te zien.

Programma H6P2 achtbits-deling met foutmelding voor deling door 0

```
;Deling 8 bits / 8 bits met foutmelding
;
;Het programma gebruikt de registers A, B, C, H en L

        LD      A,(DEELT)
        LD      L,A             ;deeltal in HL
        XOR     A               ;A=0
        LD      H,A
        LD      (FOUT),A        ;zet foutnummer op 0
        LD      A,(DELER)
        CP      H               ;is deler 0 ?
        JP      Z,DNUL          ;ja, naar foutroutine
        LD      C,A             ;deler in C
        LD      B,8             ;bitteller
DLUS:   ADD     HL,HL           ;schuif deeltal naar H
        LD      A,H             ;kan deler worden
        SUB     C               ; afgetrokken ?
        JP      C,DEIND         ;nee, naar einde lus
        LD      H,A             ;ja, rest in H
        INC     L               ;verhoog quotient
DEIND:  DJNZ    DLUS            ;alle bits gedaan ?
        LD      A,L             ;ja, berg quotient op
        LD      (QUOT),A
        LD      A,H             ;evenals de rest
        LD      (REST),A
        RET

DEELT:  DEFB    171
DELER:  DEFB    12
QUOT:   DEFB    00
```

```
REST:   DEFB    00
FOUT:   DEFB    00

;Foutroutine
DNUL:   LD      A,3             ;foutnummer
        LD      (FOUT),A
        RET

        END
```

CP H kijkt of de deler 0 is en gebruikt daarvoor het zojuist met 0 geladen register H. Is de deler inderdaad gelijk aan 0 dan voert het programma de instructies vanaf label DNUL uit. FOUT krijgt daarbij de waarde 3. Al is dit foutnummer hier willekeurig, het is gebruikelijk om voor een bepaalde fout een nummer te reserveren. Een aantal bijeengevoegde rekenprogramma's, bijv. optellen en delen, kunnen gebruik maken van hetzelfde label FOUT. Deling door 0 geeft de inhoud van FOUT de waarde 3 en een carry bij de optelling, bijv. de waarde 2. Na de berekening kan men dan de inhoud van FOUT onderzoeken. Springt het rekenprogramma terug naar BASIC en is FOUT adres 823A dan zou u een BASIC-programma als hieronder kunnen gebruiken.

```
:
:
110 FOUT=33338:REM =823A DECIMAAL
120 IF PEEK(FOUT) <> 0 THEN PRINT ''FOUTNR
'';PEEK(FOUT)
```

De BASIC-interpreter gebruikt net zo'n systeem om aan te geven door welke fout het programma werd onderbroken. Het foutnummer kan ook dienen om uitgebreidere informatie omtrent de fout te selecteren.

```
120 IF PEEK(FOUT)=3 THEN PRINT''DELING DOOR 0''
```

In de source code van het programma staan twee return-instructies. In het programma wordt er daarvan maar één uitgevoerd. Welke dat is, hangt af van het al dan niet nul zijn van de deler.

6.4 Subroutine voor de deling

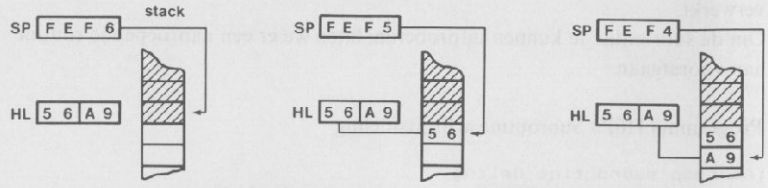
De subroutine voor de deling volgt de algemene aanpak waarover in het vorige hoofdstuk is gesproken. Aanroep ervan gebeurt met de adressen van deeltal en deler in HL en DE. Het adres van het deeltal staat in HL. De subroutine vervangt

de inhoud ervan door het quotiënt. In DE bevindt zich na RET het adres van de rest.
Een subroutine moet, met een CALL, vanuit een programma worden aangeroepen en kan niet als zelfstandig programma worden gebruikt.
Het uitvoeren van de deling gebeurt op dezelfde manier als in het vorige programma. De gedeeltes voor en na de eigenlijke deling wijken uiteraard af vanwege de manier waarop de te delen getallen toegevoerd en de resultaten worden verwerkt.
Om de subroutine te kunnen uitproberen, laten we er een aanroepende routine aan voorafgaan.

Programma H6P3 Subroutine achtbits-deling

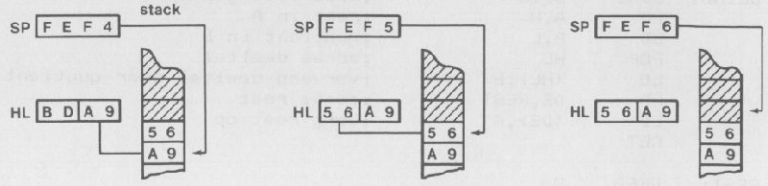
```
;Aanroep subroutine deling
        LD      HL,DEELT        ;adres deeltal
        LD      DE,DELER        ;adres deler
        CALL    DEEL
        RET

DEELT:  DEFB    171
DELER:  DEFB    12

;Subroutine delen
;Bij aanroep in HL adres deeltal, in DE adres deler.
;De subroutine vervangt het deeltal door het
;quotient en geeft het adres van de rest terug in DE.
DEEL:   PUSH    HL              ;bewaar adres deeltal
        LD      A,(DE)
        LD      E,A             ;deler in E
        XOR     A               ;A=0
        LD      L,(HL)
        LD      H,A             ;in HL deeltal
        LD      B,8             ;bitteller
DLUS:   ADD     HL,HL           ;schuif deeltal naar H
        LD      A,H             ;kan deler worden
        SUB     E               ; afgetrokken ?
        JP      C,DEIND         ;nee, naar einde lus
        LD      H,A             ;ja, in H rest
        INC     L               ;verhoog quotient
DEIND:  DJNZ    DLUS            ;alle bits gedaan ?
        LD      A,H             ;rest in A
        LD      B,L             ;quotient in B
        POP     HL              ;adres deeltal
        LD      (HL),B          ;vervang deeltal door quotient
        LD      DE,REST         ;adres rest
        LD      (DE),A          ;berg rest op
        RET

REST:   DEFB    00

        END
```

PUSH HL zet het adres van het deeltal op de stack, net zoals de inhoud van de PC door een CALL naar de stack ging. PUSH betekent duw. De SP wijst, zoals bekend, naar de laatst bezette plaats van de stapel. De Z-80 verlaagt de inhoud van SP en zet het msb van HL, dus de inhoud van H, in het adres waarnaar SP wijst. Zie afb. 6.1. De Z-80 verlaagt de SP nogmaals en nu gaat de lsb van HL, de inhoud van L, naar het adres waarnaar SP wijst.



Afb. 6.1 Een PUSH-operatie: PUSH HL

Registerpaar HL zelf kan vervolgens dienen als opslag voor deeltal en quotiënt, zoals dat ook in het vorige deelprogramma gebeurde. POP HL aan het einde van het programma haalt het oorspronkelijke adres van het deeltal weer van de stack. POP betekent trek. POP HL haalt de laatste twee bytes van de stapel en zet ze in HL. Zie afb. 6.2. De Z-80 zet de inhoud van het adres waarnaar SP wijst in L. Ten slotte verhoogt de Z-80 de SP en zet de inhoud van het adres waarnaar SP wijst in H. Ten slotte verhoogt de Z-80 nogmaals de SP zodat deze weer naar de laatst bezette plaats van de stapel wijst.
Op deze manier wordt de stack gebruikt als tijdelijke opslag. Het zal duidelijk zijn dat er altijd evenveel op de stapel zal moeten worden gezet als er van wordt afgehaald, anders blijven er gegevens op de stack staan die door de RET-instructie in de PC worden geladen, of het werkelijke return-adres wordt van de stack gehaald voor RET dat kan doen.



Afb. 6.2 Een POP-operatie: POP HL

Volledige notaties zijn PUSH *qq* en POP *qq*, waarin *qq* kan zijn: AF, BC, DE en HL. Hiernaast zijn mogelijk PUSH IX, PUSH IY en POP IX, POP IY. Stapelmanipulaties gaan bij de Z-80 altijd met 16 bits tegelijk. Na PUSH HL hoeft niet per se POP HL te volgen. Alle genoemde registerparen zijn toegestaan. PUSH BC gevolgd door POP HL zet de waarde van BC via de stack in HL.
Wat het laatste op de stack gaat, komt er het eerst weer af. Dit noemt men een Last In First Out structuur, afgekort LIFO.
Terugkerend naar het programma zien we dat de stack, zoals het hoort, door de subroutine niet is veranderd.
Aanroep van de subroutine vanuit het hoofdprogramma of vanuit een andere subroutine met CALL DEEL zet de inhoud van de PC, die dan naar de instructie volgend op CALL DEEL wijst, op de stack. PUSH HL zet de inhoud van HL op de stack en POP HL verwijdert deze. Ten slotte haalt RET het adres van de instructie volgend op CALL DEEL in het aanroepende programma van de stack en plaatst deze in de PC.

STACK na:

CALL DEEL	PUSH HL	POP HL	RET
return-adres	return-adres	return-adres	
	adres deeltal		

In feite verandert een RET- of POP-instructie niet de inhoud van de stack maar slechts die van de Stack Pointer. Het adres van het deeltal blijft op de stack staan maar de inhoud van de Stack Pointer SP is verhoogd zodat deze naar het return-adres wijst.
Een PUSH- of CALL-instructie wijzigt de Stack Pointer plus de inhoud van de stack.
Na PUSH HL laadt het programma het deeltal in register E en de deler in HL, waarbij H nul is.
Als de deling in de lus is uitgevoerd, staat het quotiënt in L en de rest in H. Deze twee waarden worden tijdelijk opgeslagen in de registers B en A. De rest moet daarbij in A staan. De routine keert terug met het adres van de rest in DE. Voor deze rest is een byte gereserveerd aan het einde van het programma. Als het adres hiervan in DE staat, kan alleen de inhoud van A daarin worden geladen. Andere mogelijkheden nemen meer tijd en geheugenruimte in beslag. Naar het adres in HL, dat van het oorspronkelijke deeltal, kan de inhoud van elk register worden gekopieerd.

De mogelijkheden zijn:

LD A,(BC) LD A,(DE) LD (BC),A LD (DE),A

Voor HL is dat LD *r*,(HL) en LD (HL), *r* waarbij *r* elk achtbits-register kan zijn: A, B, C, D, E, H en L.

7 Het 2-complement

7.1 Positieve en negatieve getallen

Tot dusver rekenden de programma's in dit boek met positieve getallen van 0 t/m 255. Voor de meeste toepassingen is dit echter een te beperkte reeks.
Grotere getallen nemen meer ruimte in beslag. Een tweebytes- ofwel zestienbits-formaat levert getallen op van 0 t/m 65.535. Verwerking hiervan is mogelijk in registerparen. Het verdelen over meer registers of registerparen leidt uiteraard tot nog grotere getallen. Een vierbytes- of 32-bits-formaat geeft een reeks van 0 t/m $2^{32}-1$. Dit laatste getal heeft de decimale waarde 4.294.967.295.
Er is hier echter alleen sprake van positieve getallen, terwijl programma's meestal ook negatieve waarden moeten aankunnen. De vraag is hoe aan te geven dat het getal positief dan wel negatief is. Eén mogelijkheid is het opnemen van een extra byte, waarin zich de ASCII-code voor '+' of '−' bevindt. Een vermenigvuldigingsprogramma kan dan als beide getallen bijvoorbeeld negatief zijn het produkt de ASCII-code '+' meegeven. Er bestaat echter een standaardmethode voor het onderscheiden van positieve en negatieve getallen: het 2-complement. In grote lijnen komt het erop neer dat een deel van de positieve getallen als negatief getal wordt beschouwd. Hoe dat kan, komt in dit hoofdstuk aan de orde.

7.2 De eindigheid van de getallenreeks

In hoeveel bits we de getallen ook weergeven, altijd is er een groot getal dat nog binnen het gekozen formaat past. In een achtbits-formaat is dat 255. Stel de inhoud van de accumulator is bij aanvang 0. Een steeds herhaalde instructie INC A verhoogt deze tot 255. Een volgende verhoging maakt de inhoud weer 0 en set de zero-vlag.
In tegenstelling tot de reeks natuurlijke getallen waarbij verhoging met één – hoe groot een getal ook is – altijd kan, is de binnen een bepaald formaat weer te geven reeks als eindig te beschouwen. Dit verschil, alsmede het weer op 0 springen bij overschrijding van het maximum vormen de essentie van het 2-complement.
Om wat minder abstract te zijn, nemen we als voorbeeld een alledaags eindig

systeem, namelijk de kilometerteller van een auto. Om de getallen simpel te houden, heeft de denkbeeldige teller maar twee cijfers. Minimum- en maximumstand zijn dus 00 en 99 kilometer. Wie een dergelijk karretje tweedehands koopt, weet heel goed dat bij een tellerstand van zeg, 42, het werkelijk aantal gereden kilometers 42, 142, 242, 342 enz. kan zijn. Staat namelijk de teller op 99 dan is na het rijden van nog een kilometer de stand weer 00. Is het aantal afgelegde kilometers 542 dan is de tellerstand de rest van 542/100. Berekening van dit quotiënt levert 5 rest 42 op. De rest is hetzelfde bij een stand van 642, 742 enz. Deze 42, als rest van 42, 142, 242, 342 enz. na deling door 100 heet een restklasse. Het deeltal, in dit geval 100, heet de modulus. Voor de reeks mogelijke tellerstanden, 00 t/m 99, is de benaming: restklassensysteem modulo 100.
Een flink aantal BASIC-dialecten en de meeste andere talen kennen een modulobewerking, namelijk MOD. Deze geeft de rest bij deling van twee integer-getallen. Bijvoorbeeld 14 MOD 3 = 2.
Om wat simpele berekeningen uit te proberen, nemen we een restklassensysteem modulo 10, ofte wel een kilometerteller met één cijfer. De mogelijke stand van de teller varieert van 0 t/m 9. Stel dat bij aanvang van een tochtje de teller op 8 staat en we 4 kilometer rijden. De nieuwe stand berekenen we door optelling van 4 en 8 op heel simpele wijze, namelijk door 8 viermaal met 1 te verhogen. De eerste keer levert dat 9 op, de tweede keer 0. De derde en vierde verhoging brengen de stand op respectievelijk 1 en 2. Een bewerking modulo 10 op:

$8 + 4 = 12$
$12 \text{ MOD } 10 = 2$

In diagram 7.1 staan alle mogelijke optellingen voor dit systeem met hun uitkomsten. Wie 3 en 4 wil optellen, neemt de kolom waar 4 boven staat en de rij waar het getal 3 naast staat. De som staat op het kruispunt van rij en kolom. Zo is de som van 6 en 7 gelijk aan 3.

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

Diagram 7.1 Optellen in een restklassensysteem modulo 10

$6 + 7 = 13$
$13 \text{ MOD } 10 = 3$

Van linksonder naar rechtsboven loopt een rij nullen. Dit zijn de uitkomsten van optellingen die 10 opleveren.
Eerder is opgemerkt dat de truc om in een dergelijk systeem negatieve getallen op te nemen eruit bestaat een deel van de positieve getallen als negatief te beschouwen. Aangezien in ons systeem $8 + 2$ nul oplevert, vormen 8 en 2 hier elkaars complement zoals in het meer vertrouwde stelsel 8 en -8, 2 en -2 elkaars complement zijn.
Om negatieve getallen te creëren, beschouwen we nu 9 als -1, 8 als -2, 7 als -3 en 6 als -4. Elk van deze getallen levert, opgeteld bij het complement ervan, 0 op. De som van 4 en -1 geeft als uitkomst 3.

$4 + -1 = 4 + 9 = 13$
$13 \text{ MOD } 10 = 3$

Dit is correct. In de fysieke wereld van de kilometerteller betekent de optelling van 4 en -1 dat de beginstand 4 is en dat we die negenmaal met 1 verhogen. Een probleem is of het getal 5 negatief dan wel positief is, dus $+5$ of -5. In beide gevallen heeft het geen complement. We kiezen voor -5 en geven in diagram 7.2 het resultaat van alle optellingen die binnen het systeem mogelijk zijn.

+	0	1	2	3	4	-5	-4	-3	-2	-1
0	0	1	2	3	4	-5	-4	-3	-2	-1
1	1	2	3	4	-5	-4	-3	-2	-1	0
2	2	3	4	-5	-4	-3	-2	-1	0	1
3	3	4	-5	-4	-3	-2	-1	0	1	2
4	4	-5	-4	-3	-2	-1	0	1	2	3
-5	-5	-4	-3	-2	-1	0	1	2	3	4
-4	-4	-3	-2	-1	0	1	2	3	4	-5
-3	-3	-2	-1	0	1	2	3	4	-5	-4
-2	-2	-1	0	1	2	3	4	-5	-4	-3
-1	-1	0	1	2	3	4	-5	-4	-3	-2

Diagram 7.2 Optellen van positieve en negatieve getallen

Zolang de getallen klein zijn, is de uitkomst correct. Zodra echter 2 grote positieve of 2 grote negatieve getallen worden opgeteld, overschrijden we de door onszelf gelegde grens tussen positief en negatief. 2 + 2 geeft netjes 4 maar 4 + 2 hoort 6 op te leveren, wat in onze zienswijze −4 voorstelt. Dit overschrijden van de grens waardoor het teken verandert en de uitkomst niet meer klopt, heet in het Engels overflow. De Z-80 heeft een speciale vlag om dit tijdens een berekening te signaleren. Behalve bij het optellen van twee getallen met gelijk teken treedt overflow ook op bij aftrekking van getallen met ongelijk teken:

$$4 - -2 = 4 + 2 = 6 (= -4).$$

In diagram 7.2 zijn de gebieden waarin van overflow sprake is aangegeven met driehoeken.

7.3 Het 2-complement

We keren terug naar het achtbits-formaat en verdelen de beschikbare getallen, van 0 t/m 255, in een positief en negatief deel. Het ligt voor de hand 1 t/m 127 als positief te nemen. De reeks 255 t/m 129 vormt dan de getallen van −1 t/m −127. Zoals in het vorige systeem met 5 het geval was, rijst nu de vraag of 128 positief of negatief moet zijn. We kiezen voor negatief en wel om de volgende reden. Beschouw de getallen van 0 t/m 255 binair:

0	0000 0000	
1	0000 0001	
:	:	
:	:	
126	0111 1110	
127	0111 1111	
128	1000 0000	= −128
129	1000 0001	= −127
130	1000 0010	= −126
:	:	:
:	:	:
254	1111 1110	= −2
255	1111 1111	= −

Van alle negatieve getallen is het hoogste bit 1, van alle positieve getallen is het hoogste bit 0. Dit is een heel gemakkelijk te gebruiken onderscheid. De Z-80 heeft een tekenvlag waarin na rekenkundige bewerkingen het hoogste bit staat van het

resultaat in de accumulator. Er kunnen nu voorwaardelijke sprongen of calls worden gemaakt:

```
JP M,routine
JP P,routine
```

In het eerste geval springt het programma naar een routine als het teken min en het hoogste bit dus 1 is, in het tweede geval als het plus is. CALL M,subroutine en CALL P,subroutine werken op dezelfde manier.
De getallen +127 en −128 als grootste positieve en negatieve getallen komen misschien bekend voor. Ze kwamen eerder ter sprake bij de relatieve sprong JR. Bij vertaling van deze instructie ‚JR LABEL, zet de assembler het label niet om in een zestienbits-adres, zoals bij de absolute sprong, maar in een achtbits-getal dat opgeteld bij de inhoud van de PC het adres van label oplevert. De Z-80 hanteert hier zelf het 2-complement formaat. Het achtbits-getal, ook wel de verplaatsing genoemd, maakt de inhoud van de PC maximaal 127 groter of 128 kleiner.
De Z-80 heeft ook een instructie om een positief getal om te zetten in een negatief getal en omgekeerd. Het is de instructie:

```
NEG
```

NEG staat voor NEGate, maak negatief, en bewerkt op de volgende manier de inhoud van de accumulator.

inhoud A	1111 1100	=252 (−4)
na NEG	0000 0100	=4

Rest nog de vraag waarom deze notatie het 2-complement heet. Dit is ter onderscheid van het eveneens bestaande 1-complement. Het nemen van het 1-complement is een handeling waarbij alle bits in een byte worden geïnverteerd; 0 wordt 1 en andersom. De Z-80 heeft ook hiervoor een instructie en wel CPL (ComPLement). Net als NEG bewerkt deze de inhoud van de accumulator.

inhoud A	1111 1100
na CPL	0000 0011

Het is, via het 1-complement, mogelijk het 2-complement van een binair getal snel te berekenen. Hiervoor neemt men het 1-complement van het getal en telt daar 1 bij op.
Het 2-complement van een getal is dus hetzelfde als het nemen van het 1-complement plus de extra handeling van een verhoging met 1.

getal	0000 0100	= 4
1-complement	1111 1011	= 251
+1	0000 0001 +	
2-complement	1111 1100	= 252 (−4)

Denk niet dat dit een soort getallenmagie is. Complement betekent: gedeelte dat ontbreekt om iets volledig te maken. Als het bijv. over rechte hoeken gaat, is 60 graden het complement van 30 graden. Zo is het 1-complement de aanvulling om alle bits 1 te maken.
Het 1-complement van een getal nemen, dus het inverteren van alle bits, is binnen een bepaald formaat hetzelfde als het getal aftrekken van het grootste getal dat in dat formaat mogelijk is.

grootste getal	1111 1111	= 255
	0000 0100 −	= 4
	1111 1011	= 1-complement

Het 1-complement van 4 is blijkbaar 255 − 4. Het 2-complement vinden we door daarbij 1 op te tellen, wat in feite betekent:

$$255 - 4 + 1 = 256 - 4$$

Dit was ook de uitdrukking voor het 2-complement. In het voorbeeld van de kilometerteller met één cijfer is het grootste getal 9. Het 1-complement van 4 is daar 9 − 4 = 5. We vinden het 2-complement door daarbij 1 op te tellen.
Voor het berekenen van het 2-complement in het tientallig stelsel hoeft bij een éénbyte-formaat het desbetreffende getal alleen van 256 te worden afgetrokken:

$$-4 = 256 - 4 = 252$$
$$-31 = 256 - 31 = 225$$

En om omgekeerd uit te rekenen welk negatief getal bijv. 189 voorstelt, volgen we dezelfde weg. Dat is te zeggen, we trekken van het getal 256 af:

$$189 - 256 = -67$$

In een meerbytes-formaat is de methode hetzelfde. In een zestienbits-formaat passen getallen van 0 t/m 65.535. Verdeeld in positief en negatief is dat 0 t/m 32.767 en 0 t/m −32.768. Alle negatieve getallen hebben een most significant bit van 1. Berekening van het 2-complement in een achtbits-formaat:

2^8 − getal = 256 − getal

Berekening van het 2-complement in een zestienbits-formaat:
2^{16} − getal = 65.536 − getal

7.4 Overflow

Overflow kan optreden bij optelling van getallen met hetzelfde teken en bij aftrekking van getallen met verschillend teken. In deze beide gevallen bestaat de mogelijkheid dat het resultaat groter is dan het maximale positieve of negatieve getal en dus van teken verandert.

+121	0111 1001		binair 121
+100	0110 0100	+	binair 100
−35	1101 1101		binair 221

Beschouwd als binaire optelling is het resultaat correct. Maar omdat alles boven de 127 als negatief is bestempeld, klopt het antwoord niet in de 2-complementsnotatie.

−106	1001 0110		binair 150
−88	1010 1000	+	binair 168
62	(1) 001 1110		binair 318

De ontstane carry meegerekend, klopt het resultaat als we de binaire optelling bekijken. In de 2-complementsnotatie is het teken echter veranderd. Het optreden van een carry wil niet altijd zeggen dat het 2-complementsresultaat onjuist is.

−10	1111 0110		binair 246
−12	1111 0100	+	binair 244
−22	(1)1110 1010		binair 490

Ondanks de carry is het resultaat juist. Overflow, de verandering van teken, ontstaat onder de volgende omstandigheden:

1) Er is een carry tussen bit 6 en bit 7 maar niet tussen bit 7 en het Carry-bit
2) Er is geen carry tussen bit 6 en bit 7 maar wel tussen bit 7 en het Carry-bit

Dit ongelijk aantal in- en uitgaande carry's in het hoogste bit veroorzaakt de tekenverandering. De bovengenoemde overflow-situaties signaleert de Z-80 in de P/O (parity/overflow)-vlag. Na logische instructies, rotaties en verschuivingen is

de vlag geset bij even pariteit en is er even aantal bits 1. Na rekenkundige instructies is de vlag geset bij overflow. Er is echter maar een stel mnemonics voor deze dubbel gebruikte vlag. Om bij gesignaleerde overflow naar een routine te springen die bijvoorbeeld een waarschuwing op het scherm zet dient de instructie:

```
JP PE,label
```

waarbij PE staat voor Parity Even. Indien er gesprongen moet worden als er geen overflow is dan is de vlag, evenals bij oneven pariteit (Parity Odd), niet geset (JP PO, label). Bij aftrekking van getallen met verschillende tekens ontstaat overflow op dezelfde manier.

−90	1010 0110		binair 166
100	0110 0100	−	binair 100
66	0100 0010		binair 66

Er is een carry van bit 7 naar bit 6 maar niet van het Carry-bit naar bit 7.

7.5 Optellen, aftrekken, vermenigvuldigen en delen

Op heel eenvoudige algebraïsche wijze willen we laten zien wat, bij gebruik van de 2-complementsnotatie, het resultaat is van bovengenoemde bewerkingen.
We gebruiken getal a en het negatieve getal b, in 2-complementsnotatie voorgesteld door 256-b. Een optelling kunnen we dan als volgt weergeven:

$$a + (256 - b) = 256 + (a - b)$$

Is a gelijk aan b dan is $(a - b)$ gelijk aan 0. 256 valt weg in de carry en het resultaat is 0.
Is $a > b$ dan is $(a - b)$ positief en valt 256 weg in de carry. Het resultaat is $a - b$.
Is $a < b$ dan is $(a - b)$ negatief. Men kan in dit geval schrijven $256 - (b - a)$ waarin $(b - a)$ positief is. Het eindresultaat is een negatief getal in de 2-complements-notatie.
Het optellen van twee negatieve getallen geeft het volgende resultaat:

$$(256 - a) + (256 - b) = 256 + 256 - (a + b)$$

Hierbij is $256 - (a + b)$ het 2-complement van de som. De overblijvende 256 valt weg in de carry. Deze overblijvende factor 256 heeft in feite in een achtbits-formaat geen betekenis voor het resultaat omdat deze factor op een negende bit betrekking heeft.

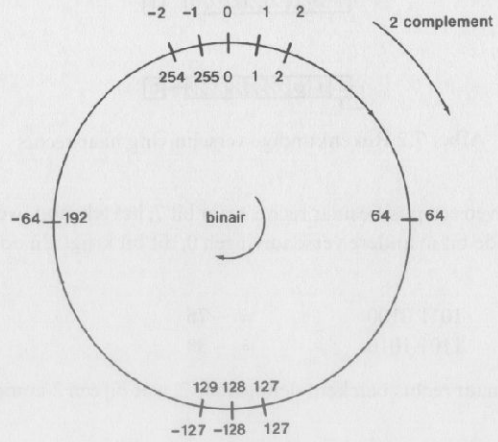
Het aftrekken laat zich op een soortgelijke wijze behandelen. Interessanter is de vraag wat er gebeurt bij vermenigvuldigen en delen. We beginnen met het eerste.

$a \times (256 - b) = a \times 256 - a \times b$

Dit komt overeen met:

$(a - 1) \times 256 + 256 - a \times b$

Daarin is $256 - a \times b$ de 2-complementsnotatie van het produkt. De factor $(a - 1) \times 256$ valt weg als a gelijk is aan 1. Voor $a > 1$ belandt deze factor in het negende bit of hoger. Is a gelijk aan 0 dan is het produkt 0.
De 2-complementsgetallen kan men zich, zoals in het voorbeeld met de kilometerteller al ter sprake kwam, als een cyclische getallenreeks voorstellen (zie afb.7.1). De getallen binnen de cirkel stellen de binaire waarden voor, de getallen erbuiten de 2-complementswaarden. Binaire en 2-complementsgetallen zijn in achtbitsformaat in het bereik van 0 t/m 127 identiek met elkaar. Binaire waarden van 128 t/m 255 stellen de 2-complementsgetallen van -128 t/m -1 voor.
De bij vermenigvuldiging ter sprake gekomen factor $(a - 1) \times 256$ wil bij deze voorstelling zeggen dat het produkt $(a - 1) \times$ de cirkel rond is gegaan, wat op het eindresultaat geen invloed heeft. Het uiteindelijk produkt, $256 - a \times b$, is een negatief getal, te vinden door in de getallencirkel vanaf 0 tegen de klok in de factor $a \times b$ af te meten. Overschrijdt men daarbij de -128-grens dan treedt overflow op.



Afb. 7.1 Cyclische getallenreeks

Vervolgens het produkt van twee negatieve getallen.

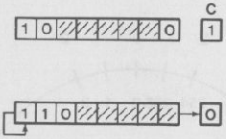
$(256 - a) \times (256 - b) =$
$256^2 - 256 \times (a + b) + a \times b$

Hierbij blijft, geheel correct, een positief produkt over.

Met de deling ligt het problematischer.

$(256 - a) / b = 256 / b - a / b$

Ten onrechte wordt hier ook 256 door b gedeeld. Het juiste quotiënt is $256 - a / b$. Deling van getallen in 2-complementsnotatie geeft altijd foutieve resultaten. We zullen dan ook zien dat bij de 32-bits-berekeningen verderop in dit boek voor het delen speciale voorzieningen worden getroffen. Op zich zijn deze heel simpel. Het programma maakt een eventuele negatieve deler of deeltal positief en voert dan pas de deling uit. Aan de hand van de tekens van deler en deeltal bepaalt het tevens of het quotiënt positief of negatief moet zijn. In dat laatste geval zet het programma het quotiënt weer om in 2-complementsnotatie.
De Z-80 kent nog een speciale schuifinstructie die rekening houdt met de 2-complementsnotatie. Dit is SRA *m* waarbij *m* kan zijn: *r*, (HL), (IX + *d*) en (IY + *d*). *r* is één van de registers A, B, C, D, E, H of L. SRA staat voor Shift Right Arithmetic, een rekenkundige verschuiving naar rechts. Zie afb. 7.2.



Afb. 7.2 Rekenkundige verschuiving naar rechts

Alle bits schuiven een positie naar rechts maar bit 7, het tekenbit, wordt niet zoals het vrijkomende bit in andere verschuivingen 0; dit bit krijgt z'n oorspronkelijke waarde terug.

	1011 0100	$= -76$
na SRA	1101 1010	$= -38$

Verschuiving naar rechts betekent deling door 2, wat bij een 2-complementsgetal neerkomt op:

$(256 - a) / 2 = 128 - a / 2$

Door nu het MSB weer één te maken, tellen we bij dit quotiënt 128 op met als resultaat:

$$128 + 128 - a / 2 = 256 - a / 2$$

Wat de juiste 2-complementsnotatie van het quotiënt is. Voor positieve getallen, waarbij bit 7 nul is, werkt de verschuiving natuurlijk ook correct.

8 Adressering, executietijd, opbouw instructies

De Z-80 heeft een uitgebreide instructieset. Het laden van de accumulator kan met LD A,3. In de instructie komt het getal 3 direct na de instructiecode. Hetzelfde kan met LD A,(3145H), waarbij de te laden data in adres 3145 staat, of met LD A,(HL) in welk geval het adres van de data zich in registerpaar HL bevindt. En er zijn nog meer mogelijkheden. Dit aanwijzen van de plaats waar data staat of waarheen gesprongen moet worden, heet adresseren.
Voor welke adressering men kiest, hangt af van het programma, de beschikbare tijd en de grootte van het geheugen. De uitvoering van een instructie neemt tijd in beslag, voor de ene meer dan voor de andere. Deze executietijd is weer afhankelijk van de opbouw van de instructie. In dit hoofdstuk behandelen we deze nauw verbonden zaken in bovengenoemde volgorde.

8.1 Adressering

Net als bij het versturen van post slaat adresseren hier op het aanwijzen van een locatie. Dat kan een intern register zijn, een poort of een geheugenplaats. Het aanwijzen van de locaties gebeurt in de instructie, zoals bijv. in ADD A,E. De benodigde data staat hier in de registers A en E. Het bekijken van de adresseermogelijkheden verschaft de programmeur inzicht in zowel de mogelijkheden als de beperkingen van de instructieset. Zo kan bijvoorbeeld ADD A,(HL) wel maar ADD A,(BC) niet.
Voor sommige instructies moet de data altijd op een bepaalde plaats staan. Aangezien hier geen variatie mogelijk is, wordt de locatie niet in de instructie vermeld. Dat is onder andere het geval bij CPL die alleen de inhoud van de accumulator kan complementeren en LDIR.
Adressering heeft alleen betrekking op dataverplaatsende of dataveranderende instructies. Andere instructies zijn bijv.: NOP (doe één machinecyclus lang niets, veel gebruikt in vertragende lussen), HALT, EI (enable interrupt) enz. Veelal hebben deze instructies betrekking op hardware-aangelegenheden; ze vallen dan ook buiten het bestek van dit boek.

Alle vormen van adressering hebben een vrij vanzelfsprekende naam. Bij LD A,B

is er sprake van registeradressering omdat de aangewezen locaties registers zijn. Een instructie als LD A,(IX + 6) heeft in feite twee adresseringsvormen. De bestemming van de data, de accumulator, is register-geadresseerd. De bron is aangegeven met geïndexeerde adressering. De Z-80 telt zes op bij de inhoud van het zestienbits-register IX en beschouwt de som als het adres vanwaar de data moet worden gehaald. De volledige vorm van de instructie is LD *r*,(IX + *d*) waarbij *r* zoals gebruikelijk A, B, C, D, E, H en L kan zijn. *d* stelt een éénbyte-getal in twee-complementsnotatie voor. Een dergelijke laadinstructie is alleen mogelijk met de registers IX en IY die daarom indexregisters heten.
De verplaatsing *d* moet men in de instructie als getal opgeven. Deze kan dus niet tijdens de uitvoer van het programma worden berekend.
De aanwezige adresseermogelijkheden zouden een maatstaf kunnen zijn bij het vergelijken van verschillende processors. Jammer genoeg hanteren fabrikanten verschillende namen voor adresseringsvormen. Erger is het als overeenkomstige adresseringsvormen in de praktijk toch een heel andere aanpak blijken te vereisen. Zo kent de 6502-processor de instructie LDA 3145,X waarbij LDA staat voor LoaD Accumulator en X een achtbits-register is zoals bijv. C. De 6502 telt de inhoud van register X op bij adres 3145 en laadt de accumulator vanuit het nieuwe adres. Bij de 6502 is het dus wel mogelijk de verplaatsing tijdens het programma te berekenen. In beide gevallen is er sprake van geïndexeerde adressering die echter bij het programmeren een totaal andere aanpak vereist.
Een goed voorbeeld hiervan is het werken met tabellen. De meeste BASIC-interpreters zijn 'tokenised'. Dat wil zeggen dat de editor (het opmaakprogramma) na het intypen van een BASIC-regel, BASIC-commando's en functies omzet in een éénbyte-getal. In een enkel teken (Engels: token) dus. Tijdens de programma-uitvoer hoeft de interpreter deze woorden niet meer letter voor letter na te gaan, wat de snelheid aanzienlijk vergroot.
Een van de eerste echte huiscomputertjes, de ZX-81, omzeilde ook het omzetten van BASIC-woorden in een teken door een zogenaamde single key entry, waarbij een BASIC-statement werd ingebracht door een enkele toetsindruk. Dit vereenvoudigde de editor aanzienlijk.
In plaats van een serie karakters die woorden vormen als LET, IF of SIN, komt de interpreter een éénbyte-getal tegen. Aan de hand daarvan moet de interpreter de juiste routine aanroepen. Het BASIC-equivalent van het daarvoor benodigde machinetaalprogramma zou eruit kunnen zien als:

```
1000 IF TOKEN=20 THEN GOTO 6000
1010 IF TOKEN=21 THEN GOTO 6180
                    :
                    : enz.
```

Is het gevonden token 100 dan moet de interpreter 100 van dergelijke vergelijkingen afwerken. Veel simpeler is het de startadressen van alle routines in een tabel te zetten.

```
tabel:   01        startadres token 00 = 2301H
         23              (routine voor LET)
         24        startadres token 02 = 3124H
         31              (routine voor IF)
         15        startadres token 04 = 1615H
         16              (routine voor SIN)
         :                : enz.
```

Bij het startadres van de tabel hoeft nu alleen het token te worden opgeteld om het adres te vinden waarin het startadres van de routine staat. Aangezien een adres twee bytes in beslag neemt, gebruiken we alleen even tokens.
De 6502 laat, zoals gezien, toe dat bij geïndexeerde adressering de verplaatsing in een register staat. We zetten dan bijv. het token in register X en brengen het startadres in de tabel over naar een bekend adres.

```
LDA TABEL,X   ;laad A met inhoud tabel + index
STA 1000      ;zet inhoud A in adres 1000
LDA TABEL+1,X ;idem voor hogere deel startadres
STA 1001
JMP (1000)
```

STA staat voor STore A, ofwel berg A op. De laatste instructie JMP (JuMP) springt naar het adres dat in geheugenlocaties 1000 en 1001 staat.
Bij de Z-80 is het niet mogelijk de inhoud van een register als index te gebruiken. We moeten hier een andere weg volgen en zetten de verplaatsing, het token dus, in register E.

```
LD D,0       ;verplaatsing in DE
LD HL,TABEL ;startadres tabel
ADD HL,DE    ;HL wijst naar adres routine
LD E,(HL)    ;lage deel adres in E
INC HL
LD D,(HL)    ;hoge deel adres in D
EX DE,HL     ;verwissel inhoud HL en DE
JP (HL)      ;spring naar de routine
```

De laatste instructie springt naar het adres dat in registerpaar HL staat. Beide processors kennen dus geïndexeerde adressering, beide kennen indirect geïndexeerde adressering, maar toch zijn de oplossingen van hetzelfde probleem nogal

verschillend. Hierbij moet voor de duidelijkheid nog even worden aangetekend dat de 6502 behalve Stack Pointer en Program Counter maar drie achtbits-registers heeft, die bovendien niet te combineren zijn tot bijvoorbeeld een zestienbits-paar.
We geven nu een kort overzicht van de adresseermogelijkheden van de Z-80.

Registeradressering
De data staat in één van de Z-80 registers of registerparen. Voorbeelden:

```
INC HL
LD C,B
```

Onmiddellijke adressering
Achtbits-data staat in het geheugen direct achter de opcode van de instructie. Voorbeelden:

```
LD E,21
ADD A,125
```

In beide voorbeelden is de bron onmiddellijk geadresseerd maar de bestemming, de plaats dus waar de data heen moet, is registergeadresseerd.
Alle achtbits-registers zijn onmiddellijk te laden, met uitzondering van I en R: het Interruptvector-register en het Refresh-register. Deze zijn alleen met data te laden via de accumulator. Begrip omtrent de functie van deze registers vereist een flinke dosis kennis omtrent hardware, reden waarom ze in dit boek niet worden besproken.

Onmiddellijke uitgebreide adressering.
Hetzelfde als de onmiddellijke adressering met dit verschil dat hier zestienbits-data direct achter de opcode van de instructie staat. Voorbeelden:

```
LD DE,6372
LD IY,2895
```

Van deze adresseringswijze is alleen sprake bij het laden van zestienbits-registers. De bestemming van de data is registergeadresseerd. Alle zestienbits-registers zijn op deze manier te laden, behalve de PC.

Uitgebreide adressering
Direct na de opcode volgt een zestienbits-adres vanwaar of waarheen geladen moet worden of waarheen een jump of call moet plaatsvinden. Voorbeelden:

```
JP 1564
LD (5847),A
LD HL,(3964)
```

Van de achtbits-registers kan de Z-80 alleen de accumulator op deze manier laden of schrijven. LD B,(5847) is dus niet toegestaan. Met uitzondering van de PC is het wél voor alle zestienbits-registers mogelijk. Bij LD HL,(3964) laadt de Z-80 eerst L met de inhoud van adres 3964 en daarna H met die van adres 3965.

Register-indirecte adressering
In één van de registerparen staat het adres van de data. Voorbeelden:

```
LD B,(HL)
LD (DE),A
LD (HL),56
RR (HL)
JP (HL)
```

In het eerste voorbeeld is de bestemming, in het tweede de bron register-geadresseerd. We stuiten hier op de eigenaardige asymmetrie van de instructieset. De accumulator kan registerindirect worden geladen van en naar het adres in BC, DE en HL. De andere achtbits-registers alleen via HL. Een register-indirect geadresseerde sprong als JP (HL) bestaat alleen onvoorwaardelijk en met het adres in register HL, IX of IY. De Z-80 kan de CALL-instructie niet op deze manier gebruiken.
In het derde voorbeeld is de bron onmiddellijk geadresseerd. De data, in dit geval 56, staat in de geheugenplaats volgend op de instructie. De bestemming, de door HL aangewezen geheugenlocatie, is register-indirect geadresseerd. Dit laden van een geheugenlocatie met onmiddellijk geadresseerde data kan alleen via (HL) of de indexregisters, bijv. LD (IX+15),56.

Geïndexeerde adressering
Deze adressering lijkt veel op de register indirecte adressering. In één van de indexregisters IX en IY staat een adres. De Z-80 telt hierbij een in de instructie op te geven achtbits-getal in twee-complementsnotatie op en beschouwt de som als een adres. Voorbeelden:

```
SRL (IY+27)
LD (IX+89),B
ADD A,(IY-56)
```

De Z-80 kan alle achtbits-registers geïndexeerd geadresseerd laden en schrijven.

Relatieve adressering

Er zijn adresseringsvormen waarbij, om geheugen en tijd te besparen, een adres wordt opgegeven met een achtbits-getal. Relatieve adressering is zo'n adresseringsvorm. Om toch aan een zestienbits-adres te komen, telt de Z-80 het achtbits-getal, in twee-complementsnotatie, op bij de inhoud van de PC. Deze adressering is alleen mogelijk bij relatieve sprongen. Het sprongbereik t.o.v. de PC is 127 adressen voorwaarts en 128 terug. Voorbeelden:

```
JR 51
JR NC,64
```

Bij het werken met een assembler geeft men in plaats van de sprong in getalvorm meestal een label op. De assembler rekent dan zelf wel de juiste grootte uit. Wil men toch de verplaatsing als getal opgeven, denk dan aan het volgende. De Z-80 telt de verplaatsing op bij de PC na het ophalen van de volledige instructie. De PC wijst op dat moment al naar de opcode van de volgende instructie en vanaf die locatie moet de verplaatsing worden uitgerekend.

Gemodificeerde pagina-nul-adressering

Nog een vorm waarbij een adres maar uit acht bits bestaat. Het volledige zestienbits-adres komt tot stand door de eerste acht bits nul te veronderstellen. Vandaar pagina nul. Men verdeelt namelijk het geheugen wel in blokken, pagina's, van 256 bytes. Het blok adressen waarvan het eerste byte nul is heet dan pagina nul, dat waarvan het eerste byte één is pagina één enz.
De Z-80 kent maar een zeer beperkte pagina-nul-adressering, namelijk alleen voor de restart-instructies. Vandaar de naam gemodificeerde pagina nul adressering. Voorbeelden:

```
RST 10H
RST 28H
```

ReSTarts zijn alleen mogelijk naar de adressen 0, 8H, 10H, 18H, 20H, 28H, 30H en 38H. RST 28H voert een call uit naar adres 0028H. Op adres 0 begint uiteraard het programma voor de koude start die hetzelfde effect heeft als het aanzetten van de computer. Meestal beginnen op de andere adressen voor het systeem belangrijke subroutines. Bijvoorbeeld een subroutine die een ASCII-karakter in A op het scherm zet.

Impliciete adressering

In de instructie worden bron, bestemming of beide niet aangegeven. Voorbeelden:

```
SUB C
EXX
LDIR
```

In het eerste voorbeeld is één van de termen voor de aftrekking, het C-register, register-geadresseerd. De andere term, alsmede de bestemming voor het resultaat, is de accumulator. Deze wordt in de instructie niet genoemd. EXX verwisselt de inhoud van de registerparen BC, DE en HL met die van de alternatieve registerset. Zowel de bron als de bestemming komen in de instructie niet voor. Deze worden verondersteld impliciet, stilzwijgend, te zijn aangegeven. In het laatste voorbeeld is de bron de inhoud van (HL) en de bestemming de inhoud van (DE).

Bitadressering
De Z-80 kent een drietal instructies die het mogelijk maken een enkel bit in een byte te testen, te setten of te resetten. Dit aanwijzen van een afzonderlijk bit heet bitadressering. Voorbeelden:

```
BIT 3,D
SET 4,(HL)
RES 6,(IX+2)
```

Het resultaat van de bittest, in het voorbeeld BIT drie,D, set of reset de zero-vlag als bit drie van register D nul respectievelijk één is. De bytes zelf zijn in de voorbeelden achtereenvolgens register, register indirect en geïndexeerd geadresseerd.

8.2 Executietijd

Het uitvoeren van een instructie kost de Z-80 uiteraard tijd. Bij de ene instructie is dat meer, soms veel meer, dan bij de ander. Voor het ontwerpen van een programma is het nodig enig inzicht te hebben in deze executietijden. Moet een programma snel zijn dan dient men tijdverslindende instructies in lussen te vermijden. Om lussen die het programma vele malen doorloopt sneller te laten uitvoeren, kan het zelfs winstgevend zijn buiten de lus wat meer "lange" instructies op te nemen die het werk binnen de lus verlichten. We beginnen met de introductie van enkele begrippen. De uitvoering van een instructie heet de instructiecyclus. Deze is verdeeld in machinecycli. Een instructiecyclus bestaat bij de Z-80 uit één tot zes machinecycli.
De ene machinecyclus duurt weliswaar iets langer dan de andere maar de tijd die nodig is voor de uitvoering ervan ligt in dezelfde orde van grootte. We kunnen dan ook stellen dat de executietijd van een instructie langer is naarmate er meer machinecycli zijn.
Nu is het zo dat de hoeveelheid machinecycli binnen een instructie, op een enkele uitzondering na, vastligt door het aantal malen dat de Z-80 voor het uitvoeren van de instructie het geheugen aanspreekt.

De instructie LD D,B neemt in het geheugen één byte in beslag en de instructiecyclus bestaat uit één machinecyclus. LD A,(HL) neemt ook maar één byte in beslag. Nadat de opcode echter uit het geheugen is gehaald, zet de Z-80 de inhoud van HL op de adresbus en plaatst het uit dat adres opgehaalde byte in A. Het geheugen wordt hier tweemaal aangesproken: de instructiecyclus bevat twee machinecycli.

LD A,(3010H) staat als volgt in het geheugen:

3A	laadt A uitgebreid geadresseerd
10	lagere deel adres
30	hogere deel adres

Het ophalen van deze drie bytes gebeurt tijdens drie machinecycli. Vervolgens gaat adres 3010H op de adresbus en wordt A met de inhoud hiervan geladen. Nodig zijn vier machinecycli.
Toch is het hiermee oppassen geblazen. Een simpele instructie als INC (HL) is maar één byte groot. De wisselwerking tussen Z-80 en geheugen is echter als volgt.

Zet de PC op de adresbus en haal de instructiecode op.

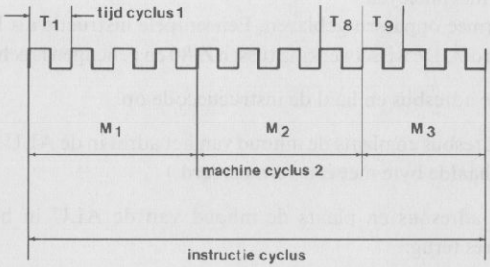
Zet HL op de adresbus en plaats de inhoud van het adres in de ALU. (In de ALU wordt het opgehaalde byte met één vermeerderd.)

Zet HL op de adresbus en plaats de inhoud van de ALU in het door HL aangewezen adres terug.

De Z-80 spreekt het geheugen driemaal aan. De instructiecyclus bestaat dan ook uit drie machinecycli.
In enkele gevallen is er één machinecyclus meer nodig dan het aantal wisselwerkingen met het geheugen rechtvaardigt. Dit is het geval bij het geïndexeerd laden van en naar registers. LD B,(IX+5) is een instructie die drie bytes vergt. Een vierde machinecyclus is nodig om de inhoud van het opgegeven adres te halen. Toch telt de totale instructiecyclus vijf machinecycli.
Een andere uitzondering is de relatieve sprong. Zoals we zagen, geeft deze niet een zestienbits-adres op maar een achtbits-verplaatsing ten opzichte van de PC. Dat spaart in de code voor de instructie één byte uit.

JP		JR
C3	code	18 code
10	adres laag	24 verplaatsing
30	adres hoog	

Toch heeft de relatieve sprong net als de absolute sprong drie machinecycli nodig. Ze zijn zelfs iets langer zodat JR net wat meer tijd neemt dan JP. Anders is het wanneer we de voorwaardelijke sprongen met elkaar vergelijken, dus bijv. JP NC,ADRES en JR NC,VERPLAATSING. Is aan de voorwaarde voldaan en moet er gesprongen worden dan gaat het bovenstaande verhaal ook op. Vindt de sprong niet plaats dan heeft JP NC,ADRES uiteraard nog steeds drie machinecycli nodig. JR NC,VERPLAATSING echter maar twee. Het niet hoeven uitrekenen van het adres aan de hand van PC en verplaatsing maakt blijkbaar de derde machinecyclus overbodig.
Een machinecyclus bestaat uit drie tot vijf tijdcycli. Een tijdcyclus is een bepaalde hoeveelheid tijd. Op het gevaar af te technisch te worden: alles wat in de computer gebeurt, staat onder invloed van een elektronische klok. Deze geeft elektrische pulsen af, de tijdcycli, zoals te zien is in afb. 8.1.



Afb. 8.1 Tijd-, machine- en instructiecyclus

Meestal is in de documentatie de klokfrequentie van de computer, d.w.z. het aantal pulsen per seconde, opgegeven. Is deze bijvoorbeeld 2,5 MHz (megahertz) dan is het aantal pulsen per seconde $2{,}5 \times 10^6$ ofwel 2.500.000. Een tijdcyclus duurt dan $1/2{,}5 \times 10^6 = 4 \times 10^{-7} = 400 \times 10^{-9} = 400$ nanoseconden.
In de getekende situatie is er sprake van een instructiecyclus die bestaat uit drie machinecycli, M1, M2 en M3, van respectievelijk vier, vier en drie tijdcycli. De instructiecyclus duurt 11 T (tijd) cycli.
Van elke instructiecyclus en machinecyclus is het aantal tijdcycli bekend. LD A,(3010H) neemt dertien tijdcycli in beslag. Uitvoering ervan duurt dus $13 \times 400 \times 10^{-9} = 5{,}2$ microseconden. De afzonderlijke machinecycli nemen respectievelijk vier, drie, drie en nogeens drie tijdcycli in beslag. De instructieset in Appendix A geeft voor elke instructie het aantal tijd- of T-cycli.

8.3 Opbouw instructies

We hebben gezien dat de instructie LD A,(3010H) als volgt in opeenvolgende plaatsen van het geheugen staat:

3A	opcode
10	lagere deel adres
30	hogere deel adres

In dit geval bepaalt het eerste byte de uit te voeren operatie, namelijk het uitgebreid geadresseerd laden van A. De twee volgende bytes bevatten data die in dit geval een adres voorstelt.
Het byte dat de operatie bepaalt, heet de operatiecode, kortweg opcode. Voert de Z-80 de instructie uit dan wordt tijdens de eerste machinecyclus de opcode uit het geheugen gehaald. Deze eerste cyclus wordt ook wel opcode-fetch genoemd.
Er zijn instructies zonder data die bestaan uit één of twee bytes opcode:

12	LD (DE),A
ED	LDIR
B0	

In dit laatste geval vindt er tweemaal een opcode-fetch plaats. Natuurlijk kunnen na één of twee bytes opcode weer één of twee bytes data volgen.

C6	opcode ADD A,3
03	data
DD	opcode LD B,(IX+3)
46	opcode
03	verplaatsing
ED	opcode LD BC,(5460H)
4B	opcode
60	lagere deel adres
54	hogere deel adres

Ook kunnen twee bytes opcode worden gevolgd door één byte data en vervolgens weer een byte opcode.

FD	opcode RR (IX+5)
CB	opcode
05	verplaatsing
1E	opcode

Instructies van de Z-80 zijn maximaal vier bytes groot. Daarvan zijn één tot drie bytes opcode. Hoe langer een instructie, des te meer tijd kost het deze uit het geheugen te halen. Eén byte opcode is in elk geval noodzakelijk. Soms zijn het er meer; dit komt door het grote aantal instructies van de Z-80. Gebruikt men slechts éénbyte-opcodes, zoals bij de 6502, dan kan men niet meer dan 256 instructies kwijt. De Z-80 heeft er veel meer en dus moet een aantal instructies beginnen met twee bytes opcode.
De Z-80 is de opvolger van de 8080-microprocessor. Bij het ontwerpen van de Z-80 ging men ervan uit dat een programma dat voor de 8080 was geschreven voor de 8080 ook op de Z-80 moest werken. Met andere woorden: alle opcodes voor de 8080 moesten op de Z-80 hetzelfde effect hebben. Dit niet zonder reden: voor de 8080 bestond een uitgebreide hoeveelheid programmatuur. Bezitters van een 8080-systeem konden overschakelen naar een Z-80 systeem zonder hun software te hoeven weggooien.
Kijken we naar de opbouw van de 8080 dan vinden we daar de bekende registers A, B en C, D en E, H en L. De indexregisters, evenals de alternatieve set, ontbreken.
De 8080 kent alleen éénbyte-opcodes. Aangezien de instructieset toch nog vrij uitgebreid was, waren er nog maar twaalf opcodes vrij. Een aantal daarvan werd besteed aan nieuwe, snelle instructies. Bijvoorbeeld:

08	EX AF,AF'
10	DJNZ verplaatsing
D9	EXX

De opcodes 18, 20, 28, 30 en 38 hexadecimaal gingen naar de onvoorwaardelijke en voorwaardelijke relatieve sprongen (JR). De vier overblijvende vormen het eerste byte van de instructie met meerbytes opcodes. Alles wat het nieuw toegevoegde register IX betreft, begint met opcode DD. Het tweede opcode-byte is gelijk aan dat voor dezelfde instructie met HL.

INC HL	INC IX
23	DD
	23

Een increment van IX (of IY) neemt dus tweemaal zoveel tijd als dat van HL. Voor het IY-register geldt hetzelfde, al begint de opcode daar met FD.

LD A,(HL)	LD A,(IY + 3)
7E	FD
	7E
	03

De overblijvende codes CB en ED zijn gebruikt als eerste opcodebyte voor diverse instructies. Juist hier zien we enkele merkwaardigheden. De 8080 kent een instructie om HL uitgebreid geadresseerd te laden, opcode 2A. De Z-80 kent deze eveneens:

```
LD HL,(3674H)  2A
               74
               36
```

Toegevoegd op de Z-80 werden instructies om hetzelfde te doen met BC, DE en SP. Al deze laadinstructies beginnen met EDH. Blijkbaar was het om technische redenen nodig in deze rij ook HL weer op te nemen. Er zijn dus twee opcodes om HL op genoemde wijze te laden.

```
LD BC,(3674H)  LD HL,(3674H)
     ED             ED
     4B             6B
     74             74
     36             36
```

Om dezelfde reden komen er wel meer verdubbelingen voor:

```
RRA            RR A
1F             CB
               1F
```

De oorzaak van de verdubbeling heeft te maken met de manier waarop instructies in elkaar zitten. We laten dat, tot slot van dit hoofdstuk, aan de hand van enkele eenvoudige voorbeelden nog even zien. Als eerste illustratie daarbij dient het laden van het ene register met de inhoud van het andere.

```
LD C,B     01 001 000
LD C,E     01 001 011
LD E,A     01 011 111
LD E,B     01 011 000
```

Om te laten zien hoe de instructie in elkaar zit, is de binaire vorm van de opcode op ongebruikelijke manier in groepjes verdeeld. De eerste twee bits zijn voor alle instructies hetzelfde. Deze definiëren hier de instructie, namelijk het laden van een register. De twee groepjes daarna bepalen achtereenvolgens bron en bestemming. Register C heeft als driebits-code 001, register E 011 enz. Met drie bits kunnen acht registers worden aangewezen. A, B, C, D, E, H en L zijn er maar zeven. De achtste mogelijkheid is gereserveerd voor (HL).

LD B,(HL)	01 000 110
LD C,(HL)	01 001 110

Instructies waarbij registerparen betrokken zijn, reserveren vaak twee bits om een registerpaar te selecteren. We vergelijken acht- en zestienbits-increment.

INC A	00 111 100
INC B	00 000 100
INC C	00 001 100

Het middelste veld wijst het register aan en gebruikt daarvoor dezelfde code als bij de laadinstructies. De overblijvende achtste mogelijkheid wordt ook hier gebruikt voor (HL).

INC BC	00 00 0011
INC DE	00 01 0011
INC HL	00 10 0011
INC SP	00 11 0011

Hier wijst het middelste veld een registerpaar aan. Het moge duidelijk zijn dat deze aanwijzende velden in een instructie worden gebruikt door het data-richtingsregister van de processor.
Dit verklaart ook de verdubbelingen in de instructieset. Er was bijv. al een instructie LD HL,(adres) met een éénbyte-opcode. De toegevoegde instructie met tweebytes-opcode om ook andere registerparen uitgebreid geadresseerd te laden, heeft de volgende vorm:

11 10 1101	
01 *ss* 1011	*ss* = BC 00
adres laag	*ss* = DE 01
adres hoog	*ss* = HL 10
	ss = SP 11

Hierin is *ss* het veld waarin de code voor het registerpaar komt. Aangezien deze codes voor het datarichtingsregister dezelfde zijn als bij de increment-instructies moet ook hier de mogelijkheid aanwezig zijn om HL als bestemming aan te wijzen. Vandaar dat, hoewel LD HL,(adres) al aanwezig was, en er vanwege de compatibiliteit met de 8080 ook moet blijven, er een tweede bestaat met een tweebytes-opcode. Hetzelfde verhaal geldt voor de andere, wat betreft opcode dubbel aanwezige instructies.

9 Input/Output

9.1 Algemeen

De term input/output, meestal afgekort tot I/O, slaat op de in- en uitgaande informatiekanalen van de computer. Na het aanzetten van de computer is het geselecteerde invoerkanaal praktisch altijd het toetsenbord. Daarvandaan komen de te verwerken opdrachten en data. Het geselecteerde uitvoerkanaal is het beeldscherm. Dit geeft, ter controle, het invoerkanaal weer (echo) alsmede uitkomsten van berekeningen, foutmeldingen e.d.
Als de computer daartoe de mogelijkheid biedt, kunnen andere kanalen worden gekozen, bijv. een RS232-interface als uitvoer in plaats van het beeldscherm. Of de Centronics-interface voor de printer.
Op dezelfde manier laat zich trouwens een bestandssysteem kiezen. Bestanden en programma's kunnen worden gelezen en geschreven van en naar tape of disk.
Het kiezen van een ander in- of uitvoerkanaal lijkt, heel grof geschetst, op het omzetten van een schakelaar in de hardware.
Op de printplaat of -platen in de computer zitten de volgende soorten chips:

microprocessor;

geheugen-chips;

timer(s);

randapparatuur-chips: chip voor parallelle invoer via het toetsenbord;

chip voor parallelle uitvoer met handshake voor de Centronics-interface;

chip voor seriële uitvoer (RS232);

video-chip voor het beeldscherm;

disk-controller-chip.

Buiten dit alles is er nog een flinke hoeveelheid schakelelektronica die ervoor zorgt dat de juiste chips worden aangesproken.
Het aardige aan al deze randapparatuur-chips is dat het voor de microprocessor niet uitmaakt of bijv. een tekst naar het beeldscherm, de printer of de RS232-poort wordt gestuurd. Het enige dat bij een uitvoeropdracht gebeurt, is dat de processor kijkt welk uitvoerkanaal is geselecteerd en daar de ASCII-tekens naartoe stuurt. De randapparatuur-chips zorgen dan voor de verdere verwerking.

9.2 Het besturingssysteem

Meestal denken we als gebruiker niet aan de computer in termen als chips, printplaten en andere elektronische onderdelen. We zetten het apparaat gewoon aan en laden een tekstverwerker of een bestandsprogramma, simpelweg door het intypen van een commando plus een naam. Wat we op dat moment van de computer zien, is de gebruikerskant van het besturingssysteem. Het besturingssysteem (in het Engels Operating System) biedt onder meer de mogelijkheid tot het laden, opslaan en creëren van files, het uitvoeren van programma's, het afdrukken van files enz.
Bezitters van de meeste huiscomputers hebben in eerste instantie weinig 'direct contact' met het besturingssysteem. Bij veel tot dusver uitgebrachte microcomputers is namelijk een BASIC-interpreter ingebouwd die bij het aanzetten meteen wordt gestart. Gebruikers van zulke machines zien in feite alleen BASIC. Onder die omstandigheden zijn veel commando's van het besturingssysteem beschikbaar in de vorm van BASIC-equivalenten: het laden en opslaan van files en programma's. De interpreter maakt bij het uitvoeren daarvan (en ook in veel andere gevallen) gebruik van het besturingssysteem.
Een bekend besturingssysteem voor microcomputers is CP/M. Er zijn de laatste tijd flink wat besturingssystemen bijgekomen; we noemen onder andere MS-DOS (en PC-DOS), GEM en Unix. Bij het werken met bijv. CP/M of GEM heeft de gebruiker na de start alleen de beschikking over het besturingssysteem. Wil deze gebruiker vervolgens met BASIC aan de slag dan dient hij of zij eerst het daarvoor benodigde programma van schijf of welk medium dan ook te laden.
In een aantal gevallen moet de computer na aanzetten ook het besturingssysteem nog van schijf halen; de computer heeft dan alleen een startprogramma ter beschikking. Zo'n startprogramma stelt dan onder andere alle default-waarden in voor de aangesloten randapparatuur. Het programmeert bijv. de video-chip voor een bepaald aantal tekens per regel, maakt het videogeheugen schoon en stelt de

cursorpositie in; het zet de RS232 op een bepaalde overdrachtssnelheid, kiest als invoerkanaal het toetsenbord enz.
Zonder oneerbiedig te willen zijn, het besturingssysteem laat zich het best vergelijken met een ui. De binnenste schil ligt direct op de hardware, de buitenste schil is hetgeen de gebruiker ziet. Van de buitenste schil, waar de gebruiker soms gecompliceerde commando's kan geven, naar binnen gaand, wordt het niveau primitiever. In de binnenste schil bevindt zich bijv. het interruptsysteem. Allerlei randapparatuur-chips kunnen de Z-80 onderbreken en een karweitje laten verrichten dat niet kan wachten. Welk programma er ook draait, als de RS232-poort als invoerkanaal is gekozen en deze poort een karakter heeft ontvangen, onderbreekt de RS232-chip de Z-80. Deze zet nu eerst het ontvangen karakter in een buffer, verhoogt de teller voor het aantal buffertekens, geeft de RS232 vrij baan voor ontvangst van een nieuw teken en vervolgt dan z'n oorspronkelijk programma. Komt daarin een vraag naar invoer voor zoals INPUT A$ dan wordt de genoemde buffer uitgelezen. Een soortgelijk proces vindt plaats met het toetsenbord.
Om een onderbroken programma weer te kunnen vervolgen, is het niet voldoende de oorspronkelijke PC terug te halen. Ook registers en vlaggen moeten in de toestand van de onderbreking worden gebracht. Als de routine die een karakter van de RS232-chip in een buffer zet register A, BC en de vlaggen gebruikt, moet deze routine beginnen met AF en BC op de stapel te zetten. Net voor de terugkeer naar het onderbroken programma krijgen de registers weer hun oorspronkelijke waarden.

We zullen nu met een voorbeeld een indruk proberen te geven aangaande het primitiever worden van niveau's naarmate men in het besturingssysteem dichter bij de hardware komt. Stel, er is getracht een file te laden dat niet op schijf stond. Het besturingssysteem moet nu met een melding komen in de trant van 'File not found'. We volgen de uitvoering hiervan door de diverse lagen van het systeem. Het startadres van de af te drukken string, "File not found", is bekend. Met dat startadres in bijv. een registerpaar wordt een subroutine PRINT_STRING aangeroepen.

PRINT_STRING: Deze subroutine is een algemene afdrukroutine, gebruikt voor alle systeemmeldingen. De subroutine verwacht het startadres van de string in een registerpaar en stuurt alle tekst tot en met een newline naar een primitiever niveau van de uitvoer. Dat gebeurt teken voor teken. Met een enkel teken van de string in bijv. A wordt de subroutine PRINT_TEKEN aangeroepen.

PRINT–TEKEN: De aangeroepen subroutine wordt niet alleen door het besturingssysteem gebruikt maar ook door toepassingsprogramma's als tekstverwerkers, BASIC-interpreters en dergelijke. Het karakter in A (dat kan een leesbaar teken of een besturingsteken voor beeldscherm of printer zijn) gaat naar het gekozen uitvoerkanaal.
Is dit het beeldscherm dan komen we ten slotte bij een routine terecht die een leesbaar teken, omgezet in een puntjes-patroon, in het videogeheugen zet en de afdrukpositie op het scherm bijhoudt. Uiteindelijk zet de videoprocessor, buiten de programma's om, het dot-patroon om in een videosignaal.
Eventuele besturingstekens als bijv. ASCII 12 voor het schoonmaken van het scherm worden door een andere routine verwerkt.

9.3 Basis I/O routines

De hier behandelde I/O routines gebruiken beeldscherm en toetsenbord als respectievelijk invoer- en uitvoerkanaal. Het niveau van de basisroutines is nogal primitief: ze lezen een enkele toetsindruk of schrijven een enkel karakter weg. Elk besturingssysteem beschikt over deze mogelijkheden. Werken de basisprogrammaatjes eenmaal dan kunnen ze vanuit ingewikkelder programma's worden aangeroepen. Deze laatste zijn dan niet meer afhankelijk van één merk of type computer.

9.3.1 Basisinvoer

Zoals gezegd, beschikt elk besturingssysteem over de mogelijkheid een toetsindruk te lezen. Waar de routine in het geheugen begint en hoe deze is aan te roepen, verschilt nogal per computer. Dit uitvissen kan dus wel enig speurwerk met zich mee brengen. Het door de schrijvers gebruikte CP/M-achtig systeem doet het als volgt:

laad register C met 1
voer een CALL uit naar adres 0005

De subroutine leest een enkel teken uit de toetsenbordbuffer. Is die buffer leeg dan wacht de routine tot een toets is ingedrukt. Ervaren BASIC-programmeurs herkennen hierin een machinetaalequivalent van de BASIC-functie INKEY, waar de interpreter dan ook gebruik van maakt.
De routine geeft bij terugkeer de ASCII-code voor de ingedrukte toets in register

A. De aangeroepen routine op adres 0005 verandert de inhoud van register L en natuurlijk die van A. Aangezien we de basisinvoer later willen aanroepen vanuit ingewikkelder programma's is het zaak de inhoud van de diverse registers zo min mogelijk te veranderen. Deze zouden anders in latere programma's niet meer als opslagregister kunnen worden gebruikt.
Register A moet uiteraard van inhoud veranderen.
De basisinvoer doet dan ook het volgende:

zet BC op de stapel
zet HL op de stapel
lees een karakter van het toetsenbord
herstel inhoud HL
herstel inhoud BC

Bij uitvoer van een enkel karakter wordt ook adres 0005 aangeroepen, met echter in C niet de waarde één maar twee. Adres 0005 is hier dan ook een algemeen ingangspunt voor allerlei aanroepen naar het BDOS ofwel BASIC Disk Operating System. Welke functie men wil uitvoeren, hangt af van de inhoud van register C.

Programma H9P1 Basisinvoer en -uitvoer

```
;       Basis invoer en uitvoer
;       Leest karakters van het toetsenbord en
;       drukt ze af op het scherm
;
;       Gebruikte registers: AF, C, E
;
;


;Zet een vraagteken op het scherm
        LD      A,'?'
        CALL    SCHRKAR         ;druk teken in A af
;Lees en schrijf tot karakter een Q is
IOLUS:  CALL    LEESKAR         ;zet toetsindruk in A
        CALL    SCHRKAR
        CP      'Q'             ;is het een Q ?
        JP      NZ,IOLUS        ;nee, dan volgende teken
        RET

;Basis invoer en uitvoer routines
;Constanten:
BDOS    EQU     5       ;CP/M entry
LEES    EQU     1
SCHRYF  EQU     2
```

```
;Invoer, geeft bij terugkeer karakter in A
LEESKAR:
        PUSH    HL              ;bewaar alle betrokken
        PUSH    BC              ;  registers
        LD      C,LEES          ;roep operating
        CALL    BDOS            ;  systeem aan
        POP     BC              ;herstel registers
        POP     HL
        RET

;Uitvoer, zet karakter in A op het scherm
SCHRKAR:
        PUSH    HL              ;bewaar betrokken
        PUSH    DE              ;  registers
        PUSH    BC
        LD      C,SCHRYF        ;roep operating
        LD      E,A             ;  systeem aan
        CALL    BDOS
        POP     BC              ;herstel registers
        POP     DE
        POP     HL
        RET

        END
```

9.3.2 Basisuitvoer

Aanroep van het besturingssysteem gaat als volgt:

Laad E met de ASCII-code voor het karakter
Laad C met 2
Voer een CALL uit naar BDOS entry op adres 0005

Aangezien registerpaar DE uitermate handig is voor het bewaren van adressen of data willen we onze eigen uitvoerroutine aanroepen met het karakter in A in plaats van E. De besturingssysteemroutine verandert zelf nog de inhoud van register L. Het basisuitvoerprogramma wordt nu:

Zet HL op de stapel
Zet DE op de stapel
Zet BC op de stapel
Laad C met 2
Laad E met de inhoud van A
Roep BDOS aan
Haal BC, DE en HL van de stapel

In programma H9P1 staan de basisinvoer- en uitvoerroutine onder de labels LEESKAR (lees karakter) en SCHRKAR (schrijf karakter). Om met het programma ook nog iets te kunnen doen, is aan het begin een routine toegevoegd die invoer en uitvoer beurtelings aanroept. De routine zet eerst een vraagteken op het scherm en drukt vervolgens alle toetsaanslagen af. Het programma stopt nadat een Q is ingedrukt.
In veel gevallen zal de voor het lezen gebruikte besturingssysteemroutine zelf al een echo van de toets aanslag op het scherm zetten. Dat heeft tot gevolg dat bij het intoetsen van een A op het scherm AA verschijnt.
LEESKAR en SCHRKAR gebruiken reeds aanwezige routines. Deze liggen op een primitiever niveau en maken vaak gebruik van echte invoer- en uitvoerinstructies van de Z-80. Een toetsenbord kan bijv. verbonden zijn met een PIO, een Parallel Input Output chip, speciaal ontwikkeld om samen te werken met de Z-80. Bij ontvangst van een karakter van het toetsenbord genereert de PIO een interrupt. De PIO onderbreekt daarmee het programma waaraan de Z-80 bezig is. De Z-80 zet nu eerst het ontvangen karakter in een buffer door de poort van de PIO waarmee het toetsenbord verbonden is te lezen met een invoerinstructie als: IN A,(*n*). Hierin is *n* het poortnummer van het toetsenbord. Na uitvoering van de instructie staat het karakter in A. De Z-80 zet dit in een buffer en hervat dan het programma dat door de PIO werd onderbroken. Door de snelheid waarmee dit alles gaat, lijkt het alsof programma en toetsindruk tegelijkertijd worden verwerkt.
Een andere invoerinstructie is IN *r*,(C) die register *r* laadt met de in C aangegeven poort. Dit alles heeft echter voornamelijk betrekking op de hardware.
Vermeld zij nog dat de Z-80 blokinvoer- en blokuitvoerinstructies kent. Bijv. IND (INput Decrement), INDR (INput Decrement Repeat), OTIR (OuTput Increment Repeat).

9.4 Het gebruik van buffers

Basisinvoer en -uitvoer vindt byte voor byte plaats. Willen we een string invoeren dan zullen we de afzonderlijke tekens ergens moeten laten. Dit gebeurt met behulp van een invoerbuffer, niets anders dan een gereserveerd stukje geheugen.
Voor het eerste stringprogramma gebruiken we een heel eenvoudige techniek. We slaan het eerste byte van het buffer over en zetten de ingevoerde tekens stuk voor stuk in de daarop volgende bytes. Is de invoer ten einde (heeft de gebruiker dus de Return-toets ingedrukt) dan komt het aantal tekens van de string in het eerste nog ongebruikte byte van het buffer te staan.

Om een beperkte mogelijkheid te bieden voor het corrigeren van fouten voordat de Return-toets is ingedrukt, worden de ontvangen karakters gecontroleerd op de ASCII-code voor delete (7F hexadecimaal).
Om aan te geven dat er invoer wordt gevraagd, zet het programma bij wijze van prompt eerst een vraagteken op het scherm. Diagram 9.1 geeft een overzicht van het invoerprogramma.

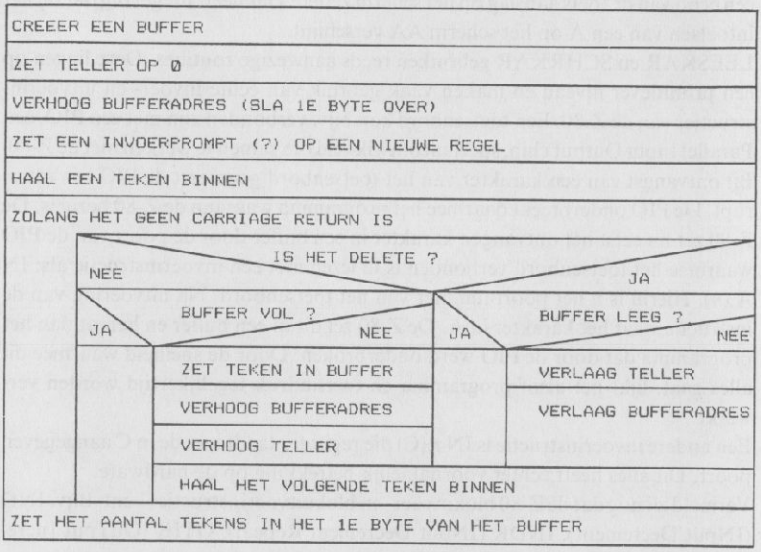


Diagram 9.1 Stringinvoer

De uitvoer is veel eenvoudiger. In het eerste byte van de string staat het aantal tekens. Precies zoveel van de daarop volgende bytes gaan naar het scherm, voorafgegaan door een uitvoer-prompt ">". Diagram 9.2 laat het principe van de uitvoer zien.
De opdrachten "haal teken binnen" en "zet teken op scherm" zijn verwijzingen naar de eerder gemaakte programma's LEESKAR en SCHRKAR. Het programma SCHRKAR dient ook voor het afdrukken van de prompts en het beginnen op een nieuwe regel. In dit laatste geval moet SCHRKAR besturingstekens naar het uitvoerkanaal sturen. Om naar een volgende regel te gaan is dit de ASCII-code 0A hex die een linefeed genereert en om op deze regel vooraan te

```
HAAL STARTADRES BUFFER
ZET AANTAL STRINGTEKENS UIT 1E BYTE BUFFER IN TELLER
VERHOOG BUFFERADRES
ZET UITVOERPROMPT (>) OP EEN NIEUWE REGEL
ZOLANG TELLER NIET Ø IS
    HAAL TEKEN UIT BUFFER
    ZET TEKEN OP SCHERM
    VERHOOG BUFFERADRES
    VERLAAG TELLER
```

Diagram 9.2 Stringuitvoer

beginnen, is dit de ASCII-code 0D hex voor een carriage return. De terminologie is afkomstig uit de wereld van de telex: carriage return betekent wagenterugloop en linefeed is een opdracht om het papier een regel verder te schuiven.
Zoals gezegd, als een machinetaalroutine het besturingssysteem aanroept om het toetsenbord uit te lezen, zal in de meeste gevallen een toetsindruk ook op het scherm verschijnen. Dit heet echo. Mocht de echo ontbreken dan moet op de aanroep van LEESKAR, "haal teken binnen", de routine "zet teken op scherm" met SCHRKAR volgen.
Programma 9.2 geeft een praktische toepassing van de invoer en uitvoer van een string. Wat in de diagrammen staat, is in de listing te vinden als de subroutines INVOER en UITVOER. Een korte hoofdroutine roept deze achtereenvolgens aan. Op hun beurt maken INVOER en UITVOER gebruik van de subroutines LEESKAR en SCHRKAR. Naar deze laatste twee, die immers het laagste niveau vormen, is alle systeemafhankelijkheid gedelegeerd. Als gevolg daarvan zijn de subroutines INVOER en UITVOER zelf volledig onafhankelijk van het systeem.
Om het programma ook nog iets nuttigs te laten doen, is een kleinigheid toegevoegd: omzetting van kleine letters in de invoer naar hoofdletters. Een aantal systemen beschikt over iets dergelijks om gebruikers de vrijheid te geven opdrachten in kleine letters of hoofdletters in te typen terwijl het programma van het besturingssysteem dat de opdracht verwerkt alleen hoofdletters aankan.
Voor de verwezenlijking is gebruik gemaakt van de opbouw van de ASCII-code. Opeenvolgende letters hebben opeenvolgende codes en kleine letters hebben een hogere code, een groter getal, dan hoofdletters. De ASCII-code voor een kleine letter is 32, 20 hex, groter dan die voor de corresponderende hoofdletter. Vindt het

programma in het invoerkanaal een teken met een code van 61 hex t/m 7A hex dan trekt het daarvan 20 hex af alvorens het teken in de string te plaatsen.

Programma H9P2 Invoer en uitvoer van een string

```
;       Invoer en uitvoer van een string van
;       max. 40 tekens met omzetting van
;       onder- naar bovenkastletters
;
;       Gebruikte registers: AF, HL, BC, E
;
;


;Constanten:
BUFLEN  EQU     40      ;lengte van buffer
CR_KEY  EQU     0DH     ;return toets
DEL_KEY EQU     7FH     ;delete toets
LF      EQU     0AH     ;linefeed

START:
        CALL    INVOER  ;hoofdprogramma
        CALL    UITVOER
        RET

INVOER:
;Initialisatie
        LD      HL,BUFFER       ;adres buffer
        LD      B,0             ;teller voor buffer
;Zet vraagteken op een nieuwe regel
        LD      A,CR_KEY        ;karakter in A naar scherm
        CALL    SCHRKAR
        LD      A,LF
        CALL    SCHRKAR
INV_?:  LD      A,'?'
        CALL    SCHRKAR
;Haal string binnen
LSLUS:  CALL    LEESKAR         ;geeft toets in A
        CP      CR_KEY          ;return toets ?
        JP      Z,EINDL         ;ja, naar einde
        CP      DEL_KEY         ;delete toets
        JP      Z,DELETE        ;ja, verwerk
        CP      'a'             ;kleiner dan a ?
        JP      M,NOMZ          ;ja, niet omzetten
        CP      'z'+1           ;groter dan z ?
        JP      P,NOMZ          ;ja, niet omzetten
        SUB     'a'-'A'         ;zet letter om
NOMZ:   LD      E,A             ;bewaar toets in E
        LD      A,B             ;teller in A
        CP      BUFLEN          ;buffer al vol ?
```

```
        JP      Z,EINDL         ;ja, naar einde
        INC     HL              ;volgend adres buffer
        INC     B               ;update teller
        LD      (HL),E          ;teken in buffer
        JP      LSLUS           ;volgende teken
EINDL:
;Zet aantal tekens in 1e byte string
        LD      A,B             ;aantal tekens in A
        LD      (BUFFER),A
        RET
;Verwerking delete toets
DELETE:
        LD      A,B             ;aantal tekens in A
        CP      0               ;buffer nog leeg ?
        JP      Z,INV_?         ;ja, begin opnieuw
        DEC     B               ;verminder teller
        DEC     HL              ;vorig adres buffer
        JP      LSLUS           ;volgende teken

UITVOER:
;Initialisatie
        LD      HL,BUFFER       ;startadres buffer
        LD      B,(HL)          ;aantal tekens in B
;Uitvoerprompt op nieuwe regel
        LD      A,CR_KEY
        CALL    SCHRKAR
        LD      A,LF
        CALL    SCHRKAR
        LD      A,'>'
        CALL    SCHRKAR
;Zet string op het scherm
SLUS:   INC     HL              ;volgende adres buffer
        LD      A,(HL)          ;teken in A
        CALL    SCHRKAR         ;naar scherm
        DJNZ    SLUS            ;alle tekens gedaan ?
        RET                     ;ja, klaar
;
;Subroutines leeskarakter en schrijfkarakter
;
;Constanten:
BDOS    EQU     5               ;CP/M entry
LEES    EQU     1
SCHRYF  EQU     2

;Lees een karakter van het toetsenbord
LEESKAR:
        PUSH    HL              ;bewaar registers
        PUSH    BC
```

```
        LD      C,LEES          ;roep operating
        CALL    BDOS            ; systeem aan
        POP     BC
        POP     HL
        RET
;Schrijf een karakter naar het scherm
SCHRKAR:
        PUSH    HL              ;bewaar registers
        PUSH    DE
        PUSH    BC
        LD      C,SCHRYF        ;roep operating
        LD      E,A             ;  systeem aan
        CALL    BDOS
        POP     BC
        POP     DE
        POP     HL
        RET

;Reserveer ruimte voor buffer
BUFFER: DEFS    BUFLEN + 1      ;40 tekens + aantal
        END
```

Om praktische redenen is er, voor wat betreft de verwerking van de delete-toets, een klein verschil tussen het diagram en het programma. Is in het diagram de string leeg dan zal indrukken van de delete-toets geen enkel effect hebben. Door de praktisch altijd aanwezige echo zal het systeem zelf de delete-opdracht naar de uitvoer sturen met als gevolg het wissen van de invoerprompt, het vraagteken. Om dit te herstellen, wordt in een dergelijk geval opnieuw een vraagteken naar het scherm gestuurd.

9.5 Algemene basis I/O subroutines

Na dit hoofdstuk komen 32-bits-integer- en floating pointberekeningen aan de orde. Om met de programma's in de volgende hoofdstukken te werken, moet er een mogelijkheid zijn tot invoer van getallen en weergave van het resultaat. De daarvoor bruikbare algemene I/O subroutines behandelen we hier.

Programma H9P3 Algemene basisinvoer en basisuitvoer

```
;       Basis invoer en uitvoer, bestaande
;       uit de subroutines LEESKAR en SCHRKAR
;
;       LEESKAR wacht tot er een toets wordt
;       ingedrukt en geeft de ASCII waarde daarvan
;       terug in A.
```

```
;       Veranderde registers: AF
;
;       SCHRKAR schrijft het karakter in A naar
;       het beeldscherm.
;       Veranderde registers: F
;


        GLOBAL  LEESKAR,SCHRKAR

;Constanten:
BDOS    EQU     5       ;CP/M entry
LEES    EQU     1
SCHRYF  EQU     2

;Invoer, geeft bij terugkeer karakter in A
LEESKAR:
        PUSH    HL              ;bewaar alle betrokken
        PUSH    BC              ;  registers
        LD      C,LEES          ;roep operating
        CALL    BDOS            ;  systeem aan
        POP     BC              ;herstel registers
        POP     HL
        RET

;Uitvoer, zet karakter in A op het scherm
SCHRKAR:
        PUSH    HL              ;bewaar betrokken
        PUSH    DE              ;  registers
        PUSH    BC
        LD      C,SCHRYF        ;roep operating
        LD      E,A             ;  systeem aan
        CALL    BDOS
        POP     BC              ;herstel registers
        POP     DE
        POP     HL
        RET

        END
```

Allereerst de basis-I/O, het systeemafhankelijke deel. Dit staat in programma H9P3. Het zijn de al bekende subroutines LEESKAR en SCHRKAR. In de vorm waarin ze nu staan, heeft het geen zin ze als programma uit te voeren. Het resultaat is dan een éénmalig doorlopen van LEESKAR.

Een toevoeging ten opzichte van de eerdere versie is:

```
GLOBAL LEESKAR,SCHRKAR
```

GLOBAL is een assembler-directive dat kleinere assemblers niet zullen kennen. Het maakt de labels, of liever de adressen van de labels, beschikbaar voor andere programma's. Dat wil zeggen, in een onafhankelijk van H9P3 ontwikkeld programma kan, bijv. met CALL LEESKAR, een verwijzing naar de labels LEESKAR of SCHRKAR staan. Om dat mogelijk te maken, moet het andere programma aangeven de beide subroutines van buiten te willen betrekken. Dat gebeurt met:

```
EXTERNAL LEESKAR,SCHRKAR
```

Het gehele proces dat zulke verbindingen mogelijk maakt, verloopt als volgt. Stel, de sourcecode van LEESKAR en SCHRKAR staat op schijf onder de naam BASISIO.ASS, waarbij de toevoeging ASS aangeeft dat het een file in assembleertaal is. Een assembler die relocatable code genereert, creëert een file BASISIO.REL waarin de opcode begint op adres 0. Dat is dus in deze fase tevens het adres van label LEESKAR, want daar begint het programma mee.
Bekijk nu programma H9P2 dat strings in- en uitvoert. Stel dat daarin LEESKAR en SCHRKAR niet zijn opgenomen. Om ze toch aan te kunnen roepen, moeten de beide labels aan het begin van het programma als EXTERNAL zijn gedeclareerd. De sourcecode staat op schijf onder de naam INUIT.ASS. De assembler creëert ook hier machinecode beginnend op adres 0 en schrijft deze naar de file INUIT.REL.
Wat echter als de assembler een aanroep voor LEESKAR of SCHRKAR tegenkomt. Deze labels komen nergens in de eerste kolom voor en hebben dus geen adres. Gewoonlijk produceert de assembler dan een foutmelding. Aangezien de labels als EXTERNAL zijn gedeclareerd, gebeurt dat niet. Elke keer dat LEESKAR in de source-tekst voorkomt, zet de assembler er een speciaal, bij de naam LEESKAR horend teken voor in de plaats. Dat geldt voor SCHRKAR eveneens. De volgende fase is het linken van INUIT.REL en BASISIO.REL. De linker begint met INUIT.REL en verplaatst de code naar het gewenste adres, in een CP/M systeem is dat 100H. Veronderstel dat het laatst bezette adres voor INUIT dan 18A is. De linker reloceert dan BASISIO naar adres 18B wat ook het juiste adres is voor het label LEESKAR. Deze waarde voor LEESKAR wordt nu alsnog in het INUIT-gedeelte van het programma ingevuld. Hetzelfde gebeurt voor SCHRKAR.

9.6 Algemene invoer- en uitvoerroutines

Voor de invoer en uitvoer van een string is een iets andere techniek gebruikt dan in programma H9P2. Het aantal tekens in de string staat niet langer voorin de buffer

maar er is een speciaal teken om het einde van de string te markeren, het End Of Line (EOL) teken, wat hier een waarde heeft van −1 ofte wel FF hexadecimaal in 2-complement. De methode heeft het voordeel dat andere programma's die de ingevoerde string verwerken geen tellers of telmechanisme hoeven te gebruiken om te controleren of ze al aan het einde van de string zijn. Evenmin is het voor uitvoer producerende programma's nodig het aantal tekens te tellen.
Het principe van de invoer is te vinden in diagram 9.3 en dat van de uitvoer in diagram 9.4.

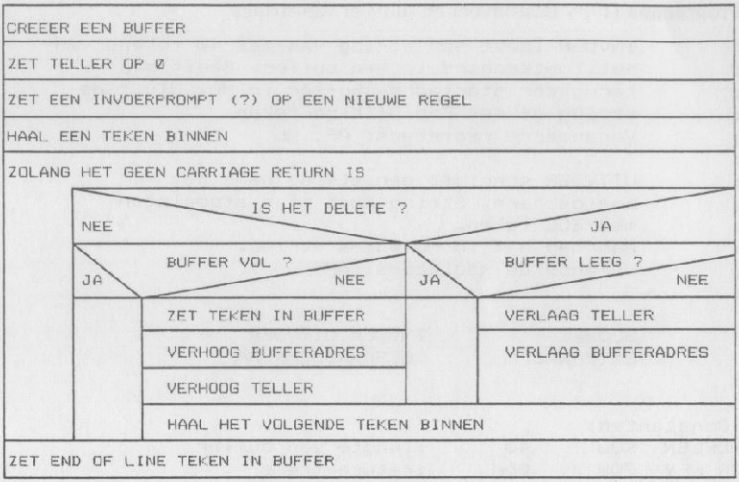


Diagram 9.3 Algemene invoer

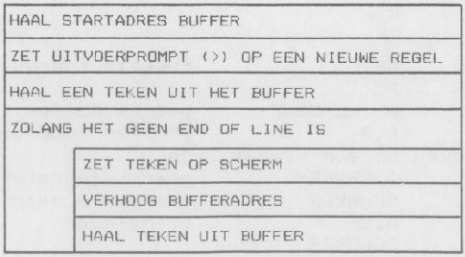


Diagram 9.4 Algemene uitvoer

Aan de hand van de diagrammen laat programma H9P4 zich eenvoudig lezen. Net als dat bij programma H9P2 het geval was, is er om dezelfde reden een klein verschil met betrekking tot het verwerken van de delete-toets.
Omdat het programma de subroutines LEESKAR en SCHRKAR aanroept, zijn deze als EXTERNAL gedeclareerd. Om te kunnen werken, moet het programma gelinkt zijn met de basis I/O. Zelf heeft het programma de labels INVOER en UITVOER als GLOBALS gedeclareerd. Deze zijn dus op hun beurt vanuit andere programma's weer aanroepbaar.

Programma H9P4 Basisinvoer en -uitvoer van strings

```
;       INVOER leest een string van max 40 tekens van
;       het toetsenbord in een buffer. Geeft bij
;       terugkeer startadres buffer in HL. Sluit de
;       string af met End Of Line teken.
;       Veranderde registers: AF, HL
;
;       UITVOER schrijft een string naar het
;       beeldscherm. String moet zijn afgesloten
;       met EOL teken.
;       Aanroep met in HL adres string.
;       Veranderde registers: AF
;

        GLOBAL          INVOER,UITVOER
        EXTERNAL        LEESKAR,SCHRKAR


;Constanten:
BUFLEN  EQU     40      ;lengte van buffer
CR_KEY  EQU     0DH     ;return toets
DEL_KEY EQU     7FH     ;delete toets
LF      EQU     0AH     ;linefeed
EOL     EQU     -1      ;End Of Line teken


INVOER:
;Initialisatie
        PUSH    DE              ;bewaar inhoud reg.
        PUSH    BC
        LD      HL,KLADBUF      ;adres buffer
        LD      B,0             ;teller voor buffer
;Zet vraagteken op een nieuwe lijn
        LD      A,CR_KEY        ;carriage return
        CALL    SCHRKAR         ;kar in A naar scherm
        LD      A,LF            ;linefeed
        CALL    SCHRKAR
INV_?:  LD      A,'?'           ;vraagteken
        CALL    SCHRKAR
```

```
;Haal string binnen
INV_LUS:
        CALL    LEESKAR         ;geeft toets in A
        CP      CR_KEY          ;return toets ?
        JP      Z,INV_EIND      ;ja, naar einde
        CP      DEL_KEY         ;delete toets
        JP      Z,DELETE        ;ja, verwerk
        LD      E,A             ;bewaar toets in E
        LD      A,B             ;teller in A
        CP      BUFLEN          ;buffer al vol ?
        JP      Z,INV_EIND      ;ja, naar einde
        LD      (HL),E          ;teken in buffer
        INC     HL              ;volgend adres buffer
        INC     B               ;update teller
        JP      INV_LUS         ;volgende teken

INV_EIND:
;Zet einde teken in string
        LD      A,EOL           ;einde teken
        LD      (HL),A
        LD      HL,KLADBUF
        POP     BC
        POP     DE
        RET
;Verwerking delete toets
DELETE:
        LD      A,B             ;aantal tekens in A
        CP      0               ;buffer nog leeg ?
        JP      Z,INV_?         ;ja, begin vanaf ?
        DEC     B               ;verminder teller
        DEC     HL              ;vorig adres buffer
        JP      INV_LUS         ;volgende teken
;Declaratie kladbuffer
KLADBUF:
        DEFS    BUFLEN + 1      ;1 meer voor einde teken


UITVOER:
;Initialisatie
        PUSH    HL              ;bewaar registers
        PUSH    BC
;Begin uitvoer op nieuwe regel met '>'
        LD      A,CR_KEY        ;carriage return
        CALL    SCHRKAR
        LD      A,LF            ;linefeed
        CALL    SCHRKAR
        LD      A,'>'
        CALL    SCHRKAR
;Zet string op het scherm
```

```
UIT_LUS:
        LD      A,(HL)          ;teken in string
        CP      EOL             ;laatste teken ?
        JP      Z,UIT_EIND      ;ja, naar einde
        CALL    SCHRKAR         ;naar beeldscherm
        INC     HL              ;adres volgende teken
        JP      UIT_LUS

UIT_EIND:
        POP     BC
        POP     HL
        RET

        END
```

Zo langzamerhand groeit het beeld van de manier waarop ingewikkelde programma's worden ontworpen. Dankzij de methode van het linken, kan men programma's in gedeeltes ontwerpen en uittesten om ze pas in een laatste fase tot een groot geheel samen te voegen.
Als het assemblerpakket deze faciliteiten niet biedt, is het ontwerpen in modules misschien mogelijk met behulp van het commando INCLUDE. Stel, het sourceprogramma voor invoer en uitvoer van een string, programma H9P4, staat op schijf onder de naam INUIT.ASS en de basisinvoer en -uitvoer met de subroutines LEESKAR en SCHRKAR onder de naam BASISIO.ASS. LEESKAR en SCHRKAR kunnen worden toegevoegd aan INUIT.ASS door in de laatste file net voor het END directive het commando INCLUDE BASISIO.ASS op te nemen. Dit tegenkomend zal de assembler niet verdergaan met de tekst van INUIT.ASS maar eerst de tekst van BASISIO.ASS verwerken alsof ze op de plaats van het INCLUDE commando in programma INUIT.ASS was ingetypt. Nadeel van de methode is onder andere dat, naarmate meer modules zo worden 'vastgeplakt', de kans op dubbel gebruikte labels toeneemt.
Biedt het assemblerpakket geen enkele mogelijkheid tot hetkoppelen van modules dan zit er niets anders op dan alle voor een werkend programma benodigde onderdelen in een file te zetten.

10 32-bits-integer-berekeningen

10.1 Rekenprocessors

Wat rekenen betreft, komen de meeste achtbits-processors niet verder dan achtbits-optellen en -aftrekken. De Z-80 biedt daarnaast de mogelijkheid deze basisbewerkingen met 16 bits tegelijk, namelijk met de inhoud van registerparen te doen. Alle andere berekeningen, zoals een vermenigvuldiging van twee 32bits-getallen of een optelling van getallen in floating point notatie, vinden plaats onder controle van software. Dat wil niets anders zeggen dan dat een programma de genoemde 32-bitsvermenigvuldiging laat uitvoeren door de Z-80 een serie basisinstructies op te geven.
De 'nieuwere' zestien- en 32-bits-processors hebben de mogelijkheid tot koppeling aan een speciale rekenprocessor. De Z8070 bijvoorbeeld, een rekenprocessor voor de zestienbits-Z8000 en de 32-bits-Z80000 (de grote broers van de Z-80) kan onder andere op getallen tot 80 bits groot in de floating point notatie de bewerkingen optellen, aftrekken, vermenigvuldigen, delen en worteltrekken uitvoeren.
Komt in een programma zo'n combinatie van processors een rekeninstructie voor dan schuift de microprocessor deze door naar de rekenprocessor en gaat zelf verder met het ophalen van de volgende instructie. Een dergelijke samenwerking van processors geeft heel wat snellere rekenresultaten dan de programma's waarmee wij ons nu gaan bezighouden.

10.2 Het 32-bits-integer-formaat

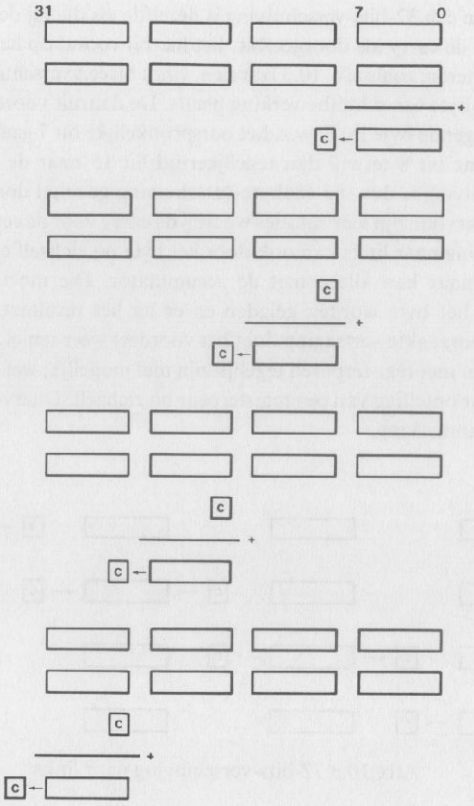
Tot dusver vonden rekenkundige bewerkingen plaats op achtbits-getallen. In dit formaat is het grootste getal 255. Bij gebruik van de 2-complementsnotatie gaat het voorste bit, dat het teken aangeeft, verloren. De grootst mogelijke getallen zijn dan +127 en −128. In een 32-bits-formaat is het grootste getal 4.294.967.295. Is het voorste bit in gebruik voor aanduiding van het teken dan is het grootste negatieve getal −231 en het grootste positieve 231−1. Uitgeschreven komt dat neer op de getallen −2.147.483.648 en +2.147.483.647. Voor menige toepassing ruim voldoende. Nadeel van integer- of gehele getallen is het ontbreken van een

komma. Meestal laat zich dit wel door een foefje ondervangen. Zo kan men financiële berekeningen in centen uitvoeren en het programma dat de uitkomst afdrukt een komma laten toevoegen. Voordeel bij het gebruik van integers is de absolute nauwkeurigheid. Deze ontbreekt bij floating point-getallen die wel een komma toestaan. Bovendien zijn integer-berekeningen door hun eenvoud veel sneller.

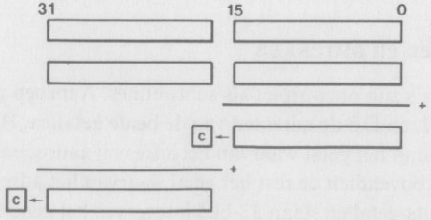
10.2.1 Basisbewerkingen

De voorgaande hoofdstukken lieten zien dat allerlei rekenkundige bewerkingen uitvoerbaar zijn met behulp van de basisbewerkingen optellen, aftrekken en verschuiven. De eerste taak is dan ook een 32-bits-versie van deze basisbewerkingen te maken. Dat gebeurt heel eenvoudig, namelijk door de achtbits-bewerking die in de instructieset van de Z-80 voorkomt op de vier opeenvolgende bytes van het getal uit te voeren. Afb. 10.1 laat een optelling zien. Een reeks van vier bytes vormt hier een 32-bits-getal. De bytes kunnen in registers staan of in opeenvolgende geheugenplaatsen.
Allereerst vindt een optelling van de twee laagste bytes van beide getallen plaats. Het resultaat is de laagste byte van de som. Is de carry na de optelling 1 dan is de betekenis daarvan in het 32-bits-resultaat 2^8. De carry wordt dan ook verwerkt in de optelling van de twee volgende bytes en wel door optelling bij de twee achtste bits van het totaal.

De eerste optelling is dus zonder carry en alle daaropvolgende optellingen zijn met carry. Aangezien de optelling van tweemaal vier bytes in opeenvolgende plaatsen van het geheugen zich bij uitstek leent voor het gebruik van een lus, is het handig ook voor de eerste optelling ADC te gebruiken. In dat geval moet voor de optelling begint de carry 0 zijn.
Ook bij optelling van de twee hoogste bytes kan een carry ontstaan. Zijn de getallen niet in 2-complementsnotatie dan geeft een carry aan dat het getal te groot is. Zijn de getallen wel in 2-complementsnotatie dan kan bij de laatste optelling een overflow worden gesignaleerd.
Het is eveneens mogelijk 32-bits-getallen in registerparen te zetten. Een optelling gebeurt dan, zoals afb. 10.2 laat zien, in twee stappen. Het resultaat van een achtbits-optelling komt altijd in A, die van een tweetal registerparen in HL, IX of IY.
Het spreekt vanzelf dat de aftrekking op dezelfde manier verloopt. Bij gebruik van registerparen is alleen een aftrekking met carry mogelijk en het resultaat komt altijd in HL.

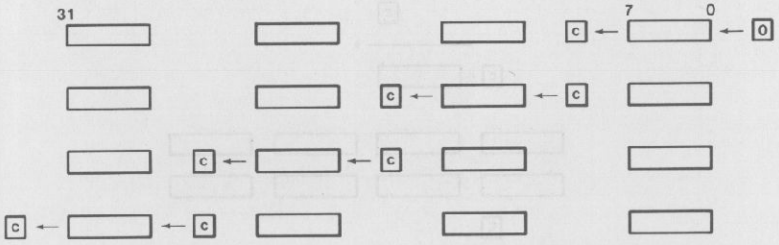


Afb. 10.1 32-bits-optelling



Afb. 10.2 32-bits-optelling met registerparen

De aanpak van een 32-bits-verschuiving is dezelfde als die bij de optelling: byte voor byte met de carry als doorgeefbit. Let hierbij vooral op het verchil tussen schuiven en roteren; zoals afb. 10.3 laat zien, vindt bij een verschuiving naar links op het laagste byte een schuifbewerking plaats. De daaruit voortkomende carry roteert het volgende byte in. D.w.z. het oorspronkelijke bit 7 gaat naar de carry, vervolgens naar bit 8 terwijl dan tegelijkertijd bit 15 naar de carry gaat. De 32-bitsverschuiving is dus een éénbyte-verschuiving gevolgd door drie rotaties. Alternatief daarvoor zijn vier rotaties waarbij de carry voor de eerste 0 moet zijn. Een verschuiving naar links kan ook door het byte bij zichzelf op te tellen. Dat gaat sneller, maar kan alleen met de accumulator. Die moet dus voor elke optelling met het byte worden geladen en er na het resultaat wegzetten. De daardoor veroorzaakte vertraging doet het voordeel weer teniet.
Verschuivingen met registerparen tegelijk zijn niet mogelijk; wel de verschuiving naar links door optelling van een registerpaar bij zichzelf. Daarvoor komen HL, IX en IY in aanmerking.



Afb. 10.3 32-bits-verschuiving naar links

10.3 De 32-bits-integer-rekenprogramma's

10.3.1 Optellen en aftrekken

Alle programma's zijn ontworpen als subroutines. Aanroep geschiedt met in de registerparen HL en DE de adressen van de beide getallen. Het resultaat van de bewerking vervangt het getal waarvan het adres bij aanroep in HL stond. Bij de deling vervangt bovendien de rest het getal waarvan het adres in DE stond.
Net als zestienbits-getallen staan 32-bits-integers in het geheugen met de laagste byte voorop. Voorbeeld voor het hexadecimale getal 01 2D 31 FE:

adres		
	x	FE
	$x+1$	31
	$x+2$	2D
	$x+3$	01

Het volgende hoofdstuk gaat over conversie en eenvoudige expressie-evaluatie. Conversie slaat in dit geval op het omzetten van een ingetypt getal, een ASCII-string dus, in een binair getal. En omgekeerd op het omzetten van het resultaat van een berekening in een ASCII-string voor de uitvoer naar het beeldscherm. De conversieroutines maken gebruik van de eerder behandelde I/O-programma's. De eenvoudige expressie-evaluatie maakt het mogelijk twee getallen en de verlangde bewerking, bijv. vermenigvuldigen, in te typen. Deze routines, gecombineerd met de hier behandelde rekenprogramma's, vormen één geheel.
De basisprincipes voor optellen en aftrekken staan in de diagrammen 10.1 en 10.2. Het aantal te bewerken bytes staat hier in een teller. Door de inhoud daarvan te veranderen, kunnen deze routines dus ook 10 of 24 bytes grote getallen bewerken. In de samenwerking tussen evaluatie, conversie, I/O en berekening staat de evaluator op het hoogste niveau. Deze roept beurtelings conversie- en rekensubroutines aan. Om de evaluator de gelegenheid te geven waarden of adressen in registerparen op te slaan, dienen de aangeroepen routines de registers BC, DE en HL ongewijzigd te laten.

```
ADRESSEN LAAGSTE BYTES VAN TERM 1 EN TERM 2 IN DE EN HL
BEWAAR INHOUD REGISTERS BC, DE EN HL OP DE STAPEL
BYTETELLER := 4
CARRY := 0
ZOLANG DE TELLER NIET 0 IS
    HAAL BYTE TERM 1
    HAAL BYTE TERM 2
    SOM := BYTE TERM 1 + BYTE TERM 2 + CARRY
    BYTE TERM 2 := SOM
    VERHOOG BYTE ADRESSEN IN DE EN HL MET 1
    TELLER := TELLER - 1
HAAL INHOUD REGISTERS HL, DE EN BC VAN STAPEL
```

Diagram 10.1 32-bits-optelling

ADRESSEN LAAGSTE BYTES VAN TERM 1 EN TERM 2 IN DE EN HL
BEWAAR INHOUD REGISTERS BC, DE EN HL OP DE STAPEL
BYTETELLER := 4
CARRY := Ø
ZOLANG DE TELLER NIET Ø IS
HAAL BYTE TERM 1
HAAL BYTE TERM 2
VERSCHIL := BYTE TERM 1 - BYTE TERM 2 - CARRY
BYTE TERM 2 := VERSCHIL
VERHOOG BYTE ADRESSEN IN DE EN HL MET 1
TELLER := TELLER - 1
HAAL INHOUD REGISTERS HL, DE EN BC VAN STAPEL

Diagram 10.2 32-bits-aftrekking

Programma H10P1 32-bits-optellen en aftrekken

```
;Rekenroutines optellen en aftrekken
;
;OPTEL telt twee 4 bytes getallen bij elkaar
;op. Aanroep met in DE en HL de adressen van
;beide termen, laagste byte eerst. Bij terugkeer
;is de tweede term vervangen door de som.
;Veranderde registers: AF


        GLOBAL  OPTEL,AFTREK

OPTEL:
        PUSH    HL              ;bewaar registers
        PUSH    BC
        PUSH    DE
        LD      B,4             ;teller
        AND     A               ;carry = Ø
OPTLUS: LD      A,(DE)          ;byte term 1 in A
        ADC     A,(HL)          ;plus byte term 2
        LD      (HL),A          ;vervangt byte term 2
        INC     HL              ;volgende bytes
        INC     DE
        DJNZ    OPTLUS          ;alle bytes gedaan ?
        POP     DE              ;zet registers terug
        POP     BC
        POP     HL
        RET
```

```
;AFTREK trekt twee 4 bytes getallen van elkaar af.
;Aanroep met DE en HL de adressen van beide
;termen. De routine vervangt de tweede term door
;het verschil.
;Veranderde registers: AF

AFTREK:
        PUSH    HL              ;bewaar registers
        PUSH    BC
        PUSH    DE
        LD      B,4             ;teller
        AND     A               ;carry = 0
AFTLUS: LD      A,(DE)          ;byte term 1 in A
        SBC     A,(HL)          ;trek af byte term 2
        LD      (HL),A          ;verschil vervangt term 2
        INC     HL              ;volgende bytes
        INC     DE
        DJNZ    AFTLUS          ;alle bytes gedaan ?
        POP     DE              ;zet registers terug
        POP     BC
        POP     HL
        RET

        END
```

10.3.2 Vermenigvuldigen

Het voor de vermenigvuldiging gehanteerde principe is bekend, namelijk het naar links schuiven van produkt en vermenigvuldiger en het aan de hand van de carry al dan niet optellen van produkt en vermenigvuldigtal. Zie diagram 10.3. Aangezien de lus 32 maal wordt doorlopen, is het qua snelheid zeer onvoordelig het schuiven en optellen uit te voeren op de inhoud van geheugenlocaties. Het programma zet daarom beide 32bits-getallen in Z-80-registers. Om dat te kunnen doen, moeten de getallen eerst naar een tweetal vierbytes-buffers. Het is namelijk onmogelijk om bijv. IY te laden vanuit een adres dat in HL staat. Vanuit de buffers, waarvan het absolute adres immers bekend is, kan dat wel.

Het programma (H10P2) volgt het diagram naar de letter. De indeling van de gebruikte registerparen is als volgt:

	hoog	laag
vermenigvuldigtal	DE	BC
vermenigvuldiger	HL'	IX
produkt	HL	IY

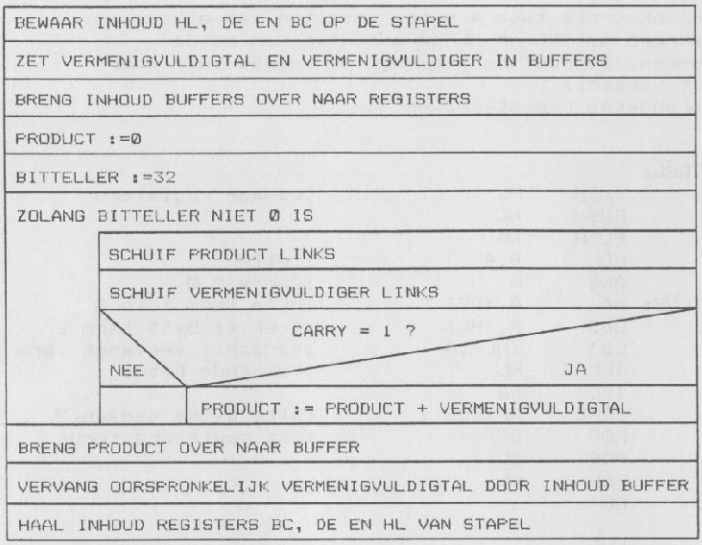


Diagram 10.3 32-bits-vermenigvuldiging

Het waarom van deze verdeling is heel eenvoudig. Het tweemaal links schuiven of roteren van een byte is trager dan het bij zichzelf optellen van een zestienbits-register. De zestienbits-registers waarmee dat kan, zijn IX, IY en HL, alsmede HL' uit de alternatieve registerset. Dat maakt vooral HL' bij uitstek geschikt voor vermenigvuldiger en produkt. Aangezien optelling van IX of IY bij zichzelf alleen zonder carry mogelijk is, moeten in deze registers de lage delen van vermenigvuldiger en produkt komen. De met carry bij zichzelf op te tellen hoge delen staan in HL en HL'.
Eén van de B-registers, B of B', dient als bitteller. De registers DE, DE' en BC (of BC') blijven dan nog over. Daar na een carry de optelling van vermenigvuldigtal en produkt volgt, ligt het voor de hand de hierbij betrokken zestienbits-registers zo te kiezen dat de optelling zonder wisseling van registerset kan plaatsvinden. De keuze voor het vermenigvuldigtal valt daarmee op DE en BC.

Programma H10P2 32-bits-vermenigvuldiging

```
;Subroutine vermenigvuldigen.
;
;VERM vermenigvuldigt twee 4 bytes getallen.
;Aanroep met in HL adres vermenigvuldigtal, in
;DE adres vermenigvuldiger. De subroutine vervangt
;het vermenigvuldigtal door het product.
;Veranderde registers: AF, AF' ,HL', IX, IY


        GLOBAL  VERM


VERM:   PUSH    BC
        PUSH    HL              ;startadres vermenigvuldigtal
        PUSH    DE              ;startadres vermenigvuldiger
;Breng getallen over naar buffers
        LD      DE,VTAL         ;buffer vermenigvuldigtal
        LD      BC,4            ;aantal bytes
        LDIR
        POP     HL              ;startadres vermenigvuldigtal
        PUSH    HL
        LD      DE,VGER         ;buffer vermenigvuldigtal
        LD      BC,4
        LDIR
;Vermenigvuldigtal in registers DE en BC
        LD      BC,(VTAL)       ;laagste deel
        LD      DE,(VTAL+2)     ;hoogste deel
;Maak product in registers HL en IY 0
        LD      IY,0            ;laagste deel
        LD      HL,0            ;hoogste deel
;Vermenigvuldiger in IX en alternatief HL'
        EXX
        LD      IX,(VGER)       ;laagste deel
        LD      HL,(VGER+2)     ;hoogste deel
        LD      B,32            ;teller aantal bits
;Vermenigvuldiging
VERMLUS:
        EXX
        ADD     IY,IY           ;schuif product
        ADC     HL,HL           ; een bit naar links
        EXX
        ADD     IX,IX           ;schuif vermenigvuldiger
        ADC     HL,HL           ; een bit naar links
        EXX
        JP      NC,VRNIET       ;is carry 1 ?
        ADD     IY,BC           ;ja, tel vermenigvuldigtal
        ADC     HL,DE           ; en product op
VRNIET: EXX
        DJNZ    VERMLUS         ;alle bits gedaan ?
        EXX
```

```
;Berg product op in buffer
        LD      (VTAL),IY       ;laagste deel
        LD      (VTAL+2),HL     ;hoogste deel
;Breng product over van buffer naar oorspronkelijk
;adres vermenigvuldigtal
        LD      HL,VTAL         ;startadres product in buffer
        POP     BC
        POP     DE              ;startadres vermenigvuldigtal
        PUSH    DE
        PUSH    BC
        LD      BC,4            ;aantal bytes
        LDIR
        POP     DE              ;registers terug
        POP     HL
        POP     BC
        RET

;Buffers voor vermenigvuldigtal en vermenigvuldiger
VTAL:   DEFS    4
VGER:   DEFS    4

        END
```

10.3.3 De deling

De deling levert meer problemen op. Wat er moet gebeuren, is het positief maken van een negatief getal, elk getal dus waarvan het hoogste bit één is. Uit de tekens van deler en deeltal volgt dan of quotiënt en rest negatief dan wel positief zijn.

deeltal	deler	quotiënt	rest
+	+	+	+
+	−	−	+
−	+	−	−
−	−	+	−

Moeten quotiënt en/of rest negatief zijn dan moeten berekende quotiënt en/of rest alsnog negatief worden gemaakt. Het mechanisme om dit uit te zoeken is vrij eenvoudig. Het begint, zie ook diagram 10.4, met het positief maken van quotiënt en rest. Bij een negatief deeltal draait het de tekens van quotiënt en rest om, bij een negatieve deler alleen dat van het quotiënt. Is deeltal zowel als deler negatief dan draait het teken van het quotiënt tweemaal om (en is daarmee weer positief (+) zoals het hoort) en het teken van de rest draait slechts eenmaal om.
De deling, zie diagram 10.5, volgt een al eerder besproken principe waarbij het

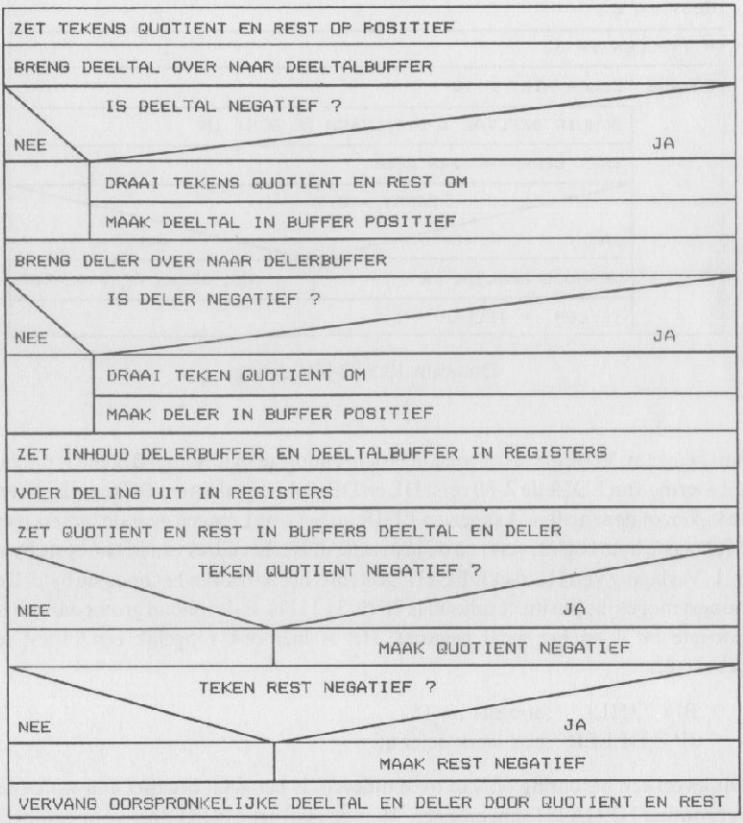


Diagram 10.4 Organisatie rond de 32-bits-deling

deeltal bit voor bit naar links de rest inschuift. Na elke verschuiving wordt gekeken of het mogelijk is de deler van de rest af te trekken.

Door al het extra werk aangaande de tekens is het programma voor de deling een stuk langer dan dat voor de vermenigvuldiging. Aan het begin worden de tekens voor quotiënt en rest op + gezet, niet een ASCII "+" maar het getal +1. Omdraaien van het teken gebeurt dan met de instructie NEG, wat −1 oplevert en na een tweede NEG weer +1.
Op het overbrengen naar de buffers volgt een test op het al dan niet negatief zijn

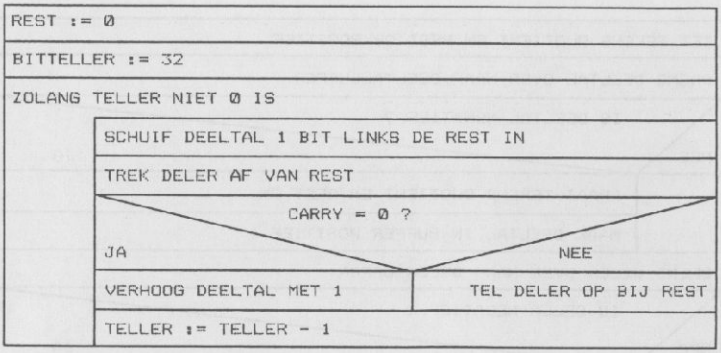


Diagram 10.5 32-bits-deling

van het getal. Voor die test is toegang tot het hoogste byte nodig. Bekend is dat bij uitvoering van LDIR de Z-80 eerst HL en DE verhoogt alvorens BC te verlagen en te kijken of deze nu 0 is. Aangezien LDIR bij het overbrengen naar de buffers met het laagste byte begint, staat na de instructie in HL het adres van het hoogste byte + 1. Verlaging van HL met DEC HL geeft dus het adres van het hoogste byte. De hoogst mogelijke positieve inhoud is 7F (0111 1111). Is de inhoud groter dan is het hoogste bit 1 en het getal negatief. Het is hier ook mogelijk een bittest te gebruiken:

```
BIT 7,(HL)    ;hoogste bit 1?
JP Z,DELER    ;nee, werk deler af
```

Dit levert een besparing op van twee tijdcycli. Is het getal negatief dan wordt de subroutine NEGATE aangeroepen. Wat de instructie NEG doet voor één byte doet deze subroutine voor een vierbytes-getal; namelijk het omdraaien van het teken door het getal van 0 af te trekken. Bij de aanroep van NEGATE moet in HL het adres van het laagste byte staan.
Voor de deling zelf is de bezetting van de registers als volgt:

	hoog	laag
deeltal	BC'	IY
deler	DE	DE'
rest	HL	HL'

De deling vervangt het deeltal door het quotiënt, vandaar dat laatstgenoemde niet in het rijtje voorkomt. Het principe van de deling is gelijk aan dat in hoofdstuk zes.

Nu nog iets over de keuze van de registers. Tijdens elke lus vindt een poging plaats de deler van de rest af te trekken. Levert dat een carry op dan wordt de aftrekking hersteld door de rest en de deler bij elkaar op te tellen. Zestienbits-aftrekken en -optellen kan alleen met HL en HL'. Deze zestienbits-registers bevatten dan ook de rest. Het ligt voor de hand de registers voor de deler in dezelfde set te kiezen. En aangezien B als teller moet fungeren, lijken DE en DE' een goede keuze.
Het deeltal moet naar links schuiven. Voor wat betreft het lage deel kan dat door optelling van IY bij zichzelf. De daarbij eventueel ontstane carry moet naar het hogere deel maar er zijn nu geen registers meer over om dat met een optelling te doen. Vandaar dat BC' byte voor byte links wordt geroteerd. In plaats van BC' had de keuze ook op BC kunnen vallen. Echter, na verschuiving van BC' naar links moet de carry naar links de rest in schuiven. Daar het lage deel van de rest in de alternatieve set staat, betekent de keuze voor BC' in plaats van BC een wisseling van registerset minder.
Na de deling gaan quotiënt en rest naar de oorspronkelijk voor deeltal en deler gebruikte buffers. De deling is uitgevoerd met positieve getallen. Is het teken voor quotiënt en/of rest negatief dan volgt een aanroep van NEGATE. Ten slotte vervangen quotiënt en rest, nu voorzien van het juiste teken, deeltal en deler op de adressen waarin zich eerder de aanvangswaarden voor de delingssubroutine bevonden.

Programma H10P3 32-bits-deling

```
;Subroutine delen.
;Deelt twee 4 bytes getallen.
;Aanroep met in HL adres deeltal en in DE
;adres deler. De routine vervangt het deeltal
;door het quotient en de deler door de rest.
;Veranderde registers: IY, BC', DE', HL'


        GLOBAL  DELEN


DELEN:
;Bewaar inhoud registers
        PUSH    BC
        PUSH    HL              ;adres deeltal
        PUSH    DE              ;adres deler
;Zet tekens quotient en rest op positief
        LD      A,+1
        LD      (TEKENQ),A
        LD      (TEKENR),A
;Breng deeltal over naar buffer
```

```
        LD      DE,DTALBF       ;adres buffer deeltal
        LD      BC,4            ;aantal bytes
        LDIR
;Als deeltal negatief is teken en deeltal veranderen
        DEC     HL              ;adres hoogste byte deeltal
        LD      A,7FH
        CP      (HL)            ;deeltal negatief ?
        JP      NC,DELER        ;nee, werk deler af
        LD      A,(TEKENØ)
        NEG                     ;draai tekens om
        LD      (TEKENØ),A
        LD      (TEKENR),A
        LD      HL,DTALBF       ;adres deeltalbuffer
        CALL    NEGATE          ;maak deeltal positief
;Breng deler over naar buffer
DELER:  POP     HL              ;adres deler
        PUSH    HL
        LD      DE,DELERBF      ;adres buffer deler
        LD      BC,4
        LDIR
;Als deler negatief is teken en deler veranderen
        DEC     HL              ;adres hoogste byte deler
        LD      A,7FH
        CP      (HL)            ;deler negatief ?
        JP      NC,DELING       ;nee, begin deling
        LD      A,(TEKENØ)
        NEG                     ;draai teken quotient om
        LD      (TEKENØ),A
        LD      HL,DELERBF      ;adres delerbuffer
        CALL    NEGATE          ;maak deler positief
DELING:
;Zet deeltal in BC' (hoog) en IY (laag), de deler in
;DE (hoog) en DE' (laag). Maak rest in HL (hoog) en
;HL' (laag) nul. Zet bitteller op 32.
        LD      IY,(DTALBF)     ;lage deel deeltal
        LD      DE,(DELERBF+2)  ;hoge deel deler
        AND     A               ;carry Ø
        SBC     HL,HL           ;hoge deel rest Ø
        LD      B,32            ;bitteller
        EXX
        LD      BC,(DTALBF+2)   ;hoge deel deeltal
        LD      DE,(DELERBF)    ;lage deel deler
        SBC     HL,HL           ;lage deel rest Ø
        EXX
;Voer deling uit
DEELLUS:
        EXX
        ADD     IY,IY           ;schuif deeltal links
        RL      C
        RL      B
        ADC     HL,HL           ;in rest
        EXX
        ADC     HL,HL
```

```
        EXX
        SBC     HL,DE           ;trek deler af van rest
        EXX
        SBC     HL,DE
        INC     IY              ;verhoog quotient
        JP      NC,DEELEIND     ;was aftrekken mogelijk ?
        DEC     IY              ;nee, maak ongedaan
        EXX
        ADD     HL,DE           ;tel deler op bij rest
        EXX
        ADC     HL,DE
DEELEIND:
        DJNZ    DEELLUS         ;alle bits gehad ?
;Zet quotient in BC' (hoog) en IY (laag) en rest in
;HL (hoog) en HL' (laag) in buffers.
        EXX
        LD      (DTALBF+2),BC
        LD      (DELERBF),HL
        EXX
        LD      (DELERBF+2),HL
        LD      (DTALBF),IY
;Kijk in teken of uitkomst negatief moet zijn
        LD      HL,DTALBF       ;adres quotient
        LD      A,(TEKENQ)
        CP      +1              ;teken negatief ?
        CALL    NZ,NEGATE       ;ja, maak quotient negatief
        LD      HL,DELERBF      ;adres rest
        LD      A,(TEKENR)
        CP      +1              ;teken negatief ?
        CALL    NZ,NEGATE       ;ja, maak rest negatief

;Breng inhoud van de buffers over naar oorspronkelijke
;deler en deeltal.

        POP     DE              ;adres deler
        PUSH    DE
        LD      HL,DELERBF
        LD      BC,4
        LDIR
        POP     BC
        POP     DE              ;adres deeltal
        PUSH    DE
        PUSH    BC
        LD      HL,DTALBF
        LD      BC,4
        LDIR
;Zet inhoud registers terug
        POP     DE
        POP     HL
        POP     BC
        RET
```

```
TEKENQ: DEFB    Ø                ;teken quotient
TEKENR: DEFB    Ø                ;teken rest
DTALBF: DEFS    4                ;buffer voor deeltal
DELERBF:
        DEFS    4                ;buffer voor deler

;Inverteer teken van getal
NEGATE: AND     A                ;carry Ø
        LD      B,4              ;byteteller
NEGLUS: LD      A,Ø              ;trek getal af van Ø
        SBC     A,(HL)
        LD      (HL),A           ;resultaat vervangt getal
        INC     HL               ;volgende byte
        DJNZ    NEGLUS           ;alle bytes gedaan ?
        RET

        END
```

Het is nu mogelijk een, zij het bescheiden, library (bibliotheek) van 32-bits-integer-rekenfuncties te maken, op voorwaarde dat het assemblerpakket die optie biedt. Wat er dan gebeurt, is het volgende. Zoals bekend genereert de voor dit boek gebruikte assembler allereerst relocatable code vanaf adres 0. De drie programma's uit dit hoofdstuk leveren een drietal files op met als namen bijv. OPAF.REL voor optellen en aftrekken, MAAL.REL en DEEL.REL. De library-optie plakt de files aan elkaar en zet het geheel weg op schijf onder bijvoorbeeld de naam REKEN.LIB. In die file staan ook alle in de programma's toegekende GLOBALS. Is er nu later een vermenigvuldigingsroutine nodig dan hoeft die niet opnieuw te worden geschreven. Het is dan voldoende vanuit dit nieuwe programma het label VERM aan te roepen, natuurlijk met de adressen van de getallen in de juiste registerparen, en VERM zelf als EXTERNAL te declareren. Om dan de vermenigvuldigingsroutine ook echt in het nieuwe programma op te nemen, moet dat programma worden gelinkt met de library REKEN.LIB. In het algemeen kan het linken dan plaatsvinden met een speciale zoek-optie zodat de linker alleen de file met de aangegeven GLOBAL(s) uit de library haalt en aan het nieuwe programma toevoegt.

11 Integer-conversies en eenvoudige expressie-evaluatie

11.1 ASCII-binair-conversie

Invoer heeft betrekking op strings, dat wil zeggen reeksen ASCII-codes van letters, cijfers en tekens die op het toetsenbord zijn aangeslagen. De rekenprogramma's uit het vorige hoofdstuk werken met vierbytes-integer-getallen. Om een rekenkundige bewerking op ingetypte getallen te laten uitvoeren, zal een omzetting moeten plaatsvinden van een reeks ASCII-codes voor cijfers naar een vierbytes-getal.
Het aanbod aan de ASCII-binair-conversieroutine gebeurt in de vorm van een string. Deze bestaat uit de al genoemde reeks ASCII-codes voor cijfers, eventueel voorafgegaan door een + of een − teken.
Om de essentie van de conversie te begrijpen, is het nodig goed doordrongen te zijn van de wijze waarop een getal is opgebouwd. Neem als voorbeeld het getal 1625. De vier cijfers hebben de volgende betekenis:

$1 \times 10^3 = 1 \times 10 \times 10 \times 10$
$6 \times 10^2 = 6 \times 10 \times 10$
$2 \times 10^1 = 2 \times 10$
$5 \times 10^0 = 5$

De string "1625" heeft in ASCII-code de volgende vorm:

hex	31 = ASCII-code voor	1
	36	6
	32	2
	35	5

Het basisprincipe van de conversie is te vinden in diagram 11.1. Daarbij zij opgemerkt dat de omzetting van cijfers tot een getal plaatsvindt tot in de string de code voor een niet-cijfer staat. Dat kan een End Of Line teken zijn maar ook, zoals bij de nog te bespreken expressie-evaluator, een spatie of codes als + en / die een uit te voeren bewerking aangeven.
Het eerste ASCII-teken in de string is 31. Verminderd met de ASCII-code voor '0' (= 30 hex) blijft de binaire waarde van het cijfer, en wel 1, over. Deze wordt

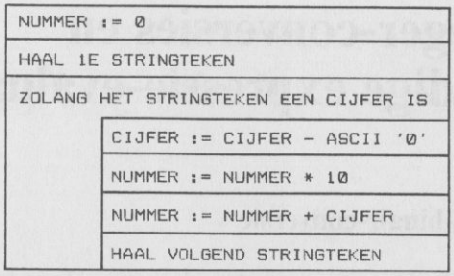


Diagram 11.1 Basisprincipe ASCII-binair-conversie

opgeteld bij het in eerste instantie 0 gemaakte getal. Het volgende stringteken is weer een ASCII-cijfer. Optelling hiervan bij het getal vindt pas plaats nadat het getal met 10 is vermenigvuldigd. De opeenvolgende waarden van het getal zijn:

na verwerking	getal
1e teken	1
2e teken	$1 \times 10 + 6$
3e teken	$1 \times 10 \times 10 + 6 \times 10 + 2$
4e teken	$1 \times 10 \times 10 \times 10 + 6 \times 10 \times 10 + 2 \times 10 + 5$

Aan de cijfers in de string kan een '+' of '−' teken voorafgaan. De omzetting zelf echter is altijd naar een positief getal. Pas na de conversie wordt, als het teken een '−' was, het getal negatief gemaakt, eenvoudig door het 2-complement ervan te nemen.

Diagram 11.2 geeft het overzicht van de totale ASCII naar binairconversie. Het programma is ontwikkeld als subroutine. Bij aanroep ervan staat in HL het startadres van de string, binnengehaald met de I/O-subroutine INVOER, en in DE het startadres voor het te creëren getal. Bij terugkeer staat in HL het adres van het eerste, niet meer tot het getal behorende stringteken.

Het eigenlijke programma, de subroutine ASC_BIN, beslaat het eerste deel van het complete conversieprogramma H11P1. Het programma bouwt het getal op in de vierbytes-buffer ASCNUM.

Voor het vermenigvuldigen met 10 was het mogelijk geweest gebruik te maken van de al ontwikkelde vermenigvuldigingssubroutine VERM. De uiteindelijke vermenigvuldiging daar bestaat echter uit een 32 maal te doorlopen lus terwijl viermaal, de binaire vorm van 10 is 1010, voldoende zou zijn. Om aan snelheid te winnen, is het handig om een aparte MAAL10-subroutine te maken. Deze werkt als volgt:

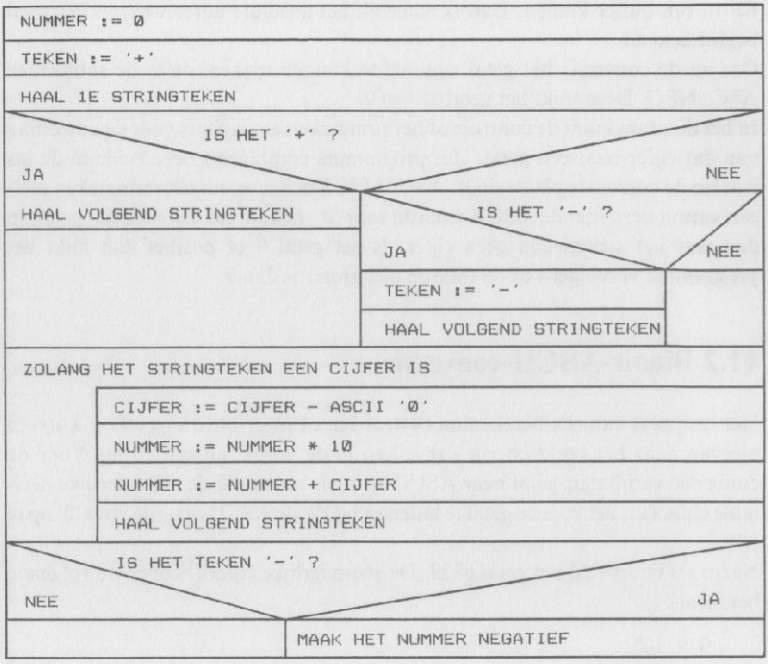


Diagram 11.2 ASCII-binair-conversie

Tel het getal op bij zichzelf. Dat geeft getal × 2.
Zet een kopie van het resultaat apart.
Tel getal × 2 op bij zichzelf. De som is getal × 4.
Tel getal × 4 op bij zichzelf. De som is getal × 8.
Tel getal × 8 op bij apart gezette getal.

Resultaat: (8 + 2) × getal = 10 × getal.

De complete vermenigvuldiging is gerealiseerd met een viertal optellingen. De MAAL10 subroutine telt tevens het in een getal omgezette ASCII-cijfer bij het getal op.
Het is bij de Z-80 niet mogelijk de inhoud van een registerpaar op te slaan in een adres dat in een ander registerpaar staat. Om daarom het getal over te kunnen brengen naar het startadres dat bij aanroep van de subroutine in DE staat, moet

het in een buffer komen. Dan is namelijk het absolute adres waarop het getal begint bekend.
Om na de conversie het getal negatief te kunnen maken, dient de subroutine ASC_NEG. Deze trekt het getal af van 0.
In het diagram komt de controle of het stringteken een cijfer is voor de omzetting van dat cijfer naar een getal. Het programma combineert deze beide in de lus waarin de conversie plaatsvindt: ASC_LUS. Daarin wordt het stringteken met-een verminderd met de ASCII-waarde voor '0'. Levert dat een negatief getal op dan was het stringteken geen cijfer. Is het getal 0 of positief dan kijkt het programma vervolgens of de waarde niet groter is dan 9.

11.2 Binair-ASCII-conversie

Het resultaat van een berekening bestaat één of meer binaire getallen. Uitvoer hiervan naar het beeldscherm kan alleen in de vorm van een string. Voor de conversie van binair getal naar ASCII-string is het nodig de afzonderlijke decimale cijfers uit het binaire getal te halen en bij elk de ASCII-waarde voor '0' op te tellen.
Neem als voorbeeld het getal 6529. De afzonderlijke cijfers hebben de volgende betekenis:

9×10^0
2×10^1
5×10^2
6×10^3

De conversie bestaat uit het steeds weer door 10 delen van het binaire getal. De rest die bij elke deling ontstaat, is het laagste cijfer van het getal voor de deling.

na deling	getal	rest
1	652	9
2	65	2
3	6	5
4	0	6

Het principe van de conversie is weergegeven in diagram 11.3. De enige moeilijkheid is dat de cijfers in de verkeerde volgorde ontstaan. Het laagste cijfer komt het eerst. Een oplossing zou kunnen zijn de string van achter naar voren te vullen. Omdat niet bekend is hoeveel cijfers elk getal in beslag neemt, is de af te drukken string altijd zo groot als het maximum aantal cijfers. In de na conversie overblij-

vende stringruimte moeten spaties staan. Deze methode geeft een mooie geformatteerde uitvoer voor reeksen getallen waarbij de laagste cijfers netjes onder elkaar staan.

```
                NUMMER := NUMMER/10 + REST
                CIJFER := REST + ASCII '0'
                ZET CIJFER WEG
HERHAAL TOT NUMMER = 0
```

Diagram 11.3 Principe van de binair-ASCII-conversie

Een andere aanpak is de volgorde van de cijfers omdraaien. Dat kan bijv. door de cijfers in de volgorde van ontstaan in een hulpstring te zetten en deze dan omgekeerd naar de uit te voeren string te kopiëren.
Omkering is ook mogelijk via de stapel. In dat geval gaat na deling het ontstane cijfer naar de stapel. Is de conversie klaar dan worden de cijfers van de stapel gehaald. Het hoogste cijfer komt dan eerst. Ogenschijnlijk noodzaakt deze methode het via een teller bijhouden van het aantal cijfers dat naar de stapel gaat.

```
BRENG NUMMER OVER NAAR NUMMERBUFFER
1E TEKEN STRINGBUFFER := '+'
        IS HET NUMMER NEGATIEF ?
NEE                                    JA
                        MAAK NUMMER POSITIEF
                        1E TEKEN STRINGBUFFER := '-'
BRENG NUMMER OVER NAAR REGISTERS
        NUMMER := NUMMER/10 + REST
        CIJFER := REST + ASCII '0'
        ZET CIJFER OP STAPEL
HERHAAL TOT NUMMER = 0
ZET CIJFERS VAN STAPEL IN STRINGBUFFER
VOEG END OF NUMBER TEKEN TOE
ZET INHOUD STRINGBUFFER OVER NAAR STRING
```

Diagram 11.4 Binair-ASCII-conversie

Niet meer en niet minder dan dat aantal moet immers weer van de stapel. Hoe dit te omzeilen is, komt nog aan de orde.
Diagram 11.4 geeft een overzicht van de volledige conversie. Het programma voor de ASCII-binair-omzetting heeft de vorm van een subroutine. Bij aanroep ervan staat het adres van het binaire getal in DE. Het getal moet dus, om inmiddels bekende redenen naar een buffer. Tevens staat bij aanroep het startadres van de string in HL. De string wordt eerst ontwikkeld in een buffer en daarna overgebracht naar de uit te voeren string.

Het programma vormt het tweede deel van het complete conversieprogramma H11P1. Aangezien de conversie betrekking heeft op positieve getallen, roept het programma bij een negatief getal de ook bij ASCII-binair-omzetting gebruikte subroutine ASC_NEG aan. Deze keert het teken van het getal in de buffer ASCNUM om. Beide routines gebruiken dus voor het getal ditzelfde buffer.
Nadat het juiste teken in de stringbuffer ASCSTR is gezet, volgt de omzettingsroutine BIN_OMZ. Deze roept de subroutine BIN_REC aan met het getal in de registerparen HL en DE. Bij terugkeer uit deze subroutine staan de cijfers in de juiste volgorde in de string. BIN_OMZ voegt daaraan een End Of Number-teken toe. Daarna verplaatst een simpele lus alle tekens tot en met EON van de stringbuffer naar de string. EON heeft trouwens dezelfde waarde als het al eerder gebruikte EOL.
Voor de werking van de BIN_REC aan de orde komt eerst nog de speciale subroutine DEEL10. Voor de deling door 10 was het mogelijk geweest de al ontwikkelde delingsroutine te gebruiken. Nodig echter is alleen een deling van een positief getal door 10 zodat een veel korter en dus sneller programma mogelijk is. Er hoeft namelijk geen controle op tekens plaats te vinden en daar de deler één byte groot is, kan de telkens uitgevoerde aftrekking een éénbyte-operatie zijn. Bovendien kan het binaire getal in de registerparen blijven staan. Voor elke aanroep van de subroutine DELEN, die de deler vervangt door de rest, zou opnieuw een vierbytes-voorstelling van 10 in een buffer moeten worden geplaatst. De DEEL10-subroutine schuift tijdens elke lus het getal één bit links in register C. In register A vindt een poging plaats om hiervan 10 af te trekken. Is aftrekking mogelijk dan ontstaat geen carry en de nieuw ontstane rest gaat van C naar A. Ontstaat er wel een carry dan slaat het programma de laatste verplaatsing over zodat C ongewijzigd blijft. Voor de test op een carry inverteert CCF echter de inhoud hiervan. Indien er kan worden afgetrokken, is na CCF de carry 1. Tijdens de eerstvolgende lus wordt de carry in het deeltal geroteerd en vormt zo het quotiënt. Aan het einde van het programma moet een extra rotatie de laatste carry in het quotiënt roteren. Hoewel het verschuiven van een vierbytes-getal sneller

kan met twee zestienbits-optellingen is hier, om in een voorbeeld het al eerder besproken principe te laten zien, een roteerinstructie gebruikt.
De subroutine BIN_REC verzorgt de eigenlijke omzetting. De werking ervan is in de vorm van een aparte module weergegeven in diagram 11.5.
Tijdens het omzetten roept de subroutine zichzelf aan. De benaming voor een dergelijk proces is recursie. Gebruik van recursie kan leiden tot verrassend korte programma's en tevens tot hoofdpijn want wat er gebeurt is vaak nogal gecompliceerd. We volgen daarom het proces stap voor stap.

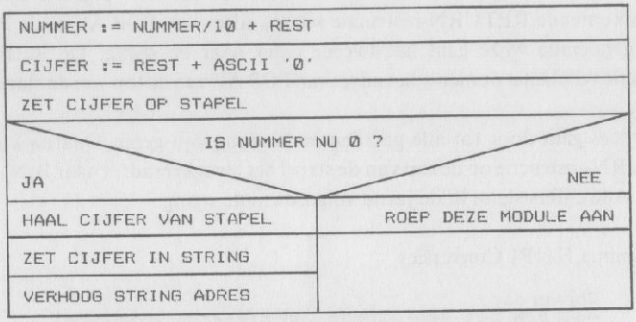


Diagram 11.5 Recursieve ASCII-binair-omzetting

De eerste aanroep van BIN_REC gebeurt vanuit BIN_OMZ en het het adres van BIN_OMZ komt dan ook bovenop de stapel.

BIN_REC berekent het laatste cijfer voor de uitvoer, zet dat op de stapel en kijkt dan of het getal 0 is door een OR van alle bytes. Levert dat 0 op dan is het getal 0. Is het getal niet 0 dan roept BIN_REC zichzelf aan. Het terugkeeradres, in dit geval dus het adres van de instructie volgend op de call, POP AF, gaat naar de stapel. Is het getal bijv. 2597 dan ziet de stapel op het moment dat het getal 0 is er als volgt uit:

terugkeeradres naar BINOMZ (adres van LD A,EON)

ASCII 7 plus op de stapel gezette vlag

terugkeeradres naar POP AF in BINREC

ASCII 9 etc.
terugkeeradres naar POP AF

ASCII 5 etc.

terugkeeradres naar POP AF

ASCII 2 etc.

Daar het getal nu 0 is, roept de routine niet zichzelf aan maar gaat door naar POP AF. Dit haalt het voorste cijfer van de stapel naar A. Het cijfer gaat naar de string en vervolgens verhoogt INC BC het stringadres voor het volgende cijfer. De hierna komende RETURN-instructie zet het adres van POP AF in de PC. Op bovengenoemde wijze gaat het tweede cijfer naar de string. De RETURN-instructie verplaatst opnieuw het adres van POP AF van de top van de stapel naar de PC.
Dit proces gaat door tot alle getallen in de string zijn gezet. Daarna komt de RETURN-instructie op de top van de stapel het terugkeeradres naar BIN_OMZ tegen. Alle cijfers staan in de juiste volgorde in de string.

Programma H11P1 Conversies

```
;       Conversie
;       ASC_BIN zet een string van cijfers, eventueel
;       voorafgegaan door + of -, om in een binair
;       getal van 32 bits.
;       Aanroep met in HL adres string, in DE adres
;       nummer. Geeft in HL adres 1e niet tot het nummer
;       behorend karakter terug.
;       Veranderde registers: HL, AF, IX, IY
;
;       BIN_ASC zet een binair getal om in een string
;       van cijfers, eventueel voorafgegaan door +
;       of -. Sluit de string af met End Of Number teken.
;       Aanroep met in HL adres string, in DE adres
;       nummer.
;       Veranderde registers: AF



        GLOBAL  ASC_BIN,BIN_ASC

ASC_BIN:
        PUSH    BC              ;bewaar registers
        PUSH    DE
;Maak nummer 0
        PUSH    HL              ;bewaar HL
        LD      HL,0
        LD      (ASCNUM),HL     ;ASCNUM is buffer voor
        LD      (ASCNUM+2),HL   ;creeren nummer
        POP     HL
```

```
;Maak teken +
        LD      A,'+'
        LD      (TEKEN),A
;Start conversie
        CP      (HL)            ;1e teken + ?
        JP      NZ,ASCGP        ;nee, dan naar ASCii Geen Plus
        INC     HL              ;ja, adres volgende teken
        JP      ASC_LUS         ;naar conversie lus
ASCGP:  LD      A,'-'
        CP      (HL)            ;1e teken - ?
        JP      NZ,ASC_LUS      ;nee, naar conversie lus
        LD      (TEKEN),A       ;ja, pas TEKEN aan
        INC     HL              ;adres volgende teken
ASC_LUS:
        LD      A,(HL)          ;teken string
        SUB     '0'             ;maak van ASCII cijfer getal
        JP      M,ASC_EIND      ;kleiner dan 0 ?: naar einde
        CP      10              ;groter dan 9 ?
        JP      P,ASC_EIND      ;ja, naar einde
        PUSH    HL              ;bewaar adres stringteken
        CALL    MAAL10          ;nummer maal 10 + nieuwe getal
        POP     HL
        INC     HL              ;adres volgende teken
        JP      ASC_LUS
ASC_EIND:
        LD      A,(TEKEN)
        CP      '-'             ;stond er een - voor ?
        CALL    Z,ASC_NEG       ;ja, maak ASCNUM negatief
        POP     DE              ;adres nummer
        PUSH    DE
        PUSH    HL
        LD      HL,ASCNUM       ;breng ASCNUM over naar
        LD      B,0             ;NUMMER
        LD      C,4
        LDIR
        POP     HL              ;adres in string
        POP     DE              ;adres nummer
        POP     BC
        RET


ASC_NEG:
        PUSH    HL              ;bewaar adres string
        XOR     A               ;maak carry 0
        LD      HL,0            ;trek nummer af van 0
        LD      DE,(ASCNUM)     ;laagste 2 bytes eerst
        SBC     HL,DE
        LD      (ASCNUM),HL
        LD      HL,0
        LD      DE,(ASCNUM+2)
        SBC     HL,DE
        LD      (ASCNUM+2),HL
        POP     HL
        RET
```

```
MAAL10:
        LD      IY,(ASCNUM)     ;nummer in IY en HL
        LD      HL,(ASCNUM+2)   ;laagste 2 bytes in IY
        ADD     IY,IY           ;nummer maal 2
        ADC     HL,HL
        LD      E,L             ;berg nummer * 2 op
        LD      D,H
        PUSH    IY              ;in BC en DE
        POP     BC
        ADD     IY,IY           ;nummer maal 4
        ADC     HL,HL
        ADD     IY,IY           ;nummer maal 8
        ADC     HL,HL
        ADD     IY,BC           ;nummer maal 10
        ADC     HL,DE
        LD      C,A             ;nieuwe getal in BC
        LD      B,0
        ADD     IY,BC           ;optellen bij nummer * 10
        LD      C,0
        ADC     HL,BC
        LD      (ASCNUM),IY     ;berg getal op
        LD      (ASCNUM+2),HL
        RET

ASCNUM: DEFS    4               ;buffer voor opbouw nummer
TEKEN:  DEFB    0               ;ruimte voor teken
ASCSTR: DEFS    15              ;buffer voor opbouw string

BIN_ASC:
;Constanten:
EON     EQU     -1              ;End Of Number=EOL
;Bewaar inhoud registers
        PUSH    BC
        PUSH    DE              ;adres nummer
        PUSH    HL              ;adres string
;Breng nummer over naar buffer
        EX      DE,HL           ;in HL adres nummer
        LD      DE,ASCNUM       ;in DE adres buffer
        LD      C,4
        LD      B,0
        LDIR
;Zet teken op positief
        LD      A,'+'
        LD      (ASCSTR),A
;Verwerk een eventueel negatief nummer
        DEC     DE              ;adres belangrijkste byte nummer
        LD      A,(DE)
        CP      80H             ;nummer negatief ?
        JP      C,BIN_OMZ       ;nee, naar omzetting
        CALL    ASC_NEG         ;trek nummer af van 0
        LD      A,'-'
        LD      (ASCSTR),A
```

```
;Omzetting
BIN_OMZ:
        LD      HL,(ASCNUM)     ;nummer in HL en DE
        LD      DE,(ASCNUM+2)   ;laagste byte in HL
        CALL    BIN_REC         ;zet nummer om in string
        LD      A,EON           ;voeg EON toe
        LD      (BC),A
;Verplaats inhoud stringbuffer naar string
        POP     HL              ;adres string
        PUSH    HL
        LD      DE,ASCSTR       ;adres stringbuffer
BIN_EIND:
        LD      A,(DE)
        LD      (HL),A
        INC     HL
        INC     DE
        CP      EON             ;laatste teken in string ?
        JP      NZ,BIN_EIND     ;nee, volgend teken
        POP     HL              ;ja, zet inhoud registers
        POP     DE              ; terug
        POP     BC
        RET

;Zet het nummer recursief om in een string
BIN_REC:
        CALL    DEEL10          ;deel nummer door 10, rest in A
        ADD     A,'0'           ;maak van rest cijfer
        PUSH    AF              ;bewaar cijfer
        LD      BC,ASCSTR+1     ;adres 1e cijfer in str. buffer
        XOR     A               ;inhoud A=0
        OR      L               ;nummer nu 0 ?
        OR      H
        OR      E
        OR      D
        CP      0
        CALL    NZ,BIN_REC      ;nee, dan opnieuw
        POP     AF              ;cijfer van stapel
        LD      (BC),A          ;in stringbuffer
        INC     BC              ;volgende cijfer
        RET
;Deel nummer in buffer door 10 en geef rest terug in A
DEEL10:
        LD      C,0             ;hierin opbouw rest
        LD      B,32            ;bitteller
        AND     A               ;carry 0
DEELLUS:
        RL      L               ;nummer 1 bit links
        RL      H
        RL      E
        RL      D
        RL      C               ;schuif bit in C
        LD      A,C             ;rest in A
        SUB     10              ;rest - 10 mogelijk ?
        CCF                     ;inverteer carry
```

```
        JP      NC,DEELEIND     ;nee, volgende bit
        LD      C,A             ;rest in C
DEELEIND:
        DJNZ    DEELLUS         ;32 bits gedaan ?
        RL      L               ;ja, roteer laatste carry
        RL      H               ; in getal
        RL      E
        RL      D
        LD      A,C             ;rest in A
        RET

        END
```

12 Floating point-berekeningen

12.1 Algemeen

Alle rekenprogramma's tot dusver bewerkten gehele getallen. Voor de meeste praktische gevallen zijn gebroken getallen nodig, getallen dus met een komma erin, zoals bijv. een bedrag van f12,75 of een inhoud van 0,5 liter. In veel landen gebruikt men in plaats van een decimale komma een decimale punt (Engels: point). De notatie voor bovenstaand bedrag is dan 12.75. In dit boek zullen we deze schrijfwijze volgen.
De floating point-notatie is een methode om gebroken getallen in binaire vorm weer te geven. In feite is deze manier van werken heel eenvoudig. De essentie ervan laat zich vangen in onderstaand rijtje getallen.

575	0.575	$\times 10^3$
−18.7	−0.187	$\times 10^2$
0.2	0.2	$\times 10^0$
0.0046	0.46	$\times 10^{-2}$

De getallen links zijn rechts in een uniforme schrijfwijze uitgedrukt. Elk getal is opgeschreven met een deel voor de decimale punt gelijk aan nul, een deel na de decimale punt waarvan het eerste cijfer niet gelijk is aan nul een macht van 10. Het eerste deel van het getal, bijv. 0.575, heet de mantisse. Om tot de waarde van het hele getal te komen, hoeft alleen nog de exponent van 10 bekend te zijn. Het getal 575 ligt dus vast met de mantisse 0.575 en de exponent 3.
Om van 575.0×10^0 naar 0.575×10^3 te komen, schuift men 575 drie plaatsen naar rechts tot het geheel achter de decimale punt staat. Bij elke verschuiving hoort een verhoging van de exponent.

575.0	$\times 10^0$	$=575$
57.50	$\times 10^1$	$=575$
5.750	$\times 10^2$	$=575$
0.575	$\times 10^3$	$=575$

Het getal 0.0046 schuiven we twee plaatsen naar links. Bij een verschuiving naar links hoort een verlaging van de exponent.
Hetzelfde proces laat zich toepassen op binaire getallen. Als voorbeeld de binaire voorstelling van het decimale getal 4:

$$
\begin{array}{lll}
100.0 & \times 2^0 & = 100 \\
10.00 & \times 2^1 & = 100 \\
1.000 & \times 2^2 & = 100 \\
0.100 & \times 2^3 & = 100
\end{array}
$$

Met andere woorden: de mantisse is 0.1 en de exponent 3. Voor het getal 14 zijn deze respectievelijk 0.111 en 4, aangezien 14 in het binaire stelsel 1110 is en dus viermaal naar rechts moet worden geschoven om achter de binaire punt te belanden.

Het is zinnig om even na te gaan wat in het decimale en binaire stelsel de waarden van cijfers na de punt is. De waarde van de cijfers in 874.25 is:

$$
\begin{array}{ll}
8 & 8 \times 10^2 \\
7 & 7 \times 10^1 \\
4 & 4 \times 10^0 \\
2 & 2 \times 10^{-1} \\
5 & 5 \times 10^{-2}
\end{array}
$$

De machten van het grondtal waarmee men de cijfers moet vermenigvuldigen, vormen van links naar rechts een aflopende rekenkundige rij:

... 4 3 2 1 0 – 1 – 2 – 3 – 4 – 5 ...

Als voorbeeld de terugrekening van het eerder gebruikte getal 14, binair 0.111 met als exponent 4.

$$
\begin{array}{l}
1 \times 2^{-1} = 0.5 \\
1 \times 2^{-2} = 0.25 \\
1 \times 2^{-3} = 0.125 \\
\qquad\qquad\quad + \\
0.875 \times 2^4 = 14
\end{array}
$$

De floating point-notatie geeft van de mantisse alleen het deel na de punt weer. Voor een achtbits-mantisse betekent dat:

8	1000 0000	exp 3
14	1110 0000	exp 4
16	1000 0000	exp 4
32	1000 0000	exp 5
59	1110 1100	exp 6
512	1000 0000	exp 10
1024	1000 0000	exp 11

Voor de eenvoud is tot dusver met gehele getallen gewerkt. Het is vrij lastig om van een gebroken binair getal de juiste waarde te zien. Toch geven we een paar voorbeelden:

12.5	1100.1	1100 1000 exp 4
1.75	1.11	1110 0000 exp 1
3.625	11.101	1110 1000 exp 2

De werkelijke plaats van de binaire punt hangt af van de exponent en "drijft" bij verhoging of verlaging daarvan door het getal. Vandaar de naam floating point (Engels voor drijvende punt). Het volgende voorbeeld laat dit "drijven" en het effect ervan zien.

mantisse	exp.	binair	decimaal	
1101	5	11010	26	
	4	1101	13	
	3	110.1	6.5	
	2	11.01	3.25	
	1	1.101	1.625	
	0	0.1101	0.8125	
	− 1	0.01101	0.40625	
	− 2	0.001101	0.203125	
	− 3	0.0001101	0.1015625	
	− 5	0.00001101		0.05078125

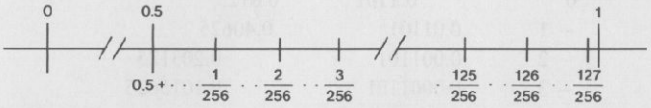
12.2 Bereik en nauwkeurigheid

Het grootste of kleinste in floating point-vorm weer te geven getal hangt samen met het aantal bits van de exponent. Een achtbits-exponent kan bij gebruik van de 2-complementsnotatie waarden hebben van 127 tot en met − 128. De grootste vermenigvuldigingsfactor met een positieve exponent, 2^{127}, ligt decimaal in de orde van grootte van 10^{38}. Bij de meeste achtbits-computers is dit het maximum voor real-variabelen in BASIC. Deze gebruiken dus een formaat van één byte voor de exponent.

Bij integers is het verschil tussen twee opeenvolgende getallen altijd 1. Het kleinste interval wordt bepaald door de waarde van het laagste bit die in dat geval 1 is. In de floating point-notatie hangt de waarde van het laagste bit af van de exponent en de grootte van de mantisse. Stel we gebruiken een achtbits-mantisse en de exponent is 0. De waarde van het MSB is dan 2^{-1} en die van het LSB 2^{-8}, wat neerkomt op $1/(2^8) = 1/256$. Het kleinste interval onder deze omstandigheden is:

```
1000 0000 = 0.50000000
1000 0001 = 0.50390625 _
            0.00390625
```

Getallen tussen de twee opeenvolgende waarden kunnen in een floating point-notatie van dit formaat niet worden weergegeven. De floating point-notatie geeft dus binnen een bepaald aantal bits veel grotere en kleinere getallen dan bij gebruik van integers mogelijk is, maar introduceert tegelijkertijd onnauwkeurigheid omdat het niet alle getallen kan voorstellen.
Het kleinste interval is (waarde exponent) × (waarde laagste bit mantisse). Voor bovenstaand voorbeeld komt dat neer op $2^0 \times 2^{-8} = 1/256 = 0.00390625$.
In afb. 12.1 zijn de mogelijke floating point-getallen voor bovengenoemd formaat op een rechte lijn uitgezet. Het kleinste getal is 0.5, het grootste $0.5 + 127/256 = 0.99609375$. Elk interval is 1/256 groot.
In totaal zijn er 128 getallen, de helft slechts van wat men in een achtbits-formaat zou verwachten. Dat komt omdat het hoogste bit altijd 1 is en er dus maar 7 bits zijn voor veranderingen in het getal.

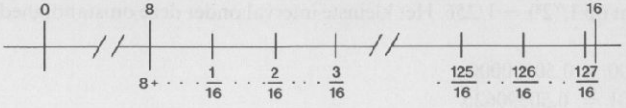


Afb. 12.1 Getallenreeks voor exponent 0

Een grotere exponent maakt het interval tussen twee getallen groter. Is in bovenstaand voorbeeld de exponent niet 0 maar 4, dan vertegenwoordigen de floating point-getallen de volgende waarden:

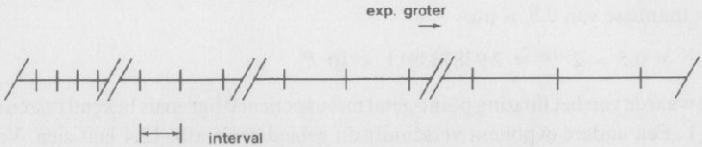
1000 0000 = 8.00000
1000 0001 = 8.06250
—————————————
0.06250

Het kleinste interval is (waarde exponent) × (waarde laagste bit mantisse) = $2^4 \times 2^{-8} = 2^{-4} = 1/16 = 0.0625$. In afb. 12.2 zijn de mogelijke getallen binnen dit formaat op een rechte lijn uitgezet. Het kleinste getal is 8, het grootste $8 + 127/16 = 15.9375$.



Afb. 12.2 Getallenreeks voor exponent 4

Het is duidelijk dat de exponent mede de grootte van het interval bepaalt. Dit verschijnsel deelt het waardebereik van floating point-getallen op in gebieden met verschillende intervallen. Ligt het formaat vast, dan neemt het interval toe naarmate de exponent groter wordt. In afb. 12.3 is dit schematisch weergegeven.



Afb. 12.3 Relatie tussen interval en exponent

De rekenprogramma's in dit boek gebruiken een 32-bits-mantisse. Bij een grotere mantisse neemt de waarde van het laagste bit en daarmee die van het interval af. Voor een 32 bits mantisse en exponent 0 is het interval 2^{-32}. Voor een exponent 0 ligt de waarde van de mantisse tussen 0.5 en 1, maar wordt nooit 1. In het eerste geval is alleen het hoogste bit 1, in het tweede geldt dat voor alle bits. Een achtbits-mantisse heeft dan de waarde:

$$0.5 + \frac{127}{2^8} = \frac{2^7}{2^8} + \left(\frac{2^7}{2^8}\right)^{-1} = \frac{2.2^7}{2^8} - \frac{1}{2^8} = 1 - \frac{1}{2^8}$$

Voor een 32-bits-mantisse vinden we:

$$\frac{2^{31}}{2^{32}} + \left(\frac{2^{31}}{2^{32}}\right)^{-1} = 1 - \frac{1}{2^{32}}$$

Het verschil met 1 is dus gelijk aan het kleinste interval, wat in feite heel vanzelfsprekend is. Bekijken we de zaak nog eens met een achtbits-mantisse, dan is in het verschil met 1 alleen het het laagste bit.

```
0.1111 1111
0.0000 0001
-----------  +
1.0000 0000
```

Het bovenstaande wil zeggen dat een grotere mantisse geen grotere getallen bevat, maar er meer kan onderscheiden.
De programma's gebruiken een achtbits-exponent in de 2-complementsnotatie.

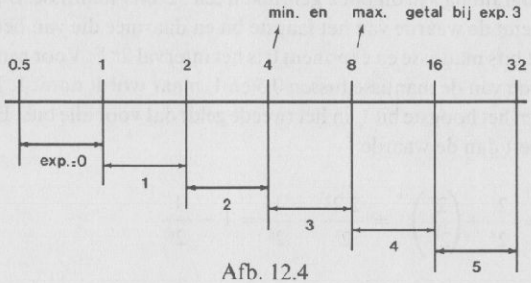
Het grootst weer te geven getal, de mantisse gemakshalve 1 verondersteld, is dan:

$2^{127} = 1.701411805 \times 10^{38}$

De grootste negatieve exponent is -127. De waarde -128 is, zoals verderop zal blijken, gereserveerd voor speciale gevallen. Het kleinst weer te geven getal, met een mantisse van 0.5, is nu:

$2^{-127} \times 0.5 = 2^{-128} = 2.938733911 \times 10^{-39}$

De waarde van het floating point-getal met exponent 0 ligt zoals bekend tussen 0.5 en 1. Een andere exponent verschuift dit gebied zoals afb. 12.4 laat zien. Voor exponent 2 ligt de waarde tussen $2^2 \times 0.5$ en $2^2 \times 1 = 2$ en 4. Voor exponent 5 tussen $2^5 \times 0.5$ en $2^5 \times 1 = 16$ en 32.



Afb. 12.4

Nog niet opgelost is de kwestie van een eventuele negatieve mantisse. Het hoogste bit van de mantisse is altijd 1. Deze eigenschap kan worden gebruikt om onderscheid te maken tussen positieve en negatieve getallen. Van negatieve getallen is het hoogste bit 1 en van positieve getallen is het hoogste bit 0.

-8 = 1000 0000 exp 3
$+8$ = 0000 0000 exp 3
$+9$ = 0001 0000 exp 3

Deze truc dient alleen om bij het doorgeven van getallen aan routines een teken aan te geven. Een rekenroutine zal allereerst het hoogste bit van $+8$ weer 1 maken. Uit de tekens van beide, aan een rekensubroutine aangeboden getallen zal moeten blijken of het resultaat positief of negatief is. In het eerste geval zal het rekenprogramma bij teruggave het hoogste bit van de uitkomst 0 maken. Het rekenen zelf gebeurt dus met floating point-getallen zonder teken.
Deze manier van omgaan met het teken maakt het onmogelijk het getal 0 weer te geven met alleen nullen. Immers, als het hoogste bit van een getal 0 is, vat het

programma dat op als een positiefteken en maakt het vervolgens 1:
0000 0000 exp 0
wordt opgevat als:
1000 0000 exp 0 = 0.5

Het vastleggen van het getal 0 moet dus met behulp van de exponent gebeuren. De maximale waarden daarvan zijn 127 en −128. De waarde −128 reserveren we voor speciale getallen: Is bij een exponent van −128 de mantisse 0 dan is het getal 0. Is de mantisse FFFF bij een exponent van −128 dan is het getal oneindig.

12.3 Optellen

De floating point-rekenprogramma's verwerken een 32-bitsmantisse. Vanwege de eenvoud en de duidelijkheid is in de nu volgende voorbeelden de mantisse 4 bits groot. Of misschien beter gezegd: 4 bits klein.
Het enige probleem bij optellen, is dat bits van overeenkomstige waarde onder elkaar moeten staan. Hetzelfde geldt voor de cijfers van een decimale optelling.

$$\begin{array}{ll} 2.00 \times 10^3 & 2.00 \times 10^3 \\ 4.00 \times 10^1 + & 0.04 \times 10^3 + \\ \hline & 2.04 \times 10^3 \end{array}$$

Optellen kan pas als de exponenten gelijk zijn. Dat is voor elkaar te krijgen door het getal met de laagste exponent naar rechts te schuiven en de exponent te verhogen tot deze overeenkomen.

$$\begin{array}{lll} 8 & 1000 \text{ exp } 4 & 1000 \text{ exp } 4 \\ 3 & 1100 \text{ exp } 2 & 0011 \text{ exp } 4 \\ & & \hline & & 1011 \text{ exp } 4 \ (=11) \end{array} +$$

Bij optelling kan een carry ontstaan. In dat geval roteert men de carry naar rechts de som in en verhoogt men de exponent.

$$\begin{array}{lll} 12 & 1100 \text{ exp } 4 & 1100 \text{ exp } 4 \\ 6 & 1100 \text{ exp } 3 & 0110 \text{ exp } 4 \\ \hline 18 + & & \hline & & 1\,0010 \text{ exp } 4 = 1001 \text{ exp } 5 \end{array} +$$

Is in dit voorbeeld het tweede getal niet 6 maar 5 dan valt bij de rotatie naar rechts een bit uit de mantisse en is het resultaat 16 in plaats van 17.

```
12        1100 exp 4
 5        0101 exp 4
                    +
          1 0001 exp 4  = 1000 exp 5
```

Deze onnauwkeurigheid ontstaat omdat de mantisse niet groot genoeg is. Is deze 8 bits breed dan is het resultaat, in dit geval, correct.

17 1 0001 0000 exp 4

roteer rechts

1000 1000 exp 5

Het zal duidelijk zijn dat voor getallen waarvan het verschil in exponenten gelijk is aan of groter dan het aantal bits in de mantisse de optelling als som het grootste getal teruggeeft. Het kleinste is immers zover naar rechts geschoven dat de mantisse 0 is.

```
128       1000 exp 8    1000   exp 8
  4       1000 exp 4    0000 1 exp 8
                        ------------ +
                        1000   exp 8
```

12.4 Aftrekken

Bij aftrekken doen zich ingewikkelder problemen voor.

```
10        1010 exp 4    1010 exp 4
 7        1110 exp 3    0111 exp 4
                        ---------- -
                        0011 exp 4
```

Het verschil voldoet niet aan de floating point-notatie aangezien het hoogste bit niet 1 is. Wat nu moet gebeuren, is het getal naar links schuiven en de exponent verlagen tot dat wél het geval is.

0011 exp 4
0110 exp 3
1100 exp 2

Dit proces, dat bij het werken met floating point-getallen vaak voorkomt, heet normaliseren. Om praktische redenen ligt het voor de hand om in een rekenprogramma het kleinste en dus te verschuiven getal steeds in dezelfde registers te

zetten. Dit betekent dat altijd het kleinste getal van het grootste wordt afgetrokken. Had het omgekeerde moeten gebeuren, dan dient de routine het berekende verschil negatief te maken.
Bij aftrekking kan het resultaat 0 zijn. De routine moet dit detecteren en de exponent aanpassen.

12.5 Vermenigvuldigen

Bij floating point-getallen komt deze bewerking neer op het vermenigvuldigen van mantissen en het optellen van exponenten. Net zoals men $(3 \times 10^3) \times (5 \times 10^2)$ uitrekent als $(3 \times 5) \times (10^3 \times 10^2) = 15 \times 10^5$. Het optellen van de exponenten is uiteraard heel eenvoudig en het vermenigvuldigen van de mantissen kan met behulp van het al eerder gebruikte schuif-en-tel-op-mechanisme.
Als voorbeeld berekenen we 5 × 14. De vermenigvuldiger schuift naar rechts de carry in waar op basis van een 1 of 0 de beslissing valt het vermenigvuldigtal al dan niet bij het produkt op te tellen. Het produkt schuift eveneens naar rechts de leeglopende vermenigvuldiger in. Er ontstaat hier dus een achtbits-produkt. In de met 32 bits werkende programma's ontstaat op dezelfde wijze een 64-bits-produkt. Een deel van de laagste 32 bits wordt gebruikt voor de afronding van het resultaat.

14	1110 exp 4	vermenigvuldigtal
5	1010 exp 3	vermenigvuldiger

Bewerking:	vermenig-vuldigtal:	vermenig-vuldiger:	produkt-carry:
Bij aanvang zijn produkt en carry 0	1110	1010	0000 0
Roteer vermenigvuldiger rechts. De carry is 0 dus er volgt geen optelling	1110	0101	0000 0
Roteer produkt rechts	1110	0101	0000 0
Roteer vermenigvuldiger rechts	1110	0010	0000 0
Er is een carry. Tel vermenigvuldigtal op bij produkt	1110	0010	1110 0
Roteer produkt rechts	1110	0010	0111 0
Roteer vermenigvuldiger rechts	1110	0001	0111 0

Bewerking:	vermenig-vuldigtal:	vermenig-vuldiger:	produkt-carry:
Geen carry: geen optelling. Roteer produkt rechts	1110	0001	0011 1
Roteer vermenigvuldiger rechts	1110	1000	0011 1
Tel vermenigvuldigtal op bij produkt	1110	1000	0001 1
roteer produkt rechts	1110	1000	1000 1
roteer de laatste carry in vermenigvuldiger	1110	1100	1000 1

Het resultaat is: produkt (MSNybble) en vermenigvuldiger (LSNybble), dus 1000 1100. De exponent van het resultaat is 7. Het uiteindelijke resultaat is 70. Was bij de laatste sommatie van produkt en vermenigvuldigtal geen carry ontstaan dan was het hoogste bit van het produkt 0 geweest en had het resultaat moeten worden genormaliseerd.
Moet het resultaat – bijv. voor doorgave aan andere routines – weer 4 bits groot zijn dan is het resultaat 64; dat is de waarde van de 4 hoogste bits.

1000 exp. 7 = 64

Op basis van het niet 0 zijn van de laagste 4 bits kan men overgaan tot afronding. Het produkt is dan 1001 exp 7, ofte wel 72.

12.6 Delen

De deling verloopt volgens het welbekende schuif-en-trek-afproces. De deling is leuker om op papier te proberen dan de hiervoor behandelde bewerkingen. Bij de deling kan men namelijk uit twee gehele getallen een gebroken quotiënt zien ontstaan. De bewerking komt neer op een deling van de mantissen en aftrekking van de exponenten. Als voorbeeld nemen we 9/5. Het quotiënt is bij aanvang 0.

9	1001 exp 4	deeltal
5	1010 exp 3	deler

Het deeltal schuift bit voor bit naar links waarna een poging volgt de deler ervan af te trekken. Het deeltal roteert links de carry in. De deler wordt van het deeltal afgetrokken als de carry 1 is of als het deeltal groter is dan de deler; het quotiënt wordt in beide gevallen verhoogd.

Bewerking:	deeltal:	deler:	Quotiënt:	Carry:
Geen aftrekking. Het quotiënt wordt niet verhoogd	1001	1010	0000	0
Schuif het deeltal naar links	0010	1010	0000	1
Carry = 1, deler van deeltal aftrekken	1000	1010	0000	1
Verhoog het quotiënt	1000	1010	0001	0
Schuif het quotiënt	1000	1010	0010	0
Schuif het deeltal naar links	0000	1010	0010	1
Carry = 1, deler van deeltal aftrekken	0110	1010	0010	1
Verhoog het quotiënt	0110	1010	0011	0
Schuif het quotiënt naar links	0110	1010	0110	0
Schuif het deeltal naar links	1100	1010	0110	0
Het deeltal is groter dan de deler: de deler wordt van het deeltal afgetrokken	0010	1010	0110	0
Verhoog het quotiënt	0010	1010	0111	0

Het quotiënt is 0111 exp 1 en genormaliseerd 1100 exp 0. De exponent die met dit algoritme wordt verkregen, moet echter met 1 worden verhoogd. Het waarom daarvan is gemakkelijk in te zien. Deelt men twee gelijke getallen, dan is het resultaat 1000 exp 0 of te wel 0.5 terwijl het 1000 exp 1 zijn. Deze correctie toegepast levert als quotiënt 1110 exp 1 op ofwel:

$$\begin{array}{l} 1 \\ 0.5 \\ 0.25 \\ \hline 1.75 \end{array} +$$

Hetgeen in een zo bescheiden formaat al een aardige benadering is voor het juiste antwoord: 1.8 Indien het hoogste bit van het quotiënt 0 is, verdient het aanbeveling in plaats van te normaliseren eenmaal extra de deellus te doorlopen en dit 33e bit in het quotiënt te roteren om een grotere nauwkeurigheid te krijgen. Een

simpele vorm van afronding is na dit hele proces kijken of het hoogste bit van het deeltal nu 1 is. In dat geval namelijk zou een volgende poging tot aftrekken succes hebben. Het volgende, nu 33e bit van het quotiënt is dan 1. De afronding komt erop neer in dit geval het quotiënt met 1 te verhogen.

12.7 De programma's

De rekenprogramma's hebben de vorm van subroutines. Ze zijn aanroepbaar vanuit de nog te bespreken evaluator.
Bij de integer-berekeningen werd aan de subroutines het startadres van de getallen meegegeven. De floating pointsubroutines verwachten de getallen op de stapel. Bijvoorbeeld:

deeltal laag
deeltal hoog
exponent deeltal
deler laag deler hoog
exponent deler
return-adres (laagste stapeladres)

Het eerste dat de aangeroepen subroutine moet doen, is het return-adres van de stapel halen en opbergen in een zestienbitsregister. Vervolgens de getallen van de stapel halen, de berekening uitvoeren, het resultaat op de stapel zetten en ten slotte het return-adres toevoegen. Situatie bij terugkeer, vlak voor RET instructie:

quotiënt laag
quotiënt hoog
exponent quotiënt
return-adres

Alle bovengenoemde items op de stapel zijn 16 bits groot. De exponent zelf is echter maar 8 bits. De overige 8 vormen een nutteloos maar meegepushed byte. We beginnen met vermenigvuldigen en delen aangezien de programma's hiervoor het simpelst zijn. Zie voor de vermenigvuldiging diagram 12.1.
De vermenigvuldiging van de mantissen verloopt identiek met die in het eerder gegeven voorbeeld. De getallen staan als volgt in de registers:

		hoog	laag	exp.
vermenigvuldiger		AC	BC'	B
vermenigvuldigtal	DE		DE'	C
produkt		HL	HL'	

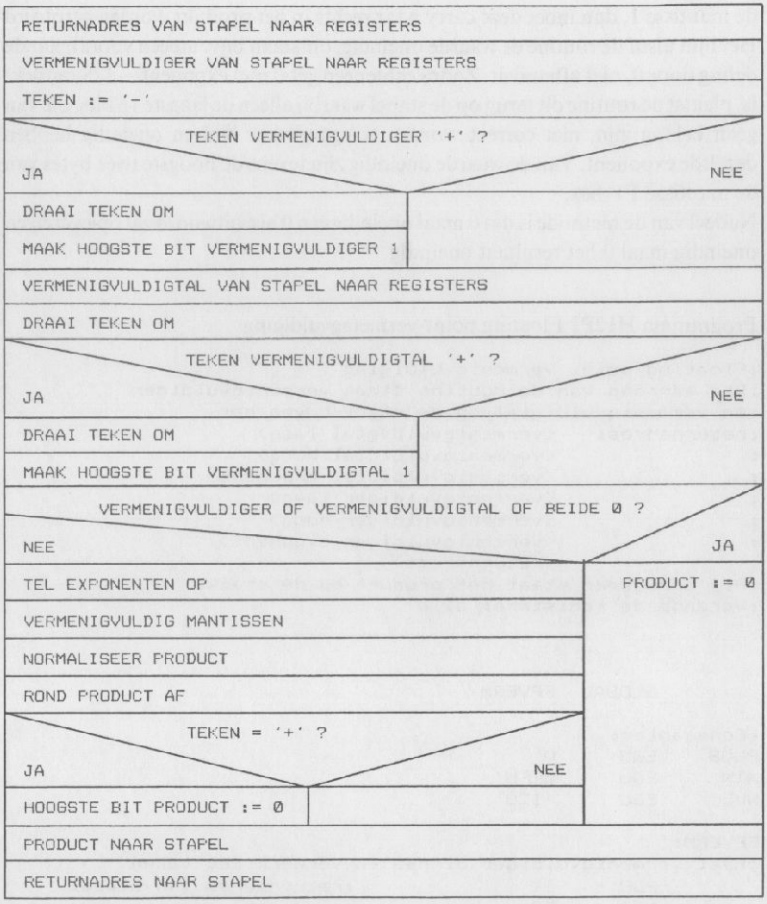


Diagram 12.1 Floating point-vermenigvuldiging

Bedenk dat het produkt naar rechts in de vermenigvuldiger schuift en dat dus in HL HL' de hoogste 32 bits staan. In AC en BC' de laagste 32 bits. Het programma zelf staat in H12P1. Ten aanzien van het afronden nog het volgende. Wordt een 1 opgeteld bij het laagste byte en waren alle bits hiervan 1 dan ontstaat een carry. In zo'n geval moet de routine een 1 optellen bij het op een na laagste byte enz. Geeft ook optelling van 1 bij het hoogte byte nog een carry, en waren dus alle 32 bits van

de mantisse 1, dan moet deze carry naar rechts in het produkt worden geroteerd Het lijkt alsof de routine de waarde oneindig, ontstaan bijv. uit een voorafgaande deling door 0, niet afhandelt. Zodra echter een getal met exponent −128 ontdekt is, plaatst de routine dit terug op de stapel waarbij alleen de laagste 16 bits, die van geen belang zijn, niet correct worden teruggegeven. Nul en oneindig hebben dezelfde exponent. Van de waarde oneindig zijn tevens de hoogste twee bytes van de mantisse FF hex.
Nadeel van de methode is dat 0 maal oneindig een 0 als antwoord zal opleveren en oneindig maal 0 het resultaat oneindig.

Programma H12P1 Floating point-vermenigvuldiging

```
;Floating point vermenigvuldiging
;Bij aanroep van de routine staan vermenigvuldiger
;en vermenigvuldigtal op de stack boven het
;returnadres:    <vermenigvuldigtal laag>
;                <vermenigvuldigtal hoog>
;                <vermenigvuldigtal exponent>
;                <vermenigvuldiger laag>
;                <vermenigvuldiger hoog>
;                <vermenigvuldiger exponent>
;                <returnadres>
;Bij terugkeer staat het product op de stack.
;Veranderde registers: alle



        GLOBAL  FPVERM

;Constanten:
PLUS    EQU     0
MIN     EQU     0FFH
NUL     EQU     -128

FPVERM:
;Haal vermenigvuldiger binnen en verwerk het teken
        POP     IY              ;returnadres van stapel
        POP     BC              ;exponent vermenigvuldiger in B
        POP     HL              ;vermenigvuldiger hoog
        EXX
        POP     BC              ;vermenigvuldiger laag in BC'
        EXX
        LD      A,MIN           ;zet teken op min
        BIT     7,H             ;vermenigvuldiger pos. ?
        JP      NZ,VERMTAL      ;nee, haal verm.tal
        CPL                     ;draai teken om
        SET     7,H             ;tekenbit 1
VERMTAL:
;haal vermenigvuldigtal binnen en verwerk het teken
```

```
        POP     DE              ;exponent verm.tal
        LD      C,D             ; naar C
        POP     DE              ;verm.tal hoog in DE
        EXX
        POP     DE              ;verm.tal laag in DE'
        EXX
        CPL                     ;draai teken om
        BIT     7,D             ;teken positief ?
        JP      NZ,TEKEN
        SET     7,D             ;tekenbit 1
        CPL                     ;draai teken om
TEKEN:  EX      AF,AF           ;bewaar teken in A
;Als vermenigvuldiger of vermenigvuldigtal Ø is
;maak dan het product Ø
        LD      A,B             ;exponent vermenigvuldiger
        CP      NUL             ;vermenigvuldiger Ø ?
        JP      NZ,VTAL_NUL     ;nee, bekijk verm.tal
        JP      FPVEIND
VTAL_NUL:
        LD      A,C             ;exponent verm.tal
        CP      NUL             ;verm.tal Ø ?
        JP      NZ,CTRLEIND     ;nee, einde controle
        LD      B,A             ;exponent in B
        EX      DE,HL           ;product hoog in HL
        JP      FPVEIND
CTRLEIND:
        ADD     A,B             ;exponent product
        PUSH    AF              ;berg exponent op
        LD      A,H             ;verm.er hoog in A
        LD      C,L             ; en C
        AND     A
        SBC     HL,HL           ;product Ø
        EXX
        SBC     HL,HL
        EXX

;Vermenigvuldigingslus
        LD      B,32            ;bitteller
        AND     A               ;carry Ø
FPVLUS: RR      A               ;verm.er naar rechts
        RR      C
        EXX
        RR      B
        RR      C
        EXX
        JP      NC,GEENOPT      ;is er een carry ?
        EXX
        ADD     HL,DE           ;ja, verm.tal + product
        EXX
        ADC     HL,DE
GEENOPT:
```

```
        RR      H               ;product naar rechts
        RR      L               ; en in verm.er
        EXX
        RR      H
        RR      L
        EXX
        DJNZ    FPVLUS
        RR      A               ;laagste bit product

;Zet het lage deel van het product in DE en de
;exponent in BC.
        EXX
        PUSH    HL              ;product hoog
        EXX
        POP     DE
        POP     BC              ;exponent in B
;Normaliseer het product.Roteer daarbij een 33e bit
;uit het 5e byte van het resultaat in A
NORM:   BIT     7,H             ;hoogste bit 1 ?
        JP      NZ,RONDAF       ;ja, afronden
        RL      A               ;5e byte product
        RL      E               ; in resultaat
        RL      D
        RL      L
        RL      H
        DEC     B               ;werk exponent bij
        JP      NORM            ;probeer nogeens

;Rond het resultaat af. Is het 5e byte van het product
;geen nul tel dan 1 op bij het resultaat.
RONDAF: AND     A               ;is 5e byte 0 ?
        JP      Z,FPVTEK        ;ja verwerk teken
        INC     E               ;nee, verhoog product
        JP      NC,FPVTEK       ;ontstond carry ?
        INC     D               ;ja, geef door
        JP      NC,FPVTEK
        INC     L
        JP      NC,FPVTEK
        INC     H
        JP      NC,FPVTEK
        RR      H               ;product naar rechts
        RR      L
        RR      D
        RR      E
        INC     B               ;verhoog exponent

;Verwerk teken in product
FPVTEK:
        EX      AF,AF           ;teken terug in A
        CP      PLUS            ;is teken positief ?
        JP      NZ,FPVEIND
```

```
        RES     7,H             ;ja, tekenbit 0
;Zet resultaat op stapel
FPVEIND:
        PUSH    DE              ;product laag
        PUSH    HL              ;product hoog
        PUSH    BC              ;exponent
        PUSH    IY              ;returnadres
        RET

        END
```

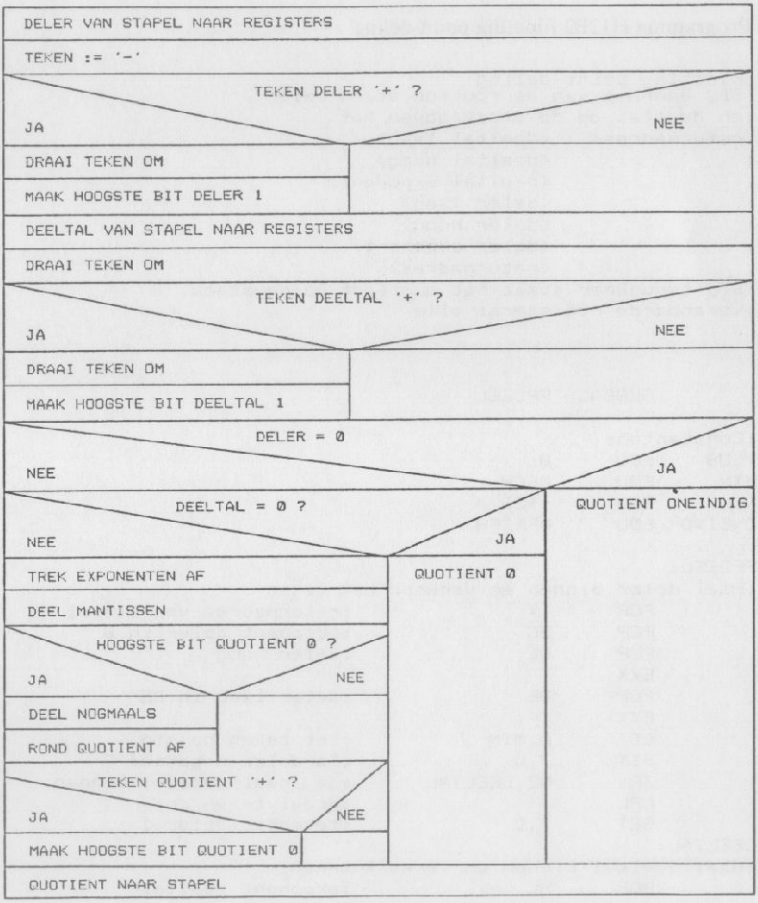


Diagram 12.2 Floating point-deling

Een overzicht van de deling is te vinden in diagram 12.2. De bezetting van de registers is:

	hoog	laag	exp
deler	DE	DE'	B
deeltal	HL	HL'	C
quotiënt	BC'	IY	

De deellus werkt analoog met het eerder gegeven voorbeeld.

Programma H12P2 Floating point-deling

```
;Floating point deling
;Bij aanroep van de routine staan deler
;en deeltal op de stack boven het
;returnadres:    <deeltal laag>
;                <deeltal hoog>
;                <deeltal exponent>
;                <deler laag>
;                <deler hoog>
;                <deler exponent>
;                <returnadres>
;Bij terugkeer staat het quotient op de stack.
;Veranderde registers: alle


        GLOBAL  FPDEEL

;Constanten:
PLUS    EQU     0
MIN     EQU     0FFH
NUL     EQU     -128
ONEIND  EQU     0FFFFH

FPDEEL:
;Haal deler binnen en verwerk het teken
        POP     IX              ;returnadres van stapel
        POP     BC              ;exponent deler in B
        POP     DE              ;deler hoog
        EXX
        POP     DE              ;deler laag in DE'
        EXX
        LD      A,MIN           ;zet teken op min
        BIT     7,D             ;is deler negatief ?
        JP      NZ,DEELTAL      ;ja, haal deeltal binnen
        CPL                     ;draai teken om
        SET     7,D             ;tekenbit deler 1
DEELTAL:
;Haal deeltal binnen en verwerk teken
        POP     HL              ;exponent deeltal
```

```
        LD      C,H             ; naar C
        POP     HL              ;deeltal hoog
        EXX
        POP     HL              ;deeltal laag in HL'
        EXX
        CPL                     ;draai teken om
        BIT     7,H             ;is deeltal positief ?
        JP      NZ,TEKEN
        CPL                     ;ja, draai teken om
        SET     7,H             ;maak tekenbit 1
TEKEN:  EX      AF,AF           ;sla teken op in A'
;Indien deler of deeltal Ø is maak dan het quotient
;oneindig respectievelijk Ø
        LD      A,B             ;exponent deler
        CP      NUL             ;deler Ø ?
        JP      NZ,DTAL_NUL     ;nee, bekijk deeltal
        EXX
        LD      DE,Ø            ;maak quotient oneindig
        LD      BC,ONEIND
        LD      H,NUL
        JP      FPDEIND
DTAL_NUL:
        LD      A,C             ;exponent deeltal
        CP      NUL             ;deeltal Ø ?
        JP      NZ,CTRLEIND     ;nee, einde controle
        EXX
        LD      DE,Ø            ;maak quotient Ø
        LD      BC,Ø
        LD      H,NUL
        JP      FPDEIND
CTRLEIND:
        SUB     B               ;exponent quotient
        LD      C,A             ; naar C
        LD      IY,Ø            ;quotient laag = Ø

;Deellus
        LD      B,32            ;bitteller
        JP      INGANG
FPDLUS: ADD     IY,IY           ;schuif quotient links
        EXX
        RL      C
        RL      B
        ADD     HL,HL           ;schuif deeltal links
        EXX
        ADC     HL,HL
        JP      NC,INGANG       ;was hoogste bit 1
        AND     A               ;ja, aftrekken kan
        EXX
        SBC     HL,DE           ;lage deel HL' en DE'
        EXX
        SBC     HL,DE           ;hoge deel
```

```
        INC     IY              ;verhoog quotient
        JP      LUSEIND
INGANG: EXX
        AND     A
        SBC     HL,DE           ;trek deler af
        EXX
        SBC     HL,DE
        INC     IY              ;verhoog quotient
        JP      NC,LUSEIND      ;was aftrek mogelijk ?
        DEC     IY              ;nee, verlaag quotient
        EXX
        ADD     HL,DE           ;corrigeer aftrek
        EXX
        ADC     HL,DE
LUSEIND:
        DJNZ    FPDLUS

;Is het hoogste bit van het quotient nu 0, deel dan
;nogmaals om de grootst mogelijke nauwkeurigheid
;te krijgen.
        EXX
        BIT     7,B             ;hoogste bit 0 ?
        EXX
        JP      NZ,AFRND        ;nee, rond quotient af
        DEC     C               ;ja, verlaag exponent
        LD      B,1             ;teller voor 1 lus
        JP      FPDLUS

;Rond het resultaat af door het hoogste bit van
;het deeltal erbij op te tellen.
AFRND:  LD      A,C             ;exponent quotient
        INC     A               ;correctie
        RL      H               ;hoogste bit deeltal in carry
        EXX
        LD      H,A             ;exponent quotient in H'
        PUSH    IY
        POP     DE              ;quotient laag in DE'
        LD      A,0             ;tel carry op bij
        ADC     A,E             ; laagste bit quotient
        LD      E,A
        JP      NC,FPDTEK       ;ontstaat weer een carry
        INC     D               ; geef deze dan door
        JP      NC,FPDTEK
        INC     C
        JP      NC,FPDTEK
        INC     B
        JP      NC,FPDTEK       ;blijft uiteindelijk een
        RR      B               ; carry over roteer dan
        RR      C               ; het quotient 1 bit
        RR      D               ; rechts
        RR      E
        INC     H               ;verhoog exponent
```

```
;Verwerk teken in quotient
FPDTEK:
        EX      AF,AF           ;teken in A
        CP      PLUS            ;is teken positief ?
        JP      NZ,FPDEIND
        RES     7,B             ;ja, tekenbit Ø
;Zet resultaat op stapel
FPDEIND:
        PUSH    DE              ;quotient laag
        PUSH    BC              ;quotient hoog
        PUSH    HL              ;exponent quotient
        PUSH    IX              ;returnadres
        EXX
        RET

        END
```

Bij de floating point-optelling ontstaan problemen wanneer de tekens van beide termen ongelijk zijn. In dat geval dient de routine het verschil van beide termen te berekenen. Zie diagram 12.3.
Alvorens op te tellen of af te trekken zorgt de routine ervoor dat TERM2 het getal met de kleinste exponent is. TERM2 wordt naar rechts geschoven tot de exponenten van beide getallen gelijk zijn. Het teken van het resultaat is altijd het teken van de grootste term. Is het verschil tussen de exponenten $> = 32$ dan is de grootste term het resultaat.
De registerbezetting is als volgt:

	hoog	laag	exp.
TERM1	HL	HL′	B′
TERM2	DE	DE′	C′

Een keuze die voor de hand ligt omdat HL voor zowel optellen als aftrekken als zestienbits-accumulator fungeert.

Omdat bij het aftrekken moeilijkheden kunnen ontstaan, is daarvoor een apart diagram en wel 12.4 getekend. Hebben beide termen namelijk dezelfde exponent dan bestaat de mogelijkheid dat TERM2 groter is dan TERM1. Is dat het geval en ontstaat dus bij aftrekking een carry dan moet de routine de beide termen verwisselen en opnieuw aftrekken.
We illustreren de gang van zaken nog even met een paar simpele getallenvoorbeelden. Als eerste $10 + -100$. De tekens zijn ongelijk dus moet aftrekking plaatsvinden. Het getal met de grootste exponent komt in TERM1 waarna de routine TERM1 − TERM2 berekent: $100 - 10 = 90$. Het resultaat krijgt het teken van

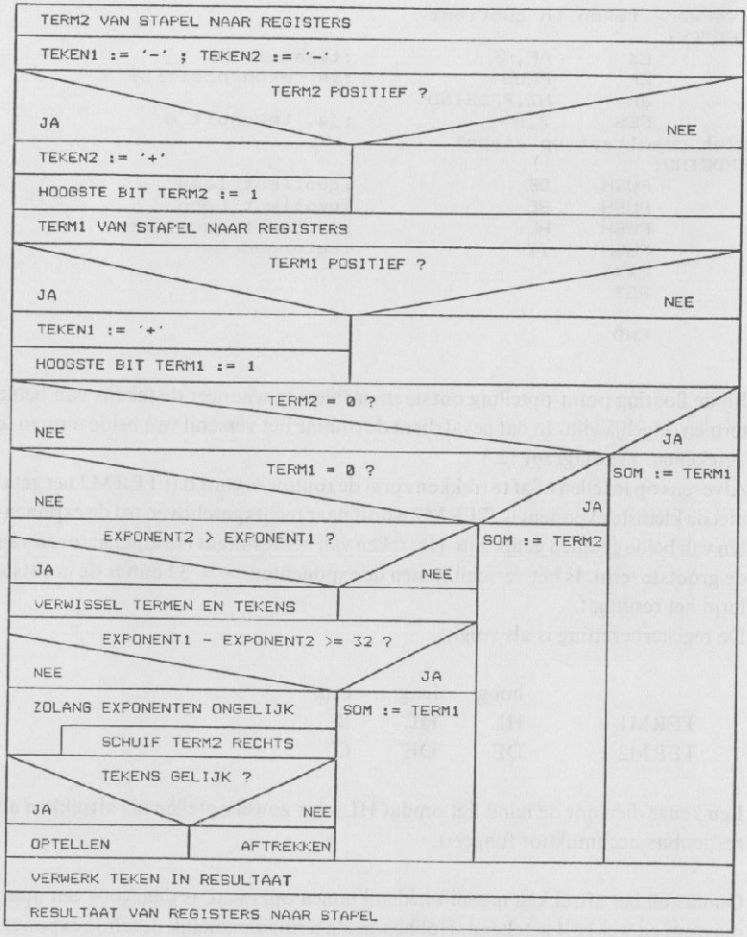


Diagram 12.3 Floating point-optelling

het grootste getal en wordt dus -90. Tweede voorbeeld: $-8 + 10$. De exponenten zijn gelijk dus verwisselt de routine de termen niet. Aangezien de tekens ongelijk zijn, vindt aftrekking plaats en wel TERM1 $-$ TERM2 dus $8 - 10$. Dat

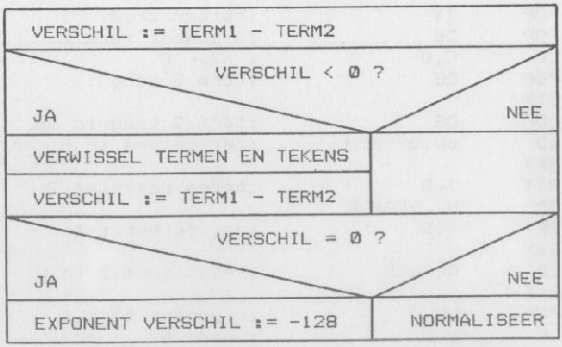


Diagram 12.4 Gedeelte floating point-optelling

levert een carry op. De routine verwisselt de termen alsnog en berekent 10 − 8 en geeft het resultaat het teken van de grootste term. Antwoord in dit geval: +2. Het resultaat van de aftrekking kan 0 zijn. De routine detecteert dit met een OR van alle bytes van de mantisse. Is het resultaat daarvan 0 dan krijgt het resultaat de exponent −128.

Programma H12P3 Floating point-optelling

```
;Floating point optelling
;Bij aanroep van de subroutine staan beide
;termen op de stapel boven het
;returnadres:    <term 1 laag>
;                <term 1 hoog>
;                <term 1 exponent>
;                <term 2 laag>
;                <term 2 hoog>
;                <term 2 exponent>
;                <returnadres>
;Bij terugkeer staat de som op de stapel.
;Veranderde registers: alle


        GLOBAL  FPPLUS

;Constanten:
PLUS    EQU     0
MIN     EQU     0FFH
NUL     EQU     -128

FPPLUS:
;Haal getallen van de stapel en verwerk de tekens
```

```
        POP     IX              ;returnadres
        POP     DE              ;exponent term 2
        LD      C,D             ; naar C
        POP     DE              ;term 2 hoog
        EXX
        POP     DE              ;term 2 laag in DE'
        LD      BC,0FFFFH       ;zet tekens in B' en C' op min
        EXX
        BIT     7,D             ;teken positief ?
        JP      NZ,VLGNDE
        SET     7,D             ;ja, tekenbit 1
        EXX
        LD      C,PLUS          ;teken term 2 in C'
        EXX
VLGNDE: POP     HL              ;exponent term 1
        LD      B,H             ; naar B
        POP     HL              ;term 1 hoog
        EXX
        POP     HL              ;term 1 laag in HL'
        EXX
        BIT     7,H             ;teken positief ?
        JP      NZ,NULCTRL
        SET     7,H             ;ja, tekenbit 1
        EXX
        LD      B,PLUS          ;teken term 1 in B'
        EXX

;Controleer of een van de termen 0 is, In dat geval
;is de andere term de som.
NULCTRL:
        LD      A,NUL
        CP      C               ;term 2 0 ?
        JP      Z,FPPEIND       ;ja, som in HL HL'
        CP      B               ;term 1 0 ?
        JP      NZ,EXP
        LD      B,C             ;ja verwissel 1 en 2
        EX      DE,HL
        EXX
        EX      DE,HL
        LD      B,C
        EXX
        JP      FPPEIND

;Zorg ervoor dat de grootste term in HL HL' staat.
;Schuif de kleinste term links tot de exponenten
;gelijk zijn.
EXP:    LD      A,B             ;exponent term 1
        CP      C               ;vergelijk exponent 2
        JP      Z,BEW           ;exponenten gelijk
        JP      P,SCHUIF        ;exponent 1 groter
        LD      B,C             ;nee, verwissel termen,
```

```
        LD      C,A             ; exponenten en tekens
        EX      DE,HL
        EXX
        EX      DE,HL
        LD      A,C
        LD      C,B
        LD      B,A
        EXX
        LD      A,B

SCHUIF:
        SUB     C               ;verschil exponenten
        CP      32              ;groter dan of gelijk aan 32 ?
        JP      P,FPPEIND       ;som is term 1
        LD      C,B             ;exponent som
        LD      B,A             ;verschil exponenten
SCHLUS: SRL     D               ;schuif term 2 rechts
        RR      E               ; tot exponenten
        EXX                     ; gelijk zijn
        RR      D
        RR      E
        EXX
        DJNZ    SCHLUS
        LD      B,C             ;exponent terug in B

;Bepaal de bewerking die de absolute waarden van de
;termen moeten ondergaan.
BEW:    EXX
        LD      A,B             ;teken term 1 in B'
        XOR     C               ;gelijk aan teken term 2 ?
        JP      NZ,AFTREK       ;nee, dan aftrekken
;Absolute waarden van de termen optellen als
;de tekens gelijk zijn.
        ADD     HL,DE           ;som lagere delen in HL' DE'
        EXX
        ADC     HL,DE           ;som hogere delen
        JP      NC,FPPEIND      ;is er een carry ?
        RR      H               ;roteer in som
        RR      L
        EXX
        RR      H
        RR      L
        EXX
        INC     B               ;verhoog exponent
        JP      FPPEIND
;Absolute waarden van de termen aftrekken als de
;tekens ongelijk zijn.
AFTREK: SBC     HL,DE           ;verschil lagere delen
        EXX
        SBC     HL,DE           ;verschil hogere delen
        JP      NC,SOMNUL?      ;carry door aftrek ?
        EXX                     ;ja, maak aftrek
        ADD     HL,DE           ; ongedaan, verwissel
```

```
        EX      DE,HL           ; getallen en tekens
        LD      B,C
        EXX
        ADC     HL,DE
        AND     A               ;en trek opnieuw af
        EXX
        SBC     HL,DE           ;lagere delen
        EXX
        SBC     HL,DE           ;hogere delen
;Kijk of het nu ontstane resultaat nul is.
SOMNUL?:
        EXX
        LD      A,L             ;logische OF van alle
        OR      H               ; bytes mantisse
        EXX
        OR      L
        OR      H
        CP      Ø               ;resultaat Ø ?
        JP      NZ,NORM         ;nee, normaliseer som
        LD      B,NUL           ;ja, som is nul
        EXX
        LD      B,PLUS          ;teken plus
        EXX
        JP      FPPEIND
;Normaliseer het resultaat, d.w.z. schuif het links
;tot het hoogste bit 1 is en pas de exponent aan.
NORM:   BIT     7,H             ;hoogste bit 1 ?
        JP      NZ,FPPEIND      ;ja, klaar
        EXX                     ;nee, schuif links
        SLA     L
        RL      H
        EXX
        RL      L
        RL      H
        DEC     B               ;verlaag exponent
        JP      NORM
;Bepaal het teken van het resultaat en zet som en
;exponent op de stapel.
FPPEIND:
        EXX
        LD      A,B
        CP      PLUS            ;teken plus ?
        JP      NZ,WERKAF       ;nee, dan klaar
        EXX
        RES     7,H             ;ja, maak hoogste bit Ø
        EXX
WERKAF: PUSH    HL              ;lage deel som in HL'
        EXX
        PUSH    HL              ;hoge deel som
        PUSH    BC              ;exponent
        PUSH    IX              ;returnadres
        RET
        END
```

De floating point-aftrekking is praktisch identiek met de optelling. Zie diagram 12.5. Het enige verschil is dat bij gelijke tekens de absolute waarden van de termen van elkaar worden afgetrokken. Bij ongelijke tekens vindt een optelling daarvan plaats.

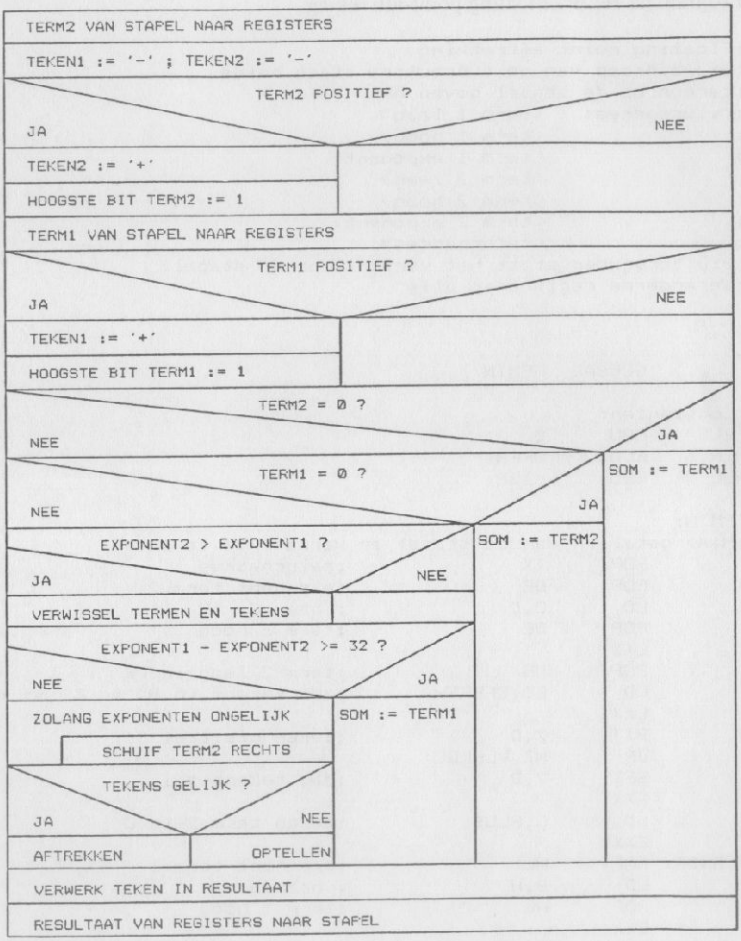


Diagram 12.5 Floating point-aftrekking

In principe is het mogelijk voor optellen en aftrekken dezelfde routine te gebruiken, bijv. het optelprogramma. Moet er in dat geval worden afgetrokken dan dient de aanroepende routine het teken van TERM2 om te draaien. De aftrekking 10 − 3 verandert dan in de optelling 10 + −3.

Programma H12P4 Floating point-aftrekking

```
;Floating point aftrekking
;Bij aanroep van de subroutine staan beide
;termen op de stapel boven het
;returnadres:    <term 1 laag>
;                <term 1 hoog>
;                <term 1 exponent>
;                <term 2 laag>
;                <term 2 hoog>
;                <term 2 exponent>
;                <returnadres>
;Bij terugkeer staat het verschil op de stapel.
;Veranderde registers: alle


        GLOBAL  FPMIN

;Constanten:
PLUS    EQU     0
MIN     EQU     0FFH
NUL     EQU     -128

FPMIN:
;Haal getallen van de stapel en verwerk de tekens
        POP     IX              ;returnadres
        POP     DE              ;exponent term 2
        LD      C,D             ; naar C
        POP     DE              ;term 2 hoog
        EXX
        POP     DE              ;term 2 laag in DE'
        LD      BC,0FFFFH       ;zet tekens in B' en C' op min
        EXX
        BIT     7,D             ;teken positief ?
        JP      NZ,VLGNDE
        SET     7,D             ;ja, tekenbit 1
        EXX
        LD      C,PLUS          ;teken term 2 in C'
        EXX
VLGNDE: POP     HL              ;exponent term 1
        LD      B,H             ; naar B
        POP     HL              ;term 1 hoog
        EXX
        POP     HL              ;term 1 laag in HL'
        EXX
```

```
        BIT     7,H             ;teken positief ?
        JP      NZ,NULCTRL
        SET     7,H             ;ja, tekenbit 1
        EXX
        LD      B,PLUS          ;teken term 1 in B'
        EXX

;Controleer of een van de termen Ø is, In dat geval
;is de andere term het verschil.
NULCTRL:
        LD      A,NUL
        CP      C               ;term 2 Ø ?
        JP      Z,FPPEIND       ;ja, verschil in HL HL'
        CP      B               ;term 1 Ø ?
        JP      NZ,EXP
        LD      B,C             ;ja verwissel 1 en 2
        EX      DE,HL
        EXX
        EX      DE,HL
        LD      A,C             ;inverteer teken
        CPL
        LD      B,A
        EXX
        JP      FPPEIND

;Zorg ervoor dat de grootste term in HL HL' staat.
;Schuif de kleinste term links tot de exponenten
;gelijk zijn.
EXP:    LD      A,B             ;exponent term 1
        CP      C               ;vergelijk exponent 2
        JP      Z,BEW           ;exponenten gelijk
        JP      P,SCHUIF        ;exponent 1 groter
        LD      B,C             ;nee, verwissel termen,
        LD      C,A             ; exponenten en tekens
        EX      DE,HL
        EXX
        EX      DE,HL
        LD      A,C
        LD      C,B
        LD      B,A
        EXX
        LD      A,B

SCHUIF:
        SUB     C               ;verschil exponenten
        CP      32              ;groter dan of gelijk aan 32 ?
        JP      P,FPPEIND       ;verschil is term 1
        LD      C,B             ;exponent verschil
        LD      B,A             ;verschil exponenten
SCHLUS: SRL     D               ;schuif term 2 rechts
        RR      E               ; tot exponenten
        EXX                     ; gelijk zijn
```

```
        RR      D
        RR      E
        EXX
        DJNZ    SCHLUS
        LD      B,C             ;exponent terug in B

;Bepaal de bewerking die de absolute waarden van de
;termen moeten ondergaan.
BEW:    EXX
        LD      A,B             ;teken term 1 in B'
        XOR     C               ;gelijk aan teken term 2 ?
        JP      Z,AFTREK        ;ja, dan aftrekken
;Absolute waarden van de termen optellen als
;de tekens ongelijk zijn.
        ADD     HL,DE           ;som lagere delen in HL' DE'
        EXX
        ADC     HL,DE           ;som hogere delen
        JP      NC,FPPEIND      ;is er een carry ?
        RR      H               ;roteer in som
        RR      L
        EXX
        RR      H
        RR      L
        EXX
        INC     B               ;verhoog exponent
        JP      FPPEIND
;Absolute waarden van de termen aftrekken als de
;tekens gelijk zijn.
AFTREK: SBC     HL,DE           ;verschil lagere delen
        EXX
        SBC     HL,DE           ;verschil hogere delen
        JP      NC,VERSNUL?     ;carry door aftrek ?
        EXX                     ;ja, maak aftrek
        ADD     HL,DE           ; ongedaan, verwissel
        EX      DE,HL           ; getallen
        LD      A,B             ;inverteer teken
        CPL
        LD      B,A
        EXX
        ADC     HL,DE
        AND     A               ;en trek opnieuw af
        EXX
        SBC     HL,DE           ;lagere delen
        EXX
        SBC     HL,DE           ;hogere delen
;Kijk of het nu ontstane resultaat nul is.
VERSNUL?:
        EXX
        LD      A,L             ;logische OF van alle
        OR      H               ; bytes mantisse
        EXX
```

```
        OR      L
        OR      H
        CP      0               ;resultaat 0 ?
        JP      NZ,NORM         ;nee, normaliseer verschil
        LD      B,NUL           ;ja, verschil is nul
        EXX
        LD      B,PLUS          ;teken plus
        EXX
        JP      FPPEIND
;Normaliseer het resultaat, d.w.z. schuif het links
;tot het hoogste bit 1 is en pas de exponent aan.
NORM:   BIT     7,H             ;hoogste bit 1 ?
        JP      NZ,FPPEIND      ;ja, klaar
        EXX                     ;nee, schuif links
        SLA     L
        RL      H
        EXX
        RL      L
        RL      H
        DEC     B               ;verlaag exponent
        JP      NORM
;Bepaal het teken van het resultaat en zet verschil en
;exponent op de stapel.
FPPEIND:
        EXX
        LD      A,B
        CP      PLUS            ;teken plus ?
        JP      NZ,WERKAF       ;nee, dan klaar
        EXX
        RES     7,H             ;ja, maak hoogste bit 0
        EXX
WERKAF: PUSH    HL              ;lage deel verschil in HL'
        EXX
PUSH    HL              ;hoge deel verschil
PUSH    BC              ;exponent
PUSH    IX              ;returnadres
RET
END
```

13 Floating point-conversies

13.1 ASCII–naar–floating–point-conversie

In principe is de ASCII–naar–floating–point-conversie gelijk aan de conversie gebruikt bij de 32-bits-integers. Het getal zelf staat in een string. Is een binnengehaald stringteken een cijfer dan komt de omzetting neer op het vermenigvuldigen met 10 van het tot dusver binnengehaalde getal en het daarbij optellen van het nieuwe cijfer. De op deze manier gevormde mantisse is 32 bits groot.
In het eenvoudigste geval is het om te zetten getal een integer, bijv. 58. Bij een achtbits-mantisse levert de conversie dan als resultaat:

58 0011 1010

Om een floating point-notatie te krijgen, moeten de laagste 6 bits naar rechts tot achter de binaire punt schuiven.

58 .1110 1000 bin. exp. 6

Dat komt praktisch op hetzelfde neer als verschuiving naar links tot het MSB gelijk is aan één. De binaire exponent is nu dus de grootte van de mantisse minus het aantal verschuivingen: 8-2=6.
Behalve een integer kan het getal ook gebroken zijn, dat wil zeggen er staat een decimale punt in, bijv. 1.06. De omzetting negeert de punt maar houdt wel het aantal cijfers na de punt bij. Het binnengehaalde getal is dus eigenlijk 106. Behalve de al eerder berekende binaire exponent is er nu ook een decimale exponent die overeen komt met het aantal cijfers na de decimale punt.

1.06 0110 1010 dec. exp. 2

1.06 .1101 0100 bin. exp. 7 dec. exp. 2

Het floating point-getal, mantisse en binaire exponent, is dus een factor 10^2 ofwel 100 te groot. Het juiste floating point getal volgt na tweemaal delen door 10.

Een eenvoudig voorbeeld hiervoor is het getal 12.5. Conversie levert in eerste instantie 125 op.

125 0111 1101 dec. exp. 1

125 .1111 1010 bin. exp. 7 dec.exp. 1

Eénmaal delen door 10 om de decimale exponent weg te werken:

10 =	.1010 0000	bin. exp. 3

Vervolgens geeft het eerder behandelde schuif-en-trek-af-principe als resultaat:

	.1100 1000	bin. exp. 4

Dit is de floating point-notatie van het volgende gebroken binaire getal:

1100.1 = 12.5

Een derde vorm behelst getallen in wetenschappelijke notatie. Enkele voorbeelden daarvan:

0.003	3×10^{-3}	3E−3
125	1.25×10^{2}	1.25E2
13.74	1.374×10^{1}	1.374E1
0.028	2.8×10^{-2}	2.8E−2

Het getal na de E (Exponent) is de exponent voor het grondtal 10. Conversie van zulke getallen vereist de mogelijkheid een E in de invoer te detecteren. De verdere verwerking is dan vrij simpel. Het getal na de E moet worden afgetrokken van het aantal cijfers na de decimale punt.

getal	312.5	
notatie	3.125E2	
binnengehaald	3125 dec. exp. 3	E-factor 2

Op grond van de decimale exponent, het aantal cijfers na de punt, moet het geconverteerde getal driemaal door 10 worden gedeeld. Vervolgens moet het programma de E-factor verwerken door tweemaal te vermenigvuldigen met 10. Welk proces neerkomt op het eenmaal delen door 10.

getal	0.0215	
notatie	2.15E−2	
binnengehaald	215 dec. exp. 2	E-factor −2

Totaal aantal delingen door 10: dec.exp − E-factor = 4.

getal	6800	
notatie	6.8E3	
binnengehaald	68 dec. exp. 1	E-factor 3

Totaal aantal delingen 1 − 3 = −2. Hetgeen, zeer terecht, neerkomt op het tweemaal vermenigvuldigen met 10 aangezien 68 × 100 = 6800.
Diagram 13.1 geeft een overzicht van de ASCII–naar–floating–point-conversie. Een aantal zaken daarin vraagt om wat extra toelichting. Allereerst het tellen van

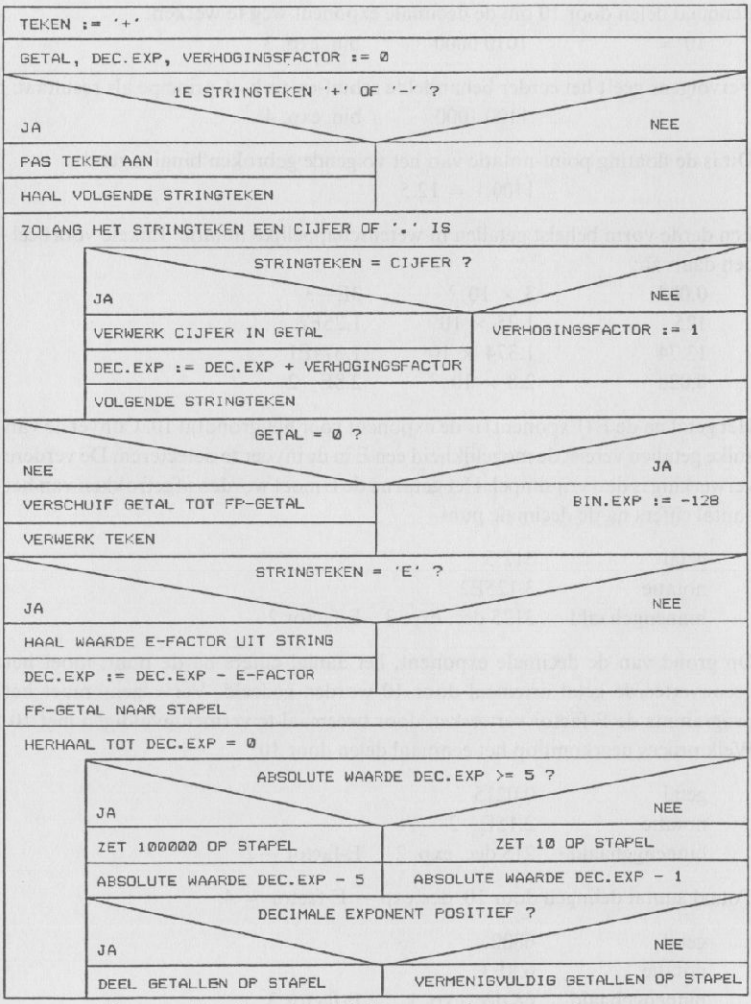


Diagram 13.1 Overzicht ASCII–naar–floating–point-conversie

de cijfers na de decimale punt. Het totaal komt in dec.exp, de decimale exponent. Voor elk cijfer in het getal wordt bij dec.exp de verhogingsfactor opgeteld. Deze is

bij aanvang 0. Na detectie van een punt in de invoerstring maakt het programma de verhogingsfactor 1.
Het delen door of vermenigvuldigen met 10 kan een tijdrovende zaak zijn als dec.exp groot is. Zolang de absolute waarde hiervan 5 of meer is vindt daarom deling door of vermenigvuldiging met 100000 plaats.

13.2 Programma voor ASCII–naar–floating–point-conversie

Het programma heeft de vorm van een subroutine. Bij aanroep verwacht het op de stapel, boven het return-adres naar de aanroepende routine, het startadres van de string:

startadres string
return-adres

Na terugkeer staat op de stapel het floating point-getal en het adres van het eerste, niet meer tot het getal behorende stringteken:

lage deel getal
hoge deel getal
exponent
stringadres

Het programma vormt de mantisse in HL', hoge deel, en IY, lage deel. Elk cijfer minus de ASCII-waarde voor 0 komt in register C' waarna optelling van BC' en IY plaatsvindt. B' is bij aanvang van het programma 0 gemaakt.
De eventuele in de string meegegeven decimale exponent, het getal na de E dus, is een integer-getal dat het programma converteert naar een achtbits-formaat.
De voor het wegwerken van decimale exponenten noodzakelijke delingen of vermenigvuldigingen voert het programma uit door aanroep van de eerder ontwikkelde rekensubroutines.

Programma H13P1

```
;ASCII naar floating-point conversie.
;Aanroep met het startadres van de string op
;de stack.
;Bij terugkeer staat het adres van het eerste,
;niet meer tot het getal behorende teken in de
;string op de stack. Daarboven het floating-point
;getal: <getal laag>
```

```
;       <getal hoog>
;       <exponent>
;       <stringadres>
;       <returnadres>
;Veranderde registers: alle


        GLOBAL  ASCFP
        EXTERNAL        FPVERM,FPDEEL

ASCFP:
;Constanten:
PLUS    EQU     0
MIN     EQU     0FFH
NUL     EQU     -128
;Initialisatie
        POP     IX              ;returnadres
        POP     HL              ;stringadres
        LD      C,PLUS          ;zet teken op plus
        XOR     A               ;A=0
        LD      D,A             ;verhoging decimale exp.
        LD      B,A             ;decimale exp.=0
        EXX
        SBC     HL,HL           ;resultaat hoog in HL' 0
        LD      IY,0            ;resultaat laag in IY 0
        LD      B,A             ;nieuw getal hoog in B' 0
        EXX
;Kijk of het eerste teken een + of - is.
        LD      A,(HL)          ;haal teken
        CP      '+'             ;is het + ?
        JP      NZ,TEKMIN       ;nee, dan een min ?
        INC     HL              ;verhoog stringadres
        JP      CONV            ;start conversie
TEKMIN: CP      '-'             ;is het - ?
        JP      NZ,CONV         ;nee, start conversie
        LD      C,MIN           ;zet teken op -
        INC     HL              ;volgende teken
;Conversie. Kijk eerst of het teken een getal is.
CONV:   LD      A,(HL)          ;haal teken
        CP      '0'             ;kleiner dan 0 ?
        JP      M,GEENGET       ;ja, naar geen getal
        CP      '9'+1           ;groter dan 9 ?
        JP      P,GEENGET       ;ja, naar geen getal
        SUB     '0'             ;verminder met ASCII basis
;Het teken is een getal. Vermenigvuldig het resultaat
;met 10 en tel het nieuwe getal daarbij op.
        EXX
        LD      C,A             ;nieuwe getal in BC'
        ADD     IY,IY
        ADC     HL,HL           ;resultaat * 2
        PUSH    HL
        PUSH    IY              ;berg op
        ADD     IY,IY
        ADC     HL,HL           ;resultaat * 4
```

```
        ADD     IY,IY
        ADC     HL,HL           ;resultaat * 8
        POP     DE
        ADD     IY,DE           ;plus resultaat * 2
        POP     DE
        ADC     HL,DE           ;resultaat * 10
        ADD     IY,BC           ;plus nieuwe getal
        LD      C,B             ;BC' 0
        ADC     HL,BC           ;eventuele carry erbij
        EXX
        LD      A,B             ;werk decimale exp. bij
        ADD     A,D             ;dwz het aantal getallen
        LD      B,A             ;na dec. punt
        INC     HL              ;volgende teken
        JP      CONV

GEENGET:
;Is het teken geen getal, kijk of het de decimale punt is.
;Is het een punt dan moeten van nu af de getallen
;geteld worden. Dat gebeurt door de factor in D' van 0
;op 1 te zetten.
        CP      '.'             ;is het een punt ?
        JP      NZ,KIJK_E       ;nee, dan misschien E
        INC     D               ;factor decimale exp. 1
        INC     HL              ;volgende teken
        JP      CONV            ;verder met conversie
KIJK_E:
;Normaliseer eerst het getal en pas het teken aan.
        EX      AF,AF           ;berg stringteken op
        LD      A,C             ;teken getal
        EXX
        LD      C,A             ; naar C'
;Is het getal 0 ? Dan niet normaliseren.
        PUSH    IY
        POP     DE
        LD      A,D             ;getal is 0 als een OR
        OR      E               ; van alle bits 0 is
        OR      H
        OR      L
        LD      D,NUL           ;exponent voor 0
        JP      Z,NORMEIND      ;is getal 0
        XOR     A               ;nee, maak D 0
        LD      D,A             ;in D' komt bin.exp.
NORM:   BIT     7,H             ;hoogste bit 1 ?
        JP      NZ,ZETTEK
        ADD     IY,IY           ;nee, schuif getal links
        ADC     HL,HL
        INC     D               ;verhoog bin.exp.
        JP      NORM
ZETTEK:
;Reken de binaire exponent uit aan de hand van het
;aantal verschuivingen. Verwerk dan het teken.
        LD      A,32            ;32 - aantal
        SUB     D               ; verschuivingen geeft
        LD      D,A             ; bin.exp in D'
```

```
        LD      A,PLUS
        CP      C               ;teken plus ?
        JP      NZ,NORMEIND
        RES     7,H             ;ja, maak 1e bit Ø
;Het getal staat nu in DE' HL' IY
NORMEIND:
        EXX
        EX      AF,AF           ;stringteken terug in A
        CP      'E'             ;is het E ?
        JP      NZ,CONVEIND     ;nee, einde conversie
;Haal de waarde van de decimale exponent binnen
        LD      E,PLUS          ;zet teken op plus
        LD      D,Ø             ;resultaat = Ø
        INC     HL              ;volgende teken
        LD      A,(HL)
        CP      '+'             ;is het +?
        JP      NZ,EMIN         ;nee, dan min ?
        INC     HL              ;volgende teken
        JP      ECONV           ;haal getal binnen
EMIN:   CP      '-'             ;is het -
        JP      NZ,ECONV        ;nee, haal getal
        LD      E,MIN           ;ja,teken min
        INC     HL              ;volgende teken
;Conversie.Kijk of het teken een getal is. Zoja
;vermenigvuldig resultaat met 10 en tel getal erbij op.
ECONV:  LD      A,(HL)          ;haal teken
        CP      'Ø'             ;kleiner dan Ø ?
        JP      M,EEIND         ;ja, einde
        CP      '9'+1           ;groter dan 9 ?
        JP      P,EEIND         ;ja, eind
        SUB     'Ø'             ;min ASCII basis
        SLA     D               ;resultaat * 2
        ADD     A,D             ;plus getal
        SLA     D               ;* 4
        SLA     D               ;* 8
        ADD     A,D             ;resultaat * 10
        LD      D,A             ; + getal in D
        INC     HL              ;volgende teken
        JP      ECONV
EEIND:
;Bereken de totale decimale exponent uit het zojuist
;binnengehaalde getal en het aantal cijfers achter
;de decimale punt van het eerste getal.
        LD      A,E             ;teken exponent
        CP      PLUS            ;+ ?
        JP      NZ,EXPMIN       ;nee, dan exp. min
        LD      A,B             ;aantal cijfers na punt
        SUB     D               ; min dec.exp
        LD      B,A             ; in B
        JP      CONVEIND
EXPMIN: LD      A,D             ;exponent
        ADD     A,B             ;plus aantal na punt
        LD      B,A
CONVEIND:
        LD      (ADRES),IX      ;berg returnadres op
```

```
        LD      (STRADR),HL     ;berg stringadres op
        EXX
        PUSH    IY              ;getal op stapel
        PUSH    HL
        PUSH    DE
        EXX
;Is de decimale exponent positief, deel het getal
;door 10 tot de decimale exponent 0 is. Is de
;exponent negatief dan vermenigvuldigen.
        BIT     7,B             ;dec.exp. positief
        JP      NZ,EXPNEG       ;nee, negatief
        LD      A,PLUS          ;ja, zet teken op plus
        LD      (TEKEXP),A
        LD      A,B             ;dec.exp in A
        JP      RED5
EXPNEG: LD      A,MIN           ;zet teken op min
        LD      (TEKEXP),A
        LD      A,B             ;absolute waarde
        NEG                     ; dec.exp in A
;Is de decimale exponent 5 of groter, deel of
;vermenigvuldig door of met 100000. Daarna door
;of met 10.
RED5:   CP      5               ;dec.exp. 5 of meer ?
        JP      M,RED1
        SUB     5               ;ja, trek 5 af
        LD      (DECEXP),A      ;bewaar dec.exp.
        LD      HL,0            ;100000 op stapel
        PUSH    HL
        LD      HL,4350H
        PUSH    HL
        LD      HL,1100H
        PUSH    HL
        LD      A,(TEKEXP)
        BIT     7,A             ;teken plus ?
        CALL    Z,FPDEEL        ;dan delen
        LD      A,(TEKEXP)
        BIT     7,A             ;teken min ?
        CALL    NZ,FPVERM       ;dan vermenigvuldigen
        LD      A,(DECEXP)      ;dec.exp. in A
        JP      RED5            ;opnieuw
RED1:   DEC     A               ;verlaag dec.exp
        JP      M,KLAAR         ;was 0
        LD      (DECEXP),A      ;bewaar dec.exp.
        LD      HL,0            ;10 op stapel
        PUSH    HL
        LD      HL,2000H
        PUSH    HL
        LD      HL,0400H
        PUSH    HL
        LD      A,(TEKEXP)      ;teken dec.exp.
        BIT     7,A             ;teken plus ?
        CALL    Z,FPDEEL        ;ja, delen
        LD      A,(TEKEXP)
        BIT     7,A             ;teken min ?
        CALL    NZ,FPVERM       ;ja, vermenigvuldigen
```

```
        LD      A,(DECEXP)      ;decimale exponent
        JP      RED1            ;opnieuw
KLAAR:  LD      HL,(STRADR)     ;adres stringteken
        PUSH    HL              ; naar stapel
        LD      HL,(ADRES)      ;returnadres
        PUSH    HL              ; naar stapel
        RET

ADRES:  DEFW    00
STRADR: DEFW    00
TEKEXP: DEFB    00
DECEXP: DEFB    00

        END
```

13.3 Floating-point-naar-ASCII-conversie

De bedoeling van deze conversie is het floating point-getal om te zetten in een string. Om de problemen die zich hierbij voordoen te doorgronden, bekijken we de diverse soorten getallen die het programma moet kunnen verwerken. Voor de eenvoud zijn de mantisses in de voorbeelden acht bits groot.

Allereerst gehele getallen, bijv. 26:

26	.1101 0000	bin. exp. 5

Het getal laat zich geheel voor de binaire punt schuiven:

26	11010.000	bin. exp. 0

Hierdoor ontstaat voor de punt de welbekende binaire representatie van 26. Natuurlijk bevindt zich in werkelijkheid geen punt in het getal. Wat de routine moet doen, is het getal naar links in één of ander register schuiven en bij elke verschuiving de bin.exp verlagen. Is deze eenmaal 0 dan staat het gehele getal in het register.

26	0001 1010

De conversie is dan hetzelfde als voor integers waarbij elke deling door 10 een nieuw cijfer geeft.
Ingewikkelder ligt het voor gebroken getallen. Het is mogelijk om net als in bovenstaand geval het gehele deel naar buiten te schuiven. In de uitvoerstring moet na conversie hiervan een punt worden gezet. Het gebroken deel van het getal blijft dan over.

2.25	.1001 0000	bin. exp. 2
2	0000 0010	

0.25	.0100 0000	bin. exp. 0

Het gebroken deel genormaliseerd is dan:

0.25	.1000 0000	bin. exp. -1

Elke vermenigvuldiging hiervan met 10 creëert een geheel deel gelijk aan een enkel cijfer, aangezien $10 \times 0.25 = 2.5$ enz.

Na de eerste vermenigvuldiging:

2.5	.1010 0000	bin. exp. 2

Het integer-deel hiervan levert, naar buiten geschoven, het getal 2 op. Over blijft 0.5 dat, vermenigvuldigd met 10, het laatste cijfer 5 geeft.

2.5	.1010 0000	bin. exp. 2

gehele deel naar buiten:

2	0000 0010

over blijft:

0.5	.1000 0000	bin. exp. 0

maal 10 geeft 5 als nieuw geheel deel:

.1010 0000	bin. exp. 3

Een eenvoudiger methode is het oorspronkelijke getal, 2.25, eerst tweemaal met 10 te vermenigvuldigen tot 225 en dan dit gehele getal op de welbekende wijze te converteren.
Een getal als 13.254 moet dan echter driemaal met 10 worden vermenigvuldigd en 12.5 slechts éénmaal. Om te bepalen waar de decimale punt komt, moet het programma het aantal vermenigvuldigingen bijhouden.
In bovenstaande voorbeelden is de mantisse 8 bits groot. Het gehele deel van het getal wordt naar een eveneens 8 bits grootte ruimte geschoven. Het waarom daarvan ligt voor de hand. In de 8 bits van de mantisse staat immers, even afgezien van de binaire exponent aan de hand waarvan de verschuiving plaatsvindt, alle relevante informatie over het getal. Toch doet zich ook hier een probleem voor en wel als de binaire exponent groter is dan het aantal bits in de mantisse. Neem bijv. het getal 1200.

1200	.1001 0110	bin. exp. 11

Het gehele deel is te groot voor 8 bits. De ruimte voor het gehele deel vergroten, is onpraktisch: de binaire exponent kan maximaal 127 zijn. Oplossing is het getal door 10 delen.

120	.1111 0000	bin. exp. 7	dec. exp. 1

De decimale exponent signaleert dat er éénmaal door 10 gedeeld is en moet bij de uitvoer in stringvorm betrokken worden, hetzij als een extra toe te voegen 0 of als een macht van 10.
Er ontstaat eveneens een decimale exponent, maar dan een negatieve, als de binaire exponent negatief is en het getal eerst een of meer malen met 10 vermenigvuldigd moet worden om de binaire exponent 0 of positief te maken.
We lossen bovenstaande problemen op door te kiezen voor uitvoer in een vast formaat en wetenschappelijke notatie. Als formaat is gekozen voor 6 cijfers. Uitvoer heeft dus de volgende vorm:

1375	+1.37500E+03
−12	−1.20000E+01
0.25	+2.50000E−01

Allereerst zorgt het programma ervoor dat het gehele deel van het getal na conversie minstens 6 cijfers heeft. De binaire exponent moet daarvoor 20 of groter zijn, maar natuurlijk niet groter dan 32. Bij een binaire exponent van 20 is het kleinste getal $2^{19} = 524288$ en het grootste $2^{20} - 1 = 1048575$.
Het programma deelt of vermenigvuldigt het getal door of met 10 tot de binaire exponent in het genoemde bereik ligt. Hierbij ontstaat een decimale exponent. Daarna schuift het gehele deel van het getal naar een 32-bits-ruimte en vindt conversie op de welbekende manier plaats. Er ontstaat dan een string van minstens 6 cijfers. Aangezien de punt na het eerste cijfer in de uitvoer komt moet het programma na deling of vermenigvuldiging de ontstane decimale exponent alsnog verhogen met het aantal cijfers min 1. In de uiteindelijke uitvoerstring gebruikt het programma de 6 hoogste cijfers van het conversieresultaat.
Het getal 12.5 bijvoorbeeld heeft in een floating point-notatie een binaire exponent 4:

12.5	bin. exp. 4

Vermenigvuldigen met 10 tot de binaire exponent 20 of meer is, levert op:

1250000	bin. exp. 21	dec. exp. −5

Er zijn namelijk 5 vermenigvuldigingen nodig dus is het getal nu een factor 10^5 te groot. Het getal 1250000 wordt naar buiten geschoven en omgezet in 7 cijfers. Aangezien de decimale punt na het eerste cijfer komt is een correctiefactor van 10^6 nodig:

1.250000	dec. exp.	−5 + 6 = 1

De uiteindelijke uitvoer gebruikt slechts 6 cijfers en wordt dus:

+1.25000E+01

Dit is een correcte presentatie van 12.5.

Diagram 13.2 geeft een overzicht van de conversie. De uitvoer heeft voor de getallen 0 en oneindig niet het bovengenoemde formaat maar respectievelijk 0 en ?.

Om snelheid te winnen bij het creëren van een binaire exponent tussen 19 en 33 wordt het getal niet met 10 vermenigvuldigd maar met 1000. Hoger is niet mogelijk. De eerstvolgende vermenigvuldigingsfactor is namelijk 10000. Daarvan is de binaire exponent 14. Heeft het getal zelf een binaire exponent 19 dan levert de eerste vermenigvuldiging een exponent 33 op waarna een deling weer exponent 19 geeft enz.

Het programma schuift het getal rechts tot alleen het gehele deel over is. Bij elke verschuiving vindt verhoging van de binaire exponent plaats. Is deze eenmaal 32 (het formaat van de mantisse) dan is de binaire fractie verdwenen. Voorbeeld hiervan in een achtbits-formaat:

24.75	1100 0110	bin. exp. 5
24	0001 1000	bin. exp. 8

In de mantisse staat nu de vertrouwde binaire notatie voor het integer-getal 24. Logisch want de binaire exponent geeft het aantal bits voor de binaire punt en de rest, de aanvulling dus tot het formaat van de mantisse, vormt het gebroken deel. Omzetting van het getal naar losse cijfers, de eigenlijke conversie dus, komt neer op het delen door 10 tot het getal 0 is. Zoals bij de binair naar ASCII-conversie van 32-bits-integers al is behandeld komen de afzonderlijke cijfers daarbij in omgekeerde volgorde uit het getal. Om de reeks cijfers om te draaien, is toen gekozen voor een manipulatie via de stapel d.m.v. een recursieve subroutine. Hier gebruikt het programma een hulpstring en vult die van achter naar voren op. Daarna worden de cijfers overgebracht naar de uitvoerstring. Een simpele rechttoe rechtaan methode die hier zo eenvoudig is toe te passen omdat het aantal over te brengen cijfers vastligt, namelijk 6.

13.4 Programma voor floating–point–naar–ASCII-conversie

Ook dit heeft de vorm van een subroutine. Aanroep ervan gebeurt met het floating point-getal op de stapel. Na terugkeer staat op de stapel het startadres van de uitvoerstring.

Voor het delen en vermenigvuldigen met of door 1000 gebruikt het programma de

eerder behandelde rekensubroutines. De integerdeling voor de conversie is als aparte subroutine opgenomen.
De conversie van de één byte grote exponent handelt het programma af door van de absolute waarde hiervan net zo vaak 10 af te trekken tot een getal kleiner dan 10 overblijft. Het aantal aftrekkingen geeft dan de tientallen en hetgeen overblijft de eenheden.

Programma H13P2 Floating–point–naar–ASCII-conversie

```
;Floating point naar ASCII conversie.
;Aanroep met op de stapel het floating point getal.
;               <getal laag>
;               <getal hoog>
;               <exponent>
;Bij terugkeer staat op de stapel het adres
;van de string met ASCCI tekens.


        GLOBAL  FPASC
        EXTERNAL        FPVERM,FPDEEL


;Constanten:
EOL     EQU     -1
NUL     EQU     -128
ONEIND  EQU     0FFH
FORMAT  EQU     6
BUFLEN  EQU     13


;Haal het returnadres van de stapel.
FPASC:  POP     HL
        LD      (ADRES),HL
;Initialisatie
        XOR     A               ;A=0
        LD      (DECEXP),A      ;decimale exponent
;Kijk of het getal 0 of oneindig is.
        POP     AF              ;binaire exponent in A
        CP      NUL             ;nul of oneindig ?
        JP      NZ,GELDIG       ;nee, geldig getal
        LD      HL,STRING       ;startadres string
        POP     AF              ;hoger deel getal
        POP     DE              ;lager deel getal
        CP      ONEIND          ;is getal oneindig ?
        JP      Z,INFIN         ;ja
        LD      (HL),'0'        ;nee, 0 naar string
        JP      KLAAR
INFIN:  LD      (HL),'?'        ;? naar string
        JP      KLAAR
;Het getal is niet 0 of oneindig. Kijk of de exponent
;kleiner is dan 33 en groter dan of gelijk aan 20.
```

```
;Is dat niet het geval deel of vermenigvuldig dan
;het getal door of met 1000.
GELDIG: PUSH    AF              ;hele getal op stapel
        LD      HL,0            ;zet 1000 op stapel
        PUSH    HL
        LD      HL,7A00H
        PUSH    HL
        LD      HL,0A00H
        PUSH    HL
        BIT     7,A             ;bin.exp. negatief ?
        JP      NZ,EXPNEG       ;ja
        CP      33              ;bin.exp. groter dan 32 ?
        JP      C,EXP20         ;nee
        CALL    FPDEEL          ;ja, deel
        LD      A,(DECEXP)      ; en werk dec.exp. bij
        ADD     A,3
        LD      (DECEXP),A
        POP     AF              ;binaire exponent
        JP      GELDIG          ;test opnieuw
EXP20:  CP      20              ;bin.exp 20 of groter
        JP      NC,TEKEN        ;ja, verwerk teken
EXPNEG: CALL    FPVERM          ;nee, vermenigvuldig
        LD      A,(DECEXP)      ; en werk dec.exp. bij
        SUB     3
        LD      (DECEXP),A
        POP     AF              ;binaire exponent
        JP      GELDIG          ;test opnieuw

;Kijk of het getal positief of negatief is. Zet het
;juiste teken in de ASCII string. Is het getal
;positief maak dan het hoogste bit 1.
TEKEN:  POP     HL              ;haal 1000 van stapel
        POP     HL
        POP     HL
        POP     AF              ; en bin.exp.
        LD      B,A             ;bin.exp
        LD      A,'-'           ;zet teken op -
        POP     HL              ;hoge deel getal
        BIT     7,H             ;is getal negatief
        JP      NZ,GETNEG       ;ja
        SET     7,H             ;nee, maak hoogste bit 1
        LD      A,'+'           ;verander teken
GETNEG: LD      (STRING),A      ;zet teken in string

;Schuif het gedeelte na de punt uit het getal
        POP     DE              ;lage deel getal
        LD      A,B             ;bin.exp.
SCHUIF: CP      32              ;is bin.exp. 32 ?
        JP      Z,ZETOM         ;ja
        SRL     H               ;nee, schuif rechts
        RR      L
        RR      D
        RR      E
        INC     A               ;verhoog bin.exp.
        JP      SCHUIF          ; en test opnieuw
```

```
;Zet de nu ontstane integer om in een ASCII string
ZETOM:  EXX
        LD      B,0             ;teller aantal cijfers
        LD      HL,HLPSTR+10    ;adres einde hulpstring
        EXX
VOLGEND:
        CALL    DEEL10          ;schuif 1 cijfer uit getal
        EXX                     ; in A
        ADD     A,'0'           ;plus ASCII basis
        LD      (HL),A          ;cijfer in string
        DEC     HL              ;adres volgende cijfer
        INC     B               ;verhoog aantal cijfers
        EXX
        XOR     A               ;A=0
        OR      H               ;kijk of getal 0 is
        OR      L               ; met logische OR van
        OR      D               ; alle bytes
        OR      E
        JP      NZ,VOLGEND      ;is getal 0 ?
        EXX                     ;ja, zet punt in string
        INC     HL              ;adres laatste cijfer
        LD      A,(HL)          ;cijfer zolang in A
        LD      (HL),'.'        ;decimale punt in string
        DEC     HL              ;volgende cijfer
        LD      (HL),A
        PUSH    HL              ;stringadres en teller
        PUSH    BC              ; aantal cijfers naar
        EXX                     ; andere registerset
        POP     BC
        LD      C,B             ;aantal cijfers in C
        POP     DE              ;adres in hulpstring
        LD      HL,STRING       ;startadres string
        INC     HL              ;sla teken over
        LD      B,FORMAT+1      ;teller
TRANSPORT:
        LD      A,(DE)          ;breng cijfers over van
        LD      (HL),A          ;hulpstring naar string
        INC     HL
        INC     DE
        DJNZ    TRANSPORT
        LD      (HL),'E'        ;E naar string
        INC     HL
;Bereken nu de grootte van de E macht. In C staat
;het aantal cijfers. Dit min 1 plus de decimale
;exponent geeft de E macht.
        DEC     C               ;aantal cijfers - 1
        LD      A,(DECEXP)      ;plus dec.exp.
        ADD     A,C
        LD      B,'+'           ;zet teken op +
        BIT     7,A             ;is E macht positief ?
        JP      Z,EXPLUS        ;ja
        LD      B,'-'           ;nee, verander teken
        NEG                     ;absolute waarde E macht
EXPLUS: LD      (HL),B          ;teken naar string
        INC     HL
```

```
;zet de waarde van de decimale exponent om in
;twee cijfers.
        LD      B,0             ;teller tientallen
OMZEXP: CP      10              ;dec.exp. 10 of groter ?
        JP      C,MIND10        ;nee, minder dan 10
        SUB     10              ;verminder met 10
        INC     B               ;verhoog teller
        JP      OMZEXP          ;opnieuw
MIND10: LD      C,A             ;eenheden in C
        LD      A,B             ;tientallen in A
        ADD     A,'0'           ;plus ASCII basis
        LD      (HL),A          ;naar string
        INC     HL
        LD      A,C             ;eenheden
        ADD     A,'0'           ;plus ASCII basis
        LD      (HL),A          ;naar string
KLAAR:  INC     HL
        LD      (HL),EOL        ;end of line naar string
        LD      HL,STRING       ;start string
        PUSH    HL              ; op stapel
        LD      HL,(ADRES)      ;returnadres
        PUSH    HL              ; op stapel
        RET

DECEXP: DEFB    00
ADRES:  DEFW    00
STRING: DEFS    13
HLPSTR: DEFS    11

;Deel10 subroutine. Deelt het getal in HL en DE door
;10 en geeft de rest terug in A.
DEEL10:
        LD      C,0             ;hierin komt rest
        LD      B,32            ;bitteller
        AND     A               ;carry 0
DEELLUS:
        RL      E               ;nummer 1 bit links
        RL      D
        RL      L
        RL      H
        RL      C               ;schuif bit in C
        LD      A,C             ;rest in A
        SUB     10              ;10 of groter ?
        CCF                     ;inverteer carry
        JP      NC,DEELEIND     ;nee, volgende bit
        LD      C,A
DEELEIND:
        DJNZ    DEELLUS         ;alle 32 bits gedaan ?
        RL      E               ;ja, roteer laatste carry
        RL      D               ; in getal
        RL      L
        RL      H
        LD      A,C             ;rest in A
        RET

        END
```

14 Expressie-evaluatie

14.1 Het principe

De expressie-evaluator, besproken bij de 32-bits-integerrekenprogramma's, kon niets ingewikkelders aan dan een uitdrukking bestaande uit twee getallen en een bewerking. Het nu te bespreken programma evalueert uitdrukkingen met meer getallen, bewerkingen en haakjes. Probleem daarbij is dat de bewerkingen vermenigvuldigen en delen voorrang hebben op optellen en aftrekken. Dat geldt ook voor een eventueel tussen haakjes staand deel van de uitdrukking. Bovendien moet het programma dit laatste eerst uitrekenen.
Eerst het probleem met de diverse bewerkingen. Het programma krijgt de uitdrukking aangeboden in de vorm van een string. Bijvoorbeeld:

```
5 + 12 * 3 - 14 / 2
```

De evaluator leest de string van links naar rechts en roept voor het omzetten van de getallen de ASCII naar floating point-conversiesubroutine aan. De omgezette getallen gaan naar de stapel. Aan elke bewerking wordt een prioriteit toegevoegd. Prioriteit één is voor de bewerkingen optellen en aftrekken. Delen en vermenigvuldigen hebben prioriteit twee. De bewerkingen gaan met hun prioriteit naar een aparte bewerkingsstapel. Gaat er een nieuwe bewerking naar de stapel dan geldt het volgende:
Is de prioriteit van de vorige bewerking gelijk of groter, verwissel dan de twee bewerkingen op de top van de stapel en voer vervolgens die op de top uit met de twee getallen op de top van de getallenstapel.
Tabel 14.1 laat het effect van deze regel zien bij het evalueren van bovenstaande uitdrukking.
In eerste instantie groeien bewerkings- en getallenstapel aan tot toevoeging van de bewerking '−'. De vorige bewerking, '*', heeft een hogere prioriteit. Het programma verwisselt beide bewerkingen zodat '*' op de top van de stapel staat. Daarna voert het deze bewerking uit op de getallen 12 en 3 aan de top van de getallenstapel.
Een dergelijke situatie komt geen tweede maal voor zodat ten slotte alle nog resterende getallen en bewerkingen op de stapels komen. Aan het einde van de

Tabel 14.1 Evaluatie van een uitdrukking zonder haakjes

GETALLEN	OPERATIES	PRIORITEIT
5	+	1
12	*	2
3	-	1
	bewerking 1	
5	+	1
12	-	1
3	*	2
5	+	1
36	-	1
5	+	1
36	-	1
14	/	2
2		
	bewerking 2	
5	+	1
36	-	1
7		
	bewerking 3	
5	+	1
29		
	bewerking 4	
34		

string vindt de evaluator het End Of Line-teken. Het programma voert nu net zolang de bewerking aan top van de bewerkingsstapel uit op de twee getallen aan top van de getallenstapel totdat de bewerkingsstapel leeg is.
Om detectie van een lege bewerkingsstapel mogelijk te maken, wordt hierop voor de evaluatie begint een speciaal beginmerkteken gezet.
In feite komt evaluatie neer op de conversie van algebraïsche naar RPN-notatie. De algebraïsche notatie van een uitdrukking is de ons bekende vorm waarbij de bewerkingen tussen de operanden in staan, bijv. A × B + C × D. Bij de RPN-notatie (RPN is de afkorting van Reverse Polish Notation – Omgekeerde Poolse Notatie) komen de bewerkingen na de operanden: AB × CD × +.

Tabel 14.2 Evaluatie van een expressie met haakjes

GETALLEN	OPERATIES	PRIORITEIT
5 2 9	+ (* -	1 0 2 1
	bewerking 1	
5 18	+ (-	1 0 1
5 18 12	+ (-)	1 0 1
	bewerking 2	
5 6	+	1
5 6 2	+ /	1 2
	bewerking 3	
5 3	+	1
	bewerking 4	
8		

Een voordeel van de RPN-notatie is dat elke uitdrukking zich laat schrijven zonder gebruik te maken van haakjes en bovendien kan worden berekend door domweg van links naar rechts te gaan en elke bewerking op de twee voorafgaande operanden uit te voeren.

Bij de algebraïsche notatie A + B x C moeten we ons bij het uitrekenen realiseren dat vermenigvuldigen een hogere prioriteit heeft dan optellen. De RPN-notatie is ABC × +. Van links naar rechts gaand is de eerste bewerking × en deze heeft betrekking op B en C. Bereken dus eerst B × C (= bijv. D). Over blijft AD + ;deze optelling geeft de juiste uitkomst.

Deze vorm is ideaal voor een computer. De omzetting zoals hierboven kost echter

tijd. Bij een taal als FORTH moet men uitdrukkingen direct in RPN-notatie invoeren wat dan ook een aanzienlijke rekensnelheid oplevert.
Ten slotte nog een voorbeeld met haakjes:

(A + B) / (C − D) AB + CD − /

Het werken met haakjes gaat als volgt. Een openingshaakje komt als merkteken op de bewerkingsstapel. Daar het niet mag voorkomen dat een volgende aan de stapel toe te voegen bewerking een gelijke of lagere prioriteit heeft, is het merkteken van een openingshaakje 0. Het haakje moet immers op een vaste plek in de stapel blijven staan en mag onder geen beding worden verward met een bewerking.
Komt er in de invoerstring een sluithaakje voor dan worden alle op de op de stapel staande bewerkingen uitgevoerd tot het programma een openingshaakje detecteert. Tabel 14.2 laat de gang van zaken zien voor de uitdrukking:

5 + (2 * 9 − 12) / 2

De stapels groeien aan tot het '−' teken naar de bewerkingsstapel gaat. Op dat moment heeft de vorige bewerking een hogere prioriteit en vindt uitvoering daarvan op de bekende wijze plaats.
Is het sluithaakje eenmaal gevonden dan gebeurt er eigenlijk hetzelfde als aan het einde van de vorige uitdrukking. Het programma voert alle bewerkingen uit, niet tot aan het speciale beginmerkteken maar tot aan het openingshaakje.
Vervolgens verloopt de evaluatie op dezelfde manier als in het vorige voorbeeld.

Voor de duidelijkheid is in tabel 14.2 ook het sluithaakje op de bewerkingsstapel gezet. In het programma zelf gebeurt dat niet.

14.2 Het diagram

Om het diagram te kunnen begrijpen, moet eerst duidelijk zijn wat de evaluator in de string kan verwachten. Na het binnenhalen van een operator of aan het begin van de expressie is dat:

een getal
of een openingshaakje

En na het binnenhalen van een getal:

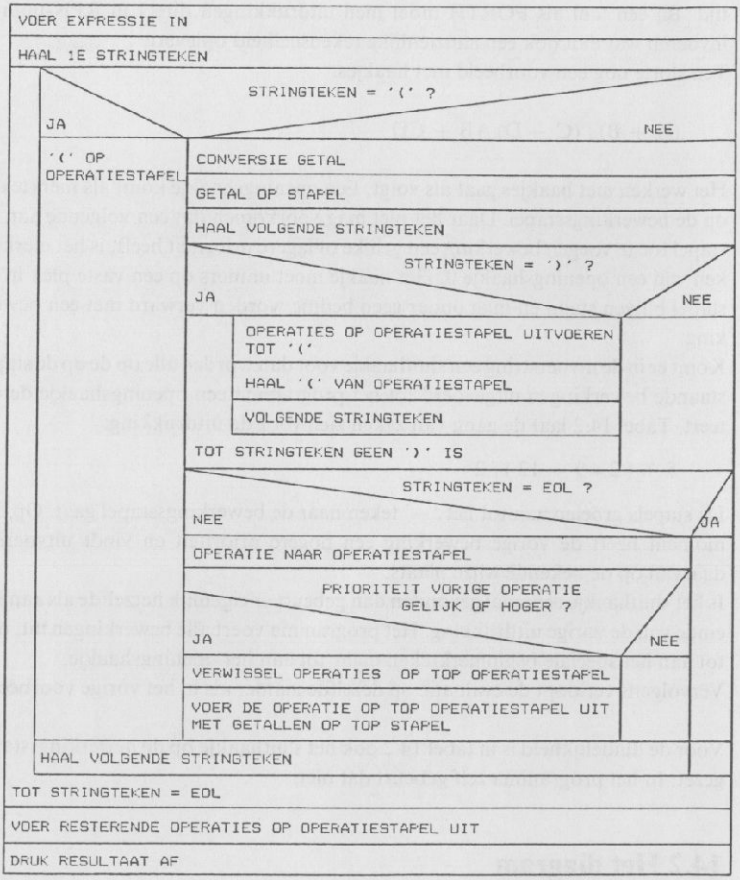


Diagram 14.1 Overzicht van de expressie-evaluatie

een operator
of een sluithaakje
of een End Of Line-teken

Na een openingshaakje is de situatie gelijk aan die na het binnenhalen van een operator, dat wil zeggen er kan een getal of opnieuw een openingshaakje komen.

Na een sluithaakje is de toestand als die na het binnenhalen van een getal: er volgt een operator of opnieuw een sluithaakje of een End Of Line-teken.
Hoe de evaluator op al deze mogelijkheden anticipeert, is te zien in diagram 14.1.

14.3 Het programma

Het programma heeft de vorm van een routine en eindigt net als de vorige evaluator met RST 0, een call naar het CP/M besturingssysteem. Zie programma H14P1.
Zoals in de declaraties is te zien, gebruikt het programma alle tot dusver ontwikkelde floating point-subroutines. Voor de invoer en uitvoer van strings roept het programma de in hoofdstuk 9 besproken subroutines INVOER en UITVOER aan.
Getallen hoeven na conversie niet apart op de stapel gezet te worden. De ASCII naar floating point-conversie geeft na terugkeer namelijk het omgezette getal en het eerste, niet meer tot het getal behorende stringteken, op de stapel.
De aparte stapel voor bewerkingen is een 20 bytes grote buffer. Een pointer, OPWYZER, geeft de laatst bezette plaats in de buffer aan. Elke bewerking neemt op deze stapel twee bytes in beslag. De laatste geeft de prioriteit van de bewerking, de eerste de bewerking zelf. Deze staat niet genoteerd in de ASCII-vorm '+' of '/' maar als getal:

+	0
–	2
*	4
/	6

Dat heeft de volgende reden. Aan het einde van het programma staan in een tabel de startadressen van alle rekensubroutines. Het programma telt het nummer van de bewerking op bij het startadres van de tabel en vindt zo de plaats in de tabel waarop het adres van de gewenste subroutine begint.
Stel de tabel begint op 01F9.

TABEL:	01F9	startadres	optellen
	01FB		aftrekken
	01FD		vermenigvuldigen
	01FF		delen

Is de verlangde bewerking vermenigvuldigen dan geeft optelling van het label TABEL en het nummer van de bewerking: 01F9 + 4 = 01FD. Daar begint het adres van de subroutine vermenigvuldigen. Met 01FD in HL springt het programma indirect naar de rekenroutine met JP (HL).

Het programma heeft een drietal subroutines. SPATIES slaat spaties in de invoerstring over. SWAP verwisselt bewerkingen met hun prioriteiten op de top van de stapel en BEWERK voert de bewerking op top van de stapel uit.
De evaluator roept BEWERK aan. BEWERK zelf maakt, zoals gezegd, een indirecte sprong naar de verlangde rekensubroutine. De RETinstructie daarin laat het programma dus terugkeren naar de instructie volgend op de aanroep van BEWERK.
Het beginmerkteken voor de bewerkingsstapel bestaat uit een EOLteken met prioriteit 0. Een openingshaakje komt als bewerking 0 met prioriteit 0 op de stapel.

Programma H14P1

```
;Expressie evaluator
;Evalueert een rekenkundige uitdrukking en drukt
;het resultaat af op het beeldscherm
;Gebruikte registers: alle


        EXTERNAL        INVOER,UITVOER
        EXTERNAL        ASCFP,FPASC
        EXTERNAL        FPPLUS,FPMIN,FPDEEL,FPVERM

;Constanten:
EOL     EQU     -1

;Initialisatie
        LD      HL,OPSTACK      ;zet begin_merk op
        LD      (HL),EOL        ; operatiestapel
        INC     HL
        LD      (HL),0          ;en bewaar wyzer
        LD      (OPWYZER),HL    ; operatiestapel
;Voer de string met de expressie in en start de
;evaluatie
        CALL    INVOER          ;invoer string
GETAL:  CALL    SPATIES         ;schrap spaties
        CP      '('             ;openingshaakje ?
        JP      NZ,CONV         ;nee, een getal
        EX      DE,HL           ;ja, bewaar stringadres
        LD      HL,(OPWYZER)    ;teken voor haakje op
        INC     HL              ; operatiestapel
        LD      (HL),0          ;operatie 0
        INC     HL
        LD      (HL),0          ;prioriteit 0
        LD      (OPWYZER),HL    ;bewaar wyzer
        EX      DE,HL
        INC     HL              ;volgende stringteken
        JP      GETAL
CONV:   PUSH    HL              ;stringadres op stack
        CALL    ASCFP           ;conversie getal
        POP     HL
```

```
TEST:   CALL    SPATIES
        CP      EOL             ;einde invoer ?
        JP      Z,EINDE
        CP      ')'             ;sluithaakje ?
        JP      NZ,OPERATOR     ;nee, dan operator
;Er staat een sluithaakje in de invoerstroom. Voer
;bewerkingen op de operatiestapel uit tot een
;openingshaakje wordt gevonden
        LD      (STRING),HL     ;berg stringadres op
ZOEKH:  LD      HL,(OPWYZER)    ;wyzer operatiestapel
        LD      A,(HL)          ;laatste operator
        CP      Ø               ; openingshaakje ?
        JP      Z,OPHAAK        ;ja
        CALL    BEWERK          ;nee, voer bewerking uit
        JP      ZOEKH           ; tot openingshaakje
OPHAAK:
;Openingshaakje gevonden
        DEC     HL              ;wyzer operatiestapel
        DEC     HL              ; terug tot voor
        LD      (OPWYZER),HL    ; openingshaakje
        LD      HL,(STRING)     ;adres invoerstring
        INC     HL              ;volgende teken is operator
        JP      TEST            ; of EOL of sluithaakje

OPERATOR:
;In de invoerstroom staat een operator. Bepaal
;prioriteit en positie van de bewerking in de tabel
;en zet deze op de operatiestapel
        LD      (STRING),HL     ;berg stringadres op
        CP      '+'             ;operator plus ?
        JP      NZ,MIN?         ;nee, min ?
        LD      B,Ø             ;ja, operatie Ø
        LD      A,1             ;prioriteit 1
        JP      OPEIND          ;zet beide op stapel
MIN?:   CP      '-'             ;operator min ?
        JP      NZ,MAAL?
        LD      B,2
        LD      A,1
        JP      OPEIND
MAAL?:  CP      '*'
        JP      NZ,DEEL?
        LD      B,4
        LD      A,2
        JP      OPEIND
DEEL?:  LD      B,6
        LD      A,2
;Opeind zet plaats van de bewerking in de tabel en de
;prioriteit op de operatiestapel. Is de prioriteit
;van de vorige bewerking gelijk of groter dan deze
;verwissel dan de twee bewerkingen en hun prioriteiten
;aan de top van de stapel en voer die aan de top uit.

OPEIND: LD      HL,(OPWYZER)    ;wyzer operatiestapel
        LD      C,(HL)          ;prioriteit vorige operatie
        INC     HL
```

```
        LD      (HL),B          ;bewerking nieuwe operator
        INC     HL
        LD      (HL),A          ;prioriteit ervan
        LD      (OPWYZER),HL    ;berg wyzer op
        LD      B,C
        LD      C,A             ;prioriteit laatste operator
        LD      A,B             ;prioriteit vorige operator
        CP      C               ;vorige gelijk of groter ?
        JP      C,GEENBEW       ;nee, geen bewerking
        CALL    SWAP            ;ja, verwissel top stapel
        CALL    BEWERK          ; en voer bewerking uit
GEENBEW:
        LD      HL,(STRING)     ;volgende stringteken
        INC     HL
        JP      GETAL           ;moet een getal of
                                ; openingshaakje zijn
EINDE:
;Er is een EOL in de invoer gevonden. Voer alle nog op
;de operatiestapel staande bewerkingen uit.
        LD      HL,(OPWYZER)    ;wyzer operatiestapel
        LD      A,(HL)          ;prioriteit operatie
        CP      0               ;0=einde operaties ?
        JP      Z,DRUKAF        ;ja, druk resultaat af
        CALL    BEWERK          ;nee, voer bewerking uit
        JP      EINDE           ;nu klaar ?
DRUKAF: CALL    FPASC           ;conversie naar ASCII
        POP     HL              ;adres uitvoerstring
        CALL    UITVOER         ;druk resultaat af
        RST     0

;Subroutines

SPATIES:
;Slaat spaties in de invoerstring over en geeft in HL
;het adres van de 1e niet spatie.
        LD      A,(HL)          ;teken in string
        CP      ' '             ;is het een spatie
        RET     NZ              ;nee, klaar
        INC     HL              ;volgende teken
        JP      SPATIES

SWAP:
;Verwisselt prioriteit en plaats in de tabel van de
;bovenste 2 operaties op de operatiestapel. Aanroep
;met in HL opwyzer.
        LD      D,(HL)          ;prioriteit laatste oper.
        DEC     HL
        LD      E,(HL)          ;plaats in tabel ervan
        DEC     HL
        LD      B,(HL)          ;prioriteit vorige oper.
        DEC     HL
        LD      C,(HL)          ;plaats in tabel ervan
        LD      (HL),E          ;plaats in tabel laatste
        INC     HL              ; operatie
        LD      (HL),D          ;prioriteit daarvan
```

```
        INC     HL
        LD      (HL),C          ;plaats in tabel vorige
        INC     HL              ; operatie
        LD      (HL),B          ;prioriteit daarvan
        RET

BEWERK:
;Voert de bewerking aan de top van de stapel uit. Aanroep
;met in HL opwyzer.
        DEC     HL              ; wyst plaats in tabel aan
        LD      E,(HL)          ;in DE verplaatsing t.o.v.
        LD      D,0             ; top tabel
        DEC     HL              ;prioriteit vlgnd. oper.
        LD      (OPWYZER),HL    ;bewaar wyzer
        LD      HL,TABEL        ;startadres tabel
        ADD     HL,DE           ;+ verplaatsing
        LD      E,(HL)          ;lage deel adres subroutine
        INC     HL
        LD      D,(HL)          ;hoge deel adres
        EX      DE,HL
        JP      (HL)            ;spring naar routine

OPWYZER:
        DEFW    00
STRING: DEFW    00
OPSTACK:
        DEFS    20              ;operatiestapel

;De tabel bevat startadressen van de reken subroutines.
TABEL:  DEFW    FPPLUS
        DEFW    FPMIN
        DEFW    FPVERM
        DEFW    FPDEEL

        END
```

Appendix A

De Z-80 instructieset

8-bit Load Group

Mnemonic	Symbolic Operation	S	Z	H	P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
LD r, r'	r ← r'	•	•	X	•	X	•	•	•	01 r r'		1	1	4	r, r' Reg.
LD r, n	r ← n	•	•	X	•	X	•	•	•	00 r 110		2	2	7	000 B
										← n →					001 C
LD r, (HL)	r ← (HL)	•	•	X	•	X	•	•	•	01 r 110		1	2	7	010 D
LD r, (IX+d)	r ← (IX+d)	•	•	X	•	X	•	•	•	11 011 101	DD	3	5	19	011 E
										01 r 101					100 H
										← d →					101 L
LD r, (IY+d)	r ← (IY+d)	•	•	X	•	X	•	•	•	11 111 101	FD	3	5	19	111 A
										01 r 110					
										← d →					
LD (HL), r	(HL) ← r	•	•	X	•	X	•	•	•	01 110 r		1	2	7	
LD (IX+d), r	(IX+d) ← r	•	•	X	•	X	•	•	•	11 011 101	DD	3	5	19	
										01 110 r					
										← d →					
LD (IY+d), r	(IY+d) ← r	•	•	X	•	X	•	•	•	11 111 101	FD	3	5	19	
										01 110 r					
										← d →					
LD (HL), n	(HL) ← n	•	•	X	•	X	•	•	•	00 110 110	36	2	3	10	
										← n →					
LD (IX+d), n	(IX+d) ← n	•	•	X	•	X	•	•	•	11 011 101	DD	4	5	19	
										00 110 110	36				
										← d →					
										← n →					
LD (IY+d), n	(IY+d) ← n	•	•	X	•	X	•	•	•	11 111 101	FD	4	5	19	
										00 110 110	36				
										← d →					
										← n →					
LD A, (BC)	A ← (BC)	•	•	X	•	X	•	•	•	00 001 010	0A	1	2	7	
LD A, (DE)	A ← (DE)	•	•	X	•	X	•	•	•	00 011 010	1A	1	2	7	
LD A, (nn)	A ← (nn)	•	•	X	•	X	•	•	•	00 111 010	3A	3	4	13	
										← n →					
										← n →					
LD (BC), A	(BC) ← A	•	•	X	•	X	•	•	•	00 000 010	02	1	2	7	
LD (DE), A	(DE) ← A	•	•	X	•	X	•	•	•	00 010 010	12	1	2	7	
LD (nn), A	(nn) ← A	•	•	X	•	X	•	•	•	00 110 010	32	3	4	13	
										← n →					
										← n →					
LD A, I	A ← I	↕	↕	X	0	X	IFF	0	•	11 101 101	ED	2	2	9	
										01 010 111	57				
LD A, R	A ← R	↕	↕	X	0	X	IFF	0	•	11 101 101	ED	2	2	9	
										01 011 111	5F				
LD I, A	I ← A	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	9	
										01 000 111	47				
LD R, A	R ← A	•	•	X	•	X	•	•	•	11 101 101	ED	2	2	9	
										01 001 111	4F				

NOTES: r, r' means any of the registers A, B, C, D, E, H, L.
IFF the content of the interrupt enable flip-flop, (IFF) is copied into the P/V flag.
For an explanation of flag notation and symbols for mnemonic tables, see Symbolic Notation section following tables.

16-Bit Load Group

Mnemonic	Symbolic Operation	Flags S	Z	H	P/V	N	C	76 543 210 Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
LD dd, nn	dd ← nn	•	•	X	•	X	•	•	•	00 dd0 001 ← n → ← n →	3	3	10	dd Pair 00 BC 01 DE
LD IX, nn	IX ← nn	•	•	X	•	X	•	•	•	11 011 101 DD 00 100 001 21 ← n → ← n →	4	4	14	10 HL 11 SP
LD IY, nn	IY ← nn	•	•	X	•	X	•	•	•	11 111 101 FD 00 100 001 21 ← n → ← n →	4	4	14	
LD HL, (nn)	$H \leftarrow (nn+1)$ $L \leftarrow (nn)$	•	•	X	•	X	•	•	•	00 101 010 2A ← n → ← n →	3	5	16	
LD dd, (nn)	$dd_H \leftarrow (nn+1)$ $dd_L \leftarrow (nn)$	•	•	X	•	X	•	•	•	11 101 101 ED 01 dd1 011 ← n → ← n →	4	6	20	
LD IX, (nn)	$IX_H \leftarrow (nn+1)$ $IX_L \leftarrow (nn)$	•	•	X	•	X	•	•	•	11 011 101 DD 00 101 010 2A ← n → ← n →	4	6	20	
LD IY, (nn)	$IY_H \leftarrow (nn+1)$ $IY_L \leftarrow (nn)$	•	•	X	•	X	•	•	•	11 111 101 FD 00 101 010 2A ← n → ← n →	4	6	20	
LD (nn), HL	$(nn+1) \leftarrow H$ $(nn) \leftarrow L$	•	•	X	•	X	•	•	•	00 100 010 22 ← n → ← n →	3	5	16	
LD (nn), dd	$(nn+1) \leftarrow dd_H$ $(nn) \leftarrow dd_L$	•	•	X	•	X	•	•	•	11 101 101 ED 01 dd0 011 ← n → ← n →	4	6	20	
LD (nn), IX	$(nn+1) \leftarrow IX_H$ $(nn) \leftarrow IX_L$	•	•	X	•	X	•	•	•	11 011 101 DD 00 100 010 22 ← n → ← n →	4	6	20	
LD (nn), IY	$(nn+1) \leftarrow IY_H$ $(nn) \leftarrow IY_L$	•	•	X	•	X	•	•	•	11 111 101 FD 00 100 010 22 ← n → ← n →	4	6	20	
LD SP, HL	SP ← HL	•	•	X	•	X	•	•	•	11 111 001 F9	1	1	6	
LD SP, IX	SP ← IX	•	•	X	•	X	•	•	•	11 011 101 DD 11 111 001 F9	2	2	10	
LD SP, IY	SP ← IY	•	•	X	•	X	•	•	•	11 111 101 FD 11 111 001 F9	2	2	10	qq Pair
PUSH qq	$(SP-2) \leftarrow qq_L$ $(SP-1) \leftarrow qq_H$ SP ← SP − 2	•	•	X	•	X	•	•	•	11 qq0 101	1	3	11	00 BC 01 DE 10 HL
PUSH IX	$(SP-2) \leftarrow IX_L$ $(SP-1) \leftarrow IX_H$ SP ← SP − 2	•	•	X	•	X	•	•	•	11 011 101 DD 11 100 101 E5	2	4	15	11 AF
PUSH IY	$(SP-2) \leftarrow IY_L$ $(SP-1) \leftarrow IY_H$ SP ← SP − 2	•	•	X	•	X	•	•	•	11 111 101 FD 11 100 101 E5	2	4	15	
POP qq	$qq_H \leftarrow (SP+1)$ $qq_L \leftarrow (SP)$ SP ← SP + 2	•	•	X	•	X	•	•	•	11 qq0 001	1	3	10	
POP IX	$IX_H \leftarrow (SP+1)$ $IX_L \leftarrow (SP)$ SP ← SP + 2	•	•	X	•	X	•	•	•	11 011 101 DD 11 100 001 E1	2	4	14	
POP IY	$IY_H \leftarrow (SP+1)$ $IY_L \leftarrow (SP)$ SP ← SP + 2	•	•	X	•	X	•	•	•	11 111 101 FD 11 100 001 E1	2	4	14	

NOTES: dd is any of the register pairs BC, DE, HL, SP.
qq is any of the register pairs AF, BC, DE, HL.
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively,
e.g., BC_L = C, AF_H = A.

Exchange, Block Transfer, Block Search Groups

Mnemonic	Symbolic Operation	Flags S	Z	H	P/V	N	C	Opcode 76 543 210 Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
EX DE, HL	DE ↔ HL	•	•	X	•	•	•	11 101 011 EB	1	1	4	
EX AF, AF'	AF ↔ AF'	•	•	X	•	•	•	00 001 000 08	1	1	4	
EXX	BC ↔ BC' DE ↔ DE' HL ↔ HL'	•	•	X	•	•	•	11 011 001 D9	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H ↔ (SP+1) L ↔ (SP)	•	•	X	•	•	•	11 100 011 E3	1	5	19	
EX (SP), IX	IX_H ↔ (SP+1) IX_L ↔ (SP)	•	•	X	•	•	•	11 011 101 DD 11 100 011 E3	2	6	23	
EX (SP), IY	IY_H ↔ (SP+1) IY_L ↔ (SP)	•	•	X	•	•	•	11 111 101 FD 11 100 011 E3	2	6	23	
LDI	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC−1	•	•	X	① 0	X	0	11 101 101 ED 10 100 000 A0	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC−1 Repeat until BC = 0	•	•	X	① 0	X	0	11 101 101 ED 10 110 000 B0	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE) ← (HL) DE ← DE−1 HL ← HL−1 BC ← BC−1	•	•	X	① 0	X	0	11 101 101 ED 10 101 000 A8	2	4	16	
LDDR	(DE) ← (HL) DE ← DE−1 HL ← HL−1 BC ← BC−1 Repeat until BC = 0	•	•	X	② 0	X	0	11 101 101 ED 10 111 000 B8	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A − (HL) HL ← HL+1 BC ← BC−1	③ ↕	↕	X	① ↕	X	1	11 101 101 ED 10 100 001 A1	2	4	16	
CPIR	A − (HL) HL ← HL+1 BC ← BC−1 Repeat until A = (HL) or BC = 0	③ ↕	↕	X	① ↕	X	1	11 101 101 ED 10 110 001 B1	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A − (HL) HL ← HL−1 BC ← BC−1	③ ↕	↕	X	① ↕	X	1	11 101 101 ED 10 101 001 A9	2	4	16	
CPDR	A − (HL) HL ← HL−1 BC ← BC−1 Repeat until A = (HL) or BC = 0	③ ↕	↕	X	① ↕	X	1	11 101 101 ED 10 111 001 B9	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

NOTES: ① P/V flag is 0 if the result of BC − 1 = 0, otherwise P/V = 1.
② P/V flag is 0 at completion of instruction only.
③ Z flag is 1 if A = (HL), otherwise Z = 0.

8-Bit Arithmetic and Logical Group

Mnemonic	Symbolic Operation	S	Z	H	P/V	N	C	Opcode 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD A, r	A ← A + r	↕	↕	X	↕	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	↕	↕	X	↕	V	0	↕ 11 000 110 ← n →		2	2	7	
ADD A, (HL)	A ← A + (HL)	↕	↕	X	↕	V	0	↕ 10 000 110		1	2	7	
ADD A, (IX+d)	A ← A + (IX+d)	↕	↕	X	↕	V	0	↕ 11 011 101 10 000 110 ← d →	DD	3	5	19	
ADD A, (IY+d)	A ← A + (IY+d)	↕	↕	X	↕	V	0	↕ 11 111 101 10 000 110 ← d →	FD	3	5	19	
ADC A, s	A ← A + s + CY	↕	↕	X	↕	V	0	↕ 001					s is any of r, n, (HL), (IX+d), (IY+d) as shown for ADD instruction. The indicated bits replace the 000 in the ADD set above.
SUB s	A ← A − s	↕	↕	X	↕	V	1	↕ 010					
SBC A, s	A ← A − s − CY	↕	↕	X	↕	V	1	↕ 011					
AND s	A ← A ∧ s	↕	↕	X	1	P	0	0 100					
OR s	A ← A ∨ s	↕	↕	X	0	P	0	0 110					
XOR s	A ← A ⊕ s	↕	↕	X	0	P	0	0 101					
CP s	A − s	↕	↕	X	↕	V	1	↕ 111					
INC r	r ← r + 1	↕	↕	X	↕	V	0	• 00 r 100		1	1	4	
INC (HL)	(HL) ← (HL) + 1	↕	↕	X	↕	V	0	• 00 110 100		1	3	11	
INC (IX+d)	(IX+d) ← (IX+d) + 1	↕	↕	X	↕	V	0	• 11 011 101 00 110 100 ← d →	DD	3	6	23	
INC (IY+d)	(IY+d) ← (IY+d) + 1	↕	↕	X	↕	V	0	• 11 111 101 00 110 100 ← d →	FD	3	6	23	
DEC m	m ← m − 1	↕	↕	X	↕	V	1	• 101					m is any of r, (HL), (IX+d), (IY+d) as shown for INC. DEC same format and states as INC. Replace 100 with 101 in opcode.

NOTE: ① P/V flag is 0 if the result of BC − 1 = 0, otherwise P/V = 1.

General-Purpose Arithmetic and CPU Control Groups

Mnemonic	Symbolic Operation	S	Z	H	P/V	N	C	Opcode 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
DAA	Converts acc. content into packed BCD following add or subtract with packed BCD operands.	↕	↕	X	↕	P	•	↕ 00 100 111	27	1	1	4	Decimal adjust accumulator.
CPL	A ← $\overline{A}$	•	•	X	1	X	•	1 • 00 101 111	2F	1	1	4	Complement accumulator (one's complement).
NEG	A ← 0 − A	↕	↕	X	↕	X	V	1 ↕ 11 101 101 01 000 100	ED 44	2	2	8	Negate acc. (two's complement).
CCF	CY ← $\overline{CY}$	•	•	X	X	X	•	0 ↕ 00 111 111	3F	1	1	4	Complement carry flag.
SCF	CY ← 1	•	•	X	0	X	•	0 1 00 110 111	37	1	1	4	Set carry flag.
NOP	No operation	•	•	X	•	X	•	• • 00 000 000	00	1	1	4	
HALT	CPU halted	•	•	X	•	X	•	• • 01 110 110	76	1	1	4	
DI *	IFF ← 0	•	•	X	•	X	•	• • 11 110 011	F3	1	1	4	
EI *	IFF ← 1	•	•	X	•	X	•	• • 11 111 011	FB	1	1	4	
IM 0	Set interrupt mode 0	•	•	X	•	X	•	• • 11 101 101 01 000 110	ED 46	2	2	8	
IM 1	Set interrupt mode 1	•	•	X	•	X	•	• • 11 101 101 01 010 110	ED 56	2	2	8	
IM 2	Set interrupt mode 2	•	•	X	•	X	•	• • 11 101 101 01 011 110	ED 5E	2	2	8	

NOTES: IFF indicates the interrupt enable flip-flop.
CY indicates the carry flip-flop.
* indicates interrupts are not sampled at the end of EI or DI.

16-Bit Arithemetic Group

ADD HL, ss	HL ← HL + ss	•	•	X	X	X	•	0	↕	00 ss1 001		1	3	11
ADC HL, ss	HL ← HL + ss + CY	↕	↕	X	X	X	V	0	↕	11 101 101 ED 01 ss1 010		2	4	15
SBC HL, ss	HL ← HL − ss − CY	↕	↕	X	X	X	V	1	↕	11 101 101 ED 01 ss0 010		2	4	15
ADD IX, pp	IX ← IX + pp	•	•	X	X	X	•	0	↕	11 011 101 DD 01 pp1 001		2	4	15
ADD IY, rr	IY ← IY + rr	•	•	X	X	X	•	0	↕	11 111 101 FD 00 rr1 001		2	4	15
INC ss	ss ← ss + 1	•	•	X	•	X	•	•	•	00 ss0 011		1	1	6
INC IX	IX ← IX + 1	•	•	X	•	X	•	•	•	11 011 101 DD 00 100 011 23		2	2	10
INC IY	IY ← IY + 1	•	•	X	•	X	•	•	•	11 111 101 FD 00 100 011 23		2	2	10
DEC ss	ss ← ss − 1	•	•	X	•	X	•	•	•	00 ss1 011		1	1	6
DEC IX	IX ← IX − 1	•	•	X	•	X	•	•	•	11 011 101 DD 00 101 011 2B		2	2	10
DEC IY	IY ← IY − 1	•	•	X	•	X	•	•	•	11 111 101 FD 00 101 011 2B		2	2	10

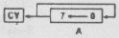
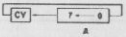
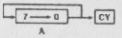
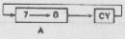
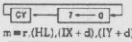
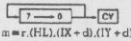
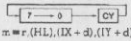
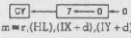
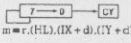
ss Reg.
00 BC
01 DE
10 HL
11 SP

pp Reg.
00 BC
01 DE
10 IX
11 SP

rr Reg.
00 BC
01 DE
10 IY
11 SP

NOTES: ss is any of the register pairs BC, DE, HL, SP.
pp is any of the register pairs BC, DE, IX, SP.
rr is any of the register pairs BC, DE, IY, SP.

Rotate and Shift Group

RLCA		•	•	X	0	X	•	0	↕	00 000 111	07	1	1	4
RLA		•	•	X	0	X	•	0	↕	00 010 111	17	1	1	4
RRCA		•	•	X	0	X	•	0	↕	00 001 111	0F	1	1	4
RRA		•	•	X	0	X	•	0	↕	00 011 111	1F	1	1	4
RLC r		↕	↕	X	0	X	P	0	↕	11 001 011 00 000 r	CB	2	2	8
RLC (HL)		↕	↕	X	0	X	P	0	↕	11 001 011 00 000 110	CB	2	4	15
RLC (IX + d)	 r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	11 011 101 DD 11 001 011 CB ← d → 00 000 110		4	6	23
RLC (IY + d)		↕	↕	X	0	X	P	0	↕	11 111 101 FD 11 001 011 CB ← d → 00 000 110		4	6	23
RL m	 m = r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	00 010				
RRC m	 m = r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	00 001				
RR m	 m = r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	00 011				
SLA m	 m = r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	00 100				
SRA m	 m = r,(HL),(IX + d),(IY + d)	↕	↕	X	0	X	P	0	↕	00 101				

r Reg.
000 B
001 C
010 D
011 E
100 H
101 L
111 A

Rotate left circular accumulator.

Rotate left accumulator.

Rotate right circular accumulator.

Rotate right accumulator.

Rotate left circular register r.

Instruction format and states are as shown for RLC's. To form new opcode replace 000 of RLC's with shown code.

Mnemonic	Symbolic Operation	S	Z	H	P/V	N	C	Opcode 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
SRL m	0 → m → CY m = r,(HL),(IX + d),(IY + d)	↕	↕	X	P	0	↕	11					
RLD	A (HL)	↕	↕	X	P	0	•	11 101 101 01 101 111	ED 6F	2	5	18	Rotate digit left and right between the accumulator and location (HL).
RRD	A (HL)	↕	↕	X	P	0	•	11 101 101 01 100 111	ED 67	2	5	18	The content of the upper half of the accumulator is unaffected.

Bit Set, Reset and Test Group

BIT b, r	Z ← r_b	X	↕	X	1	X	0	•	11 001 011 CB 01 b r	2	2	8	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A b Bit Tested 000 0 001 1 010 2 011 3 100 4 101 5 110 6 111 7
BIT b, (HL)	Z ← $(\overline{HL})_b$	X	↕	X	1	X	0	•	11 001 011 CB 01 b 110	2	3	12	
BIT b, (IX + d)$_b$	Z ← $(\overline{IX+d})_b$	X	↕	X	1	X	0	•	11 011 101 DD 11 001 011 CB ← d → 01 b 110	4	5	20	
BIT b, (IY + d)$_b$	Z ← $(\overline{IY+d})_b$	X	↕	X	1	X	0	•	11 111 101 FD 11 001 011 CB ← d → 01 b 110	4	5	20	
SET b, r	r_b ← 1	•	•	X	•	X	•	•	11 001 011 CB 11 b r	2	2	8	
SET b, (HL)	$(HL)_b$ ← 1	•	•	X	•	X	•	•	11 001 011 CB 11 b 110	2	4	15	
SET b, (IX + d)	(IX + d)$_b$ ← 1	•	•	X	•	X	•	•	11 011 101 DD 11 001 011 CB ← d → 11 b 110	4	6	23	
SET b, (IY + d)	(IY + d)$_b$ ← 1	•	•	X	•	X	•	•	11 111 101 FD 11 001 011 CB ← d → 11 b 110	4	6	23	
RES b, m	m_b ← 0 m = r, (HL), (IX + d), (IY + d)	•	•	X	•	X	•	•	10				To form new opcode replace 11 of SET b, s with 10. Flags and time states for SET instruction.

NOTES: The notation m_b indicates bit b (0 to 7) of location m.

Jump Group

JP nn	PC ← nn	•	•	X	•	X	•	•	11 000 011 C3 ← n → ← n →	3	3	10	
JP cc, nn	If condition cc is true PC ← nn, otherwise continue	•	•	X	•	X	•	•	11 cc 010 ← n → ← n →	3	3	10	cc Condition 000 NZ non-zero 001 Z zero 010 NC non-carry 011 C carry 100 PO parity odd 101 PE parity even 110 P sign positive 111 M sign negative
JR e	PC ← PC + e	•	•	X	•	X	•	•	00 011 000 18 ← e − 2 →	2	3	12	
JR C, e	If C = 0, continue If C = 1, PC ← PC + e	•	•	X	•	X	•	•	00 111 000 38 ← e − 2 →	2 2	2 3	7 12	If condition not met. If condition is met.
JR NC, e	If C = 1, continue If C = 0, PC ← PC + e	•	•	X	•	X	•	•	00 110 000 30 ← e − 2 →	2 2	2 3	7 12	If condition not met. If condition is met.

Jump Group (Continued)

Mnemonic	Symbolic Operation	S	Z	X	H	X	P/V	N	C	Opcode 76 543 210 Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
JP Z, e	If Z = 0 continue	•	•	X	•	X	•	•	•	00 101 000 28 ← e−2 →	2	2	7	If condition not met.
	If Z = 1, PC ← PC+e										2	3	12	If condition is met.
JP NZ, e	If Z = 1, continue	•	•	X	•	X	•	•	•	00 100 000 20 ← e−2 →	2	2	7	If condition not met.
	If Z = 0, PC ← PC+e										2	3	12	If condition is met.
JP (HL)	PC ← HL	•	•	X	•	X	•	•	•	11 101 001 E9	1	1	4	
JP (IX)	PC ← IX	•	•	X	•	X	•	•	•	11 011 101 DD 11 101 001 E9	2	2	8	
JP (IY)	PC ← IY	•	•	X	•	X	•	•	•	11 111 101 FD 11 101 001 E9	2	2	8	
DJNZ, e	B ← B−1 If B = 0, continue	•	•	X	•	X	•	•	•	00 010 000 10 ← e−2 →	2	2	8	If B = 0.
	If B ≠ 0, PC ← PC+e										2	3	13	If B ≠ 0.

NOTES: e represents the extension in the relative addressing mode.
e is a signed two's complement number in the range <-126, 129>.
e−2 in the opcode provides an effective address of pc + e as PC is incremented by 2 prior to the addition of e.

Call and Return Group

CALL nn	(SP−1) ← PC_H (SP−2) ← PC_L PC ← nn	•	•	X	•	X	•	•	•	11 001 101 CD ← n → ← n →	3	5	17	
CALL cc, nn	If condition cc is false continue, otherwise same as CALL nn	•	•	X	•	X	•	•	•	11 cc 100 ← n → ← n →	3	3	10	If cc is false.
											3	5	17	If cc is true.
RET	PC_L ← (SP) PC_H ← (SP+1)	•	•	X	•	X	•	•	•	11 001 001 C9	1	3	10	
RET cc	If condition cc is false continue, otherwise same as RET	•	•	X	•	X	•	•	•	11 cc 000	1	1	5	If cc is false.
											1	3	11	If cc is true.
RETI	Return from interrupt	•	•	X	•	X	•	•	•	11 101 101 ED 01 001 101 4D	2	4	14	
RETN[1]	Return from non-maskable interrupt	•	•	X	•	X	•	•	•	11 101 101 ED 01 000 101 45	2	4	14	
RST p	(SP−1) ← PC_H (SP−2) ← PC_L PC_H ← 0 PC_L ← p	•	•	X	•	X	•	•	•	11 t 111	1	3	11	

cc	Condition
000	NZ non-zero
001	Z zero
010	NC non-carry
011	C carry
100	PO parity odd
101	PE parity even
110	P sign positive
111	M sign negative

t	p
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

NOTE: [1]RETN loads IFF_2 → IFF_1

Input and Output Group

Mnemonic	Symbolic Operation	Flags S	Z	H	P/V	N	C	Opcode 76 543 210	Hex	No. of Bytes	No. of M Cycles	No. of T States	Comments
IN A, (n)	A ← (n)	*	*	X	*	X	*	*	*	11 011 011	DB	2	3	11	n to $A_0 \sim A_7$ Acc. to $A_8 \sim A_{15}$
								← n →							
IN r, (C)	r ← (C) if r = 110 only the flags will be affected	↕	↕	X	↕	X	P	0	*	11 101 101 01 r 000	ED	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
			①												
INI	(HL) ← (C) B ← B − 1 HL ← HL + 1	X	↕	X	X	X	X	1	X	11 101 101 10 100 010	ED A2	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
			②												
INIR	(HL) ← (C) B ← B − 1 HL ← HL + 1 Repeat until B = 0	X	↕	X	X	X	X	1	X	11 101 101 10 110 010	ED B2	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
			①												
IND	(HL) ← (C) B ← B − 1 HL ← HL − 1	X	↕	X	X	X	X	1	X	11 101 101 10 101 010	ED AA	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
			②												
INDR	(HL) ← (C) B ← B − 1 HL ← HL − 1 Repeat until B = 0	X	↕	X	X	X	X	1	X	11 101 101 10 111 010	ED BA	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUT (n), A	(n) ← A	*	*	X	*	X	*	*	*	11 010 011 ← n →	D3	2	3	11	n to $A_0 \sim A_7$ Acc. to $A_8 \sim A_{15}$
OUT (C), r	(C) ← r	*	*	X	*	X	*	*	*	11 101 101 01 r 001	ED	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
			①												
OUTI	(C) ← (HL) B ← B − 1 HL ← HL + 1	X	↕	X	X	X	X	1	X	11 101 101 10 100 011	ED A3	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
			②												
OTIR	(C) ← (HL) B ← B − 1 HL ← HL + 1 Repeat until B = 0	X	↕	X	X	X	X	1	X	11 101 101 10 110 011	ED B3	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
			①												
OUTD	(C) ← (HL) B ← B − 1 HL ← HL − 1	X	↕	X	X	X	X	1	X	11 101 101 10 101 011	ED AB	2	4	16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTDR	(C) ← (HL) B ← B − 1 HL ← HL − 1 Repeat until B = 0	X	↕	X	X	X	X	1	X	11 101 101 10 111 011	ED BB	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$

NOTE: ① If the result of B − 1 is zero the Z flag is set, otherwise it is reset.
② Z flag is set upon instruction completion only.

Index

De Z-80 kan zonder twijfel worden beschouwd als één van de meest gebruikte 8 bits-microprocessoren. Een aantal besturingssystemen is op deze processor gebaseerd en de Z-80 is bovendien het 'hart' in alle MSX-computers.
Het in machinetaal programmeren van de Z-80 is geen eenvoudige opgave. De uitgebreide instructieset maakt het werken met een zogenaamde assembler noodzakelijk.
Dit boek behandelt het programmeren in Z-80 machinetaal zeer grondig. Aan de hand van vele programmavoorbeelden krijgt de lezer een inzicht in het gestructureerd programmeren.
Ook gevorderde programmeurs komen in dit boek aan hun trekken; onderwerpen als het zogenaamde floating point-rekenen, één van de moeilijkste onderdelen in machinetaal, worden op een overzichtelijke manier behandeld.
De vele afbeeldingen dragen ertoe bij dat dit boek een waardevol bezit is voor Z-80 programmeurs en voor hen die dat willen worden.

De informatie in dit boek is van toepassing op onder andere:

Alle MSX- en MSX2-computers
De Schneider/Amstrad homecomputers
De Sinclair homecomputers
De Commodore 128
Alle computers met het CP/M-80 besturingssysteem

ISBN 90 201 2004 2