een volwassen computer voor beginners

1

JUNIOR COMPUTER

uitgeversmij.
elektuur b.v.

Ton van der Mark

Junior-computer 1

een volwassen computer voor beginners

A.Nachtmann
G.H.Nachbar

Uitgave:
Uitgeversmaatschappij Elektuur B.V.
Postbus 75 6190 AB Beek (L)

Eerste druk april 1980
Tweede druk september 1980
Derde druk december 1980

Printed in the Netherlands.
ISBN 90 70160 15 3

Waarom dit boek?

Voor velen betekent de computer minder werk en meer vrije tijd, voor anderen meer werk in hun vrije tijd: meer en meer hobbyisten sluiten vriendschap met deze twintigste-eeuwse "digitale slaven". Iedereen die zich met deze boeiende hobby bezig houdt of gaat houden kan kiezen uit een groot aanbod van kant-en-klaar- of zelfbouw-computers, en aan boeken en min of meer gespecialiseerde tijdschriften op dit gebied.
Prima!
En toch moeten we vaststellen dat men zich weliswaar serieus wil oriënteren, maar soms door de bomen het bos niet meer ziet. Of struikelt over de wortels van een boom. Anders gezegd: Men weet niet goed waar en hoe te beginnen en als men begint kan de kennismaking negatief uitvallen. En men struikelt liever geen tweede keer. Dat is eeuwig jammer. En onnodig.
Ziedaar het "waarom" van dit boek, en van het aansluitende deel 2.
Waar gaat dit boek over? Over de junior-computer, een volwassen microcomputer die zich jong voelt. Hij is uitgerust met de 6502-microprocessor en dat is bepaald niet de eerste de beste. Dat is een bewuste keuze omdat we van mening zijn dat je sneller piano leert spelen op een echte piano dan op een stuk kinderspeelgoed. Bovendien kent het basissysteem zoals in dit boek beschreven vele uitbreidingsmogelijkheden die tezijnertijd, als men er rijp voor is geheel of gedeeltelijk kunnen worden benut.
De junior-computer is een zelfbouw-computer. Dat scheelt een stuk op de kosten. De bezwaren die hier voor sommigen aan verbonden zouden kunnen zijn gelden hier ons insziens niet: de opzet van de junior-computer is kompakt; akrobatische vaardigheden met de soldeerbout zijn niet vereist. En dan is er ook nog een zeer uitgebreide bouwhandleiding. Hoofdstuk 1 is daar grotendeels aan gewijd; daarin ook een algemene inleiding over micro-computers: het valt allemaal wel mee met die "moeilijke" computer.
Hoofdstuk 2 gaat over zaken die we allemaal nog wel kennen van de lagere school: over taal en rekenen. Weten we hoe de digitale opvattingen van de (junior-)computer daarover zijn dan kunnen we in hoofdstuk 3 aan de weet komen hoe we via het programmeren met hem kunnen omgaan. Hoofdstuk 4 tenslotte geeft een aantal programmavoorbeelden. In hoofdstuk 3 en 4 kan overigens dankbaar gebruik worden gemaakt van de komplete, opgebouwde junior-computer.
Wat is nou de filosofie achter dit boek? Deze: Het is een misverstand om te denken dat "eenvoudig" hetzelfde is als "oppervlakkig". En deze: Het is niet verstandig om alles tegelijk te willen. Vandaar de dosering van de te verwerven kennis en vandaar de opdeling in twee boeken. In dit boek staat alles om te kunnen programmeren "zonder poespas". En op een gegeven moment is men gevorderd tot boek 2, waarin men kennis kan nemen van nieuwe mogelijkheden met de "oude" junior-computer.
We hadden het net over filosofie. Een Nederlands woord daarvoor is wijsbegeerte. Wij hopen dat dit boek ertoe zal bijdragen dat de begeerte tot computer-wijsheid zal worden opgewekt – en bevredigd.

De schrijvers

Junior-computer 2 is het vervolg van dit boek. Het boek is gewijd aan:

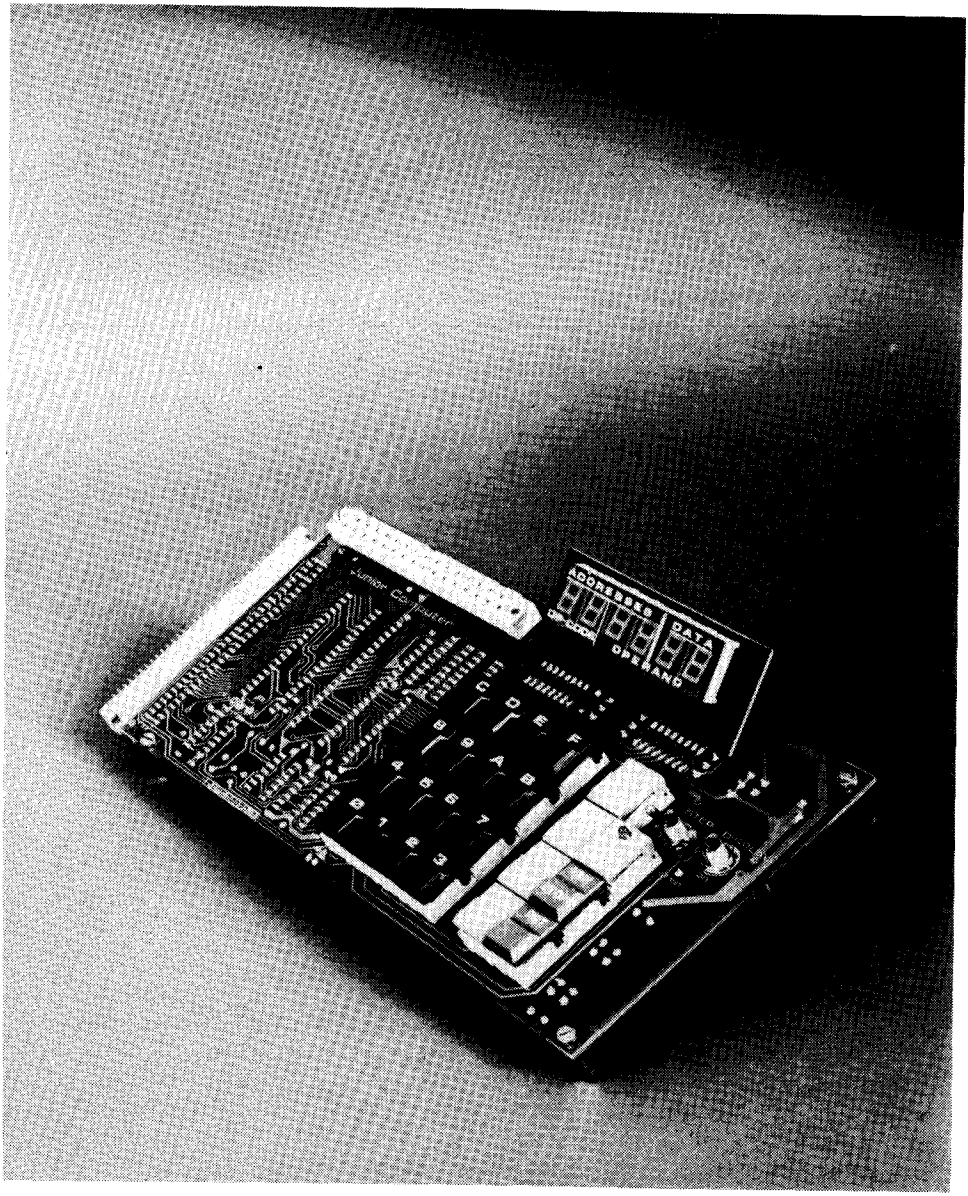
- De I/O-bouwsteen van de junior-computer en de programmering daarvan
- Het monitorprogramma, met al zijn mogelijkheden voor de gebruiker
- De hex editor
- De hex assembler
- De listing van de monitor, de editor, de assembler en van alle voorbeeldprogramma's uit dat boek.

Deel 2 vormt samen met dit boek de totale beschrijving van de standaard-junior-computer. In een derde deel, *Junior-Computer 3,* wordt aandacht besteed aan hardware- en software-uitbreidingen van de junior-computer, zoals een kassette-interface, geheugenuitbreiding, de aansluiting op een video-display + ASCII-toetsenbord, en nog veel meer.
Daarnaast verschijnen regelmatig artikelen (aanvullende opmerkingen, gebruikersprogramma's enz.) in het tijdschrift Elektuur.

Dit boek is door mensen gemaakt. De auteurs stellen schriftelijke op- of aanmerkingen van de lezers over de inhoud op prijs.

De printen uit dit boek zijn opgenomen in de EPS (Elektuur-printservice). Voor nadere informatie wordt verwezen naar de laatstverschenen uitgave van Elektuur. Daar treft men ook nadere informatie aan over de Elektuur-Software-service.

Inhoud


ADDRESSED
OP CODE
DATA
OPERAND
A
B
C
D
E
F
0
1
2
3
4
5
6
7
8
9

1

Eerste kennismaking

waarom en hoe?

Bij de junior-computer is "junior" minstens zo belangrijk als "computer". Dat betekent dat we het bij de inleidende beschrijving in dit hoofdstuk tot onze taak rekenen om allerlei misverstanden en vooroordelen uit de weg te ruimen — zonder nieuwe te creëren.
Dus niet bang zijn voor de (junior-)computer! Niet bij het (in dit hoofdstuk) in elkaar zetten van de hardware, het "vlees en bloed", en ook niet (in latere hoofdstukken) bij de omgang met de "geest" van de computer, de software.

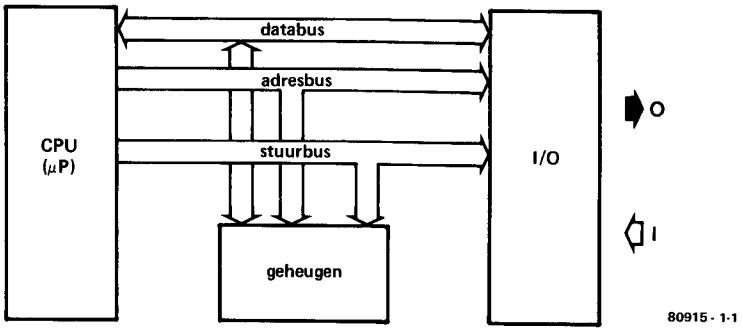
In principe zit een microcomputer eenvoudig in elkaar. Dit ondanks de voor velen misschien afschrikwekkende hoeveelheid elektronica, die tot de voorbarige konklusie kan leiden dat men er ongeschikt voor is.
Een misverstand, vinden we. Het komt allemaal doordat aan de elektronica door de (kandidaat-)computeramateur een te grote rol wordt toegekend. Bij elk elektronisch apparaat ligt de nadruk op het gebruik of de toepassing ervan; dat je aan het zelf bouwen of het begrijpen van de schakeling veel plezier kunt beleven is mooi meegenomen.
Het doel of de toepassing van een mikrocomputer is de uitvoering van logische opdrachten, die hem door de gebruiker via het programma zijn opgedragen. De uitdaging ligt niet zozeer in het doorgronden van de elektronica alswel in het bedenken van leuke en zinvolle opdrachten, die de computer "aan kan". Vergelijk het met een auto: om ermee om te kunnen gaan is het helemaal niet van belang om te weten hoe deze in elkaar zit; laat dat maar aan de specialisten op dat terrein over.
Dus beschouw de computer, in ons geval de junior-computer als een "black box" (een andere kleur van het kastje mag ook) en richt uw aandacht op de buitenkant: hoe kan ik via het toetsenbord de computer aan het werk zetten en hoe ziet het resultaat van mijn werk en dat van de computer eruit (op het display)?

Het zwart maken van de doos

Voordat we de kast van de junior-computer hebben dichtgeschroefd en we verder naar de inhoud geen omkijken meer hebben moet de junior-computer worden gebouwd. Door u, want het is een zelfbouw-computer. De rest van dit hoofdstuk staat in het teken van de werking en de bouw van de junior-computer, en de rest van dit boek in het teken van: wat kun je ermee doen. Daarbij is het van belang om al te beschikken over de black box, die na aansluiting op het lichtnet klaar staat voor het nemen van initiatieven. Uw initiatieven, wel te verstaan.
Maar zien we bij dat zelf bouwen die mensen niet over het hoofd, bij wie het eerder genoemde elektronica-schrikbeeld opdoemt? Nee, want we helpen u opweg via een goede bouwbeschrijving. We helpen u door de zure appel heenbijten. Trouwens, het valt allemaal wel mee. Echt.

Een beetje computerkunde

Het spelen met blokken is een van de eerste zaken die men onder de knie heeft. Waarom dan niet via een blokschema de computer terugbrengen tot aanvaardbare, dus de juiste proporties? Figuur 1 toont zo'n blokschema, met drie blokken en drie soorten verbindingen tussen de blokken. Een computer werkt aan de hand van informatie, die in een voor hem verteerbare vorm, namelijk elektrisch-digitaal is gegoten. Men spreekt ook wel van *data*. Informatie-uitwisseling tussen ofwel datatransport van en naar de diverse blokken vindt plaats via de *databus*.



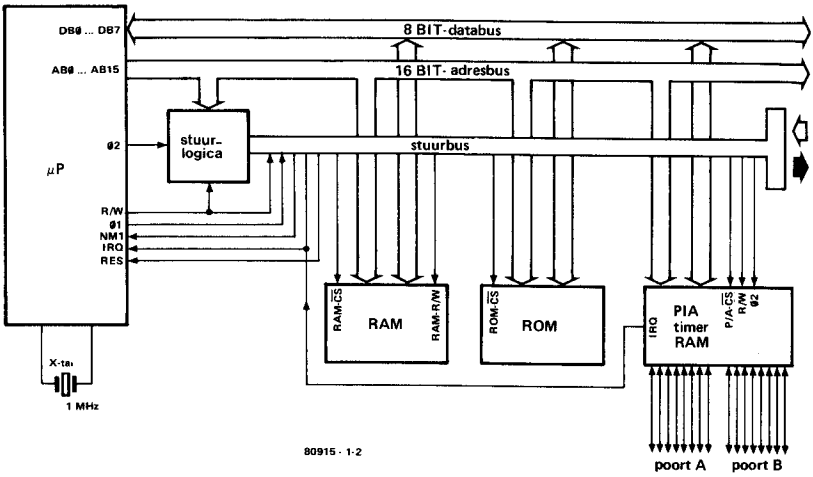
Figuur 1. Simpeler kan het niet: het elementaire blokschema van een computer bestaat uit drie blokken en drie bussen, welke laatste zorgen voor de onderlinge verbindingen.

Een computer is zinloos zonder buitenwereld; dat kan een toetsenbord zijn, een beeldscherm, een display, maar ook een olieraffinaderij. Als tussenstation (interface) tussen computer en buitenwereld dient het blok I/O, van Input/Output, waarmee het dataverkeer van en naar buiten wordt geregeld. De data van de databus wordt of is behandeld door de CPU, Central Processing Unit, die ook wel bekend staat als het centrale brein van de computer, hoewel men het betrekkelijke van die intelligentie nog

wel leert inzien. Via de CPU worden alle opdrachten afgehandeld. Zodra een bepaalde opdracht is uitgevoerd wordt er gekeken hoe de nieuwe opdracht luidt. Daartoe gaat de CPU te rade bij een deel van het geheugen, waarin de opdrachten in kode zijn vastgelegd. Het geheel aan opdrachten ofwel instrukties dat hoort bij een bepaalde toepassing van de computer heet *programma.* Is de CPU op één "chip" geïntegreerd, dan spreken we van een *microprocessor.*
In het geheugen onthoudt de computer dus wat je hem gezegd hebt te doen. De computer is dus niet intelligent op eigen initiatief ("wat zal ik nou weer eens bedenken"), maar is slechts "voorgekouwd" intelligent ("ik moet nu dat en dat bedenken, dus handel ik er maar naar"). Het geheugen wordt niet alleen voor de opslag van het programma gebruikt; er ligt ook data in kode in opgeslagen die nodig is bij het uitvoeren van opdrachten, of die als gevolg daarvan beschikbaar komt.
In een computer is data de grondstof voor de bewerkingen, maar ook de in kode vastgelegde omschrijving van de bewerkingen (programma). Die grondstof is voorbewerkt, namelijk in elektrisch-digitale vorm gegoten. Er is dataverkeer tussen de drie blokken, zoals het ophalen door de CPU van een volgende opdracht, en dataverkeer vanuit of naar de periferie, de buitenwereld. Voor het selekteren van data en van andere zaken, zoals bepaalde ingangen of uitgangen, die in een bepaalde programmafase van belang zijn is een selektie in de vorm van adressering nodig: op plaats zus en zo bevindt zich de op dat moment nodige data, of: via die en die uitgang (poort) moet data worden uitgevoerd of ingevoerd. De *adresbus* bevat in kode voor elke uit te voeren opdracht de plaats die het beginpunt of eindpunt vormt van het bij die opdracht horende datatransport.
Tot slot de *stuurbus* of, op z'n Engels, de control bus. Die is er voor het regelen van een aantal interne huishoudelijke zaken. Hij bevat ondermeer stuursignalen, die de aard en de richting van het datatransport beïnvloeden. Die stuursignalen zijn afkomstig uit de CPU (μP), hetzij "op eigen initiatief" hetzij onder invloed van buiten. Tot de huishoudelijke taken behoort ook het regelen van de voortgang van de opeenvolgende programma-onderdelen, dus van de opdrachten of instrukties. Daartoe wordt de CPU periodiek geprikkeld door iets dat je zou kunnen vergelijken met het veermechanisme van een wekker, of met de kleine hersenen, die de hartslag van het systeem in stand houden.

Nu iets technischer

Van het algemene blokschema van figuur 1 stappen we over op dat van de junior-computer, in figuur 2. Allereerst de bussen. De adresbus bestaat uit zestien elektrische leidingen. Elke leiding kan onafhankelijk van de andere twee (elektrische) toestanden aannemen: er staat al dan niet een spanning op. Bij zestien leidingen zijn er 2^{16}, dat zijn 65536 verschillende toestanden, dus een dikke 65-duizend verschillende plaatsen die de start of finish vormen van datatransport. Bij elke leiding hoort een bit, vandaar de 16-bits adresbus van figuur 2. Bits houden verband met de hoeveelheid informatie die in digitale kode kan worden bevat. Meer hierover in hoofdstuk 2.
Via de databus, die uit acht leidingen bestaat vindt het dataverkeer plaats. Er is in twee richtingen verkeer mogelijk; de databus is zoals dat zo mooi



Figuur 2. Het blokschema van figuur 1, uitgewerkt voor een konkrete computer: de junior-computer.

heet bidirektioneel. Echter nooit in twee richtingen tegelijk, dus eenrichtingsverkeer in de ene òf in de andere, tegengestelde richting. Van invloed op de richting is het lees/schrijfsignaal R/$\overline{W}$ = Read/$\overline{Write}$) van de stuurbus. Dit signaal bepaalt hoe bepaalde databuffers (eentje voor elke leiding) zich op hun beide aansluitingen gedragen, dus welke aansluiting uitgang wordt (waarbij de andere automatisch ingang wordt). Vergelijk het met een draaibaar verkeersbord. De ene kant is rond en rood en heeft een horizontale witte balk; de andere is vierkant en blauw en toont een witte, omhoog gerichte pijl: de twee bekende eenrichtingsverkeersborden. Technisch wordt e.e.a. gerealiseerd met zogenaamde tri-state-logica.

Geheugens: de data-onthouders

Geheugens zijn er in twee hoofdsoorten; beide zijn onontbeerlijk voor de goede werking van een computer. Vandaar de twee blokken RAM en ROM in figuur 2, in tegenstelling tot het ene blok van figuur 1. Er is permanente data, bijvoorbeeld de data die hoort bij de *monitor*, het basisprogramma van de computer. En er is data die veranderlijk is, bijvoorbeeld de data van een programma dat de gebruiker maakt (onder gebruikmaking van de monitor) en nog een aantal keren moet (kunnen) wijzigen alvorens het naar behoren werkt. Verder heb je de veranderlijke data als input voor of output van het programma.
Permanente data hoeft als deze eenmaal ergens vooraf is opgeborgen alleen maar te worden opgehaald. Niet-permanente data moet je zowel kunnen halen als brengen. Bij geheugens spreken we niet van data "halen" maar van data "(uit)lezen", en van data "(weg)schrijven" in plaats van "brengen". Er zijn geheugens die uitsluitend kunnen worden gelezen. Zo'n geheugen noemen we ROM, van Read Only Memory[1]. Het *programmageheugen* heeft een ROM-karakter. Een ander soort geheugen kan worden gelezen

èn beschreven: de RAM, van Random Acces Memory. Het *werkgeheugen* heeft een RAM-karakter. Ziehier een van de redenen voor het datatransport in twee richtingen via de databus: Hetzelfde lees/schrijfsignaal dat de datatransportrichting bepaalt legt ook vast of de RAM wordt gelezen (data eruit) of beschreven (data erin).
Overigens: we zeggen nou wel dat lezen overeen komt met data halen, maar de data staat na het lezen nog steeds op z'n (geheugen)plaats. In feite wordt bij het lezen data uit het geheugen gekopieerd in de CPU. Immers: na het uitlezen van een boek zijn de letters óók niet van het papier verdwenen. Hoewel . . . RAM's verliezen hun inhoud na het uitschakelen of wegvallen van hun voedingsspanning. In RAM opgeslagen data die men wil bewaren kan men overigens opbergen in een *achtergrondgeheugen,* bijvoorbeeld een floppy disc of kassetteband.

I/O: de portier

Het blok I/O onderhoudt de kontakten met de buitenwereld (en hoe kleiner de computer, des te groter de buitenwereld). In figuur 2 vinden we het blok terug als PIA, Peripheral Interface Adapter. Weliswaar gaan ook de drie bussen naar buiten (de drie pijlen rechts in figuur 2), maar dat is bedoeld voor uitbreiding van de junior-computer, dus uitbreiding van de "binnenwereld". In ons geval bestaat die buitenwereld in ieder geval uit het hexadecimale toetsenbord en het display.
Net als bij de geheugens is datatransport in twee richtingen noodzakelijk, dus mogelijk. De databus met z'n acht leidingen (acht bits) wordt naar buiten voortgezet in de vorm van twee bundels van elk acht leidingen, die heel toepasselijk *poorten* heten, en wel poort A en poort B. De poortselektie gaat via de adresbus. Elke poortleiding kan op een bepaald moment onafhankelijk van de andere 15 leidingen worden benoemd tot ingang danwel uitgang.
In de PIA kan (tijdelijk) data worden opgeslagen. Hetzij data voor de buitenwereld hetzij data van buiten, waaraan de computer een boodschap heeft. Hiertoe is een aantal in twee richtingen werkende (altijd maar in één richting tegelijk) buffers aanwezig in de vorm van een register. De stuurbus levert de nodige signalen voor het lezen van of schrijven in de poorten van de PIA.

De µP: het aktiecentrum

Aangemoedigd door een ingebouwde "pace maker", de klokgenerator (met het externe 1 MHz kristal XTAL) stuurt de mikroprocessor via de drie bussen het komplete systeem. De klokgenerator levert twee signalen

---

[1] *Naast de ROM, die tijdens het fabrikageproces wordt gevuld met de gewenste data (masker-geprogrammeerd) zijn er varianten. De EPROM (Erasable, user-Programmable ROM) kan men zelf van data voorzien met een ultraviolet-"data-inbrandmachine", de EPROM-programmer. De ROM is niet herprogrammeerbaar, de EPROM wel, evenals de EAROM. De PROM daarentegen weer niet; de data ligt vast door het al dan niet verbreken van "fusable links", dat zijn onderbreekbare verbindingen. Nadat dit eenmaal is gebeurd kunnen gedane zaken geen keer nemen.*

Ø1 en Ø2, die bij elke uit te voeren instruktie een belangrijke rol spelen bij het voorbereiden van de adresbus resp. de databus. We spreken van een *tweefasenklok*. Ook de eigenlijke data-stofwisseling gebeurt hier. Een onderdeel is de *programmateller* (PC = Program Counter). Deze levert de adresbus de nodige adresinformatie, de adressen dus. De adressen zijn afkomstig van andere plaatsen in de μP. Voor het adres van de volgende uit te voeren programma-instruktie is de stand van een bepaalde interne teller van belang, die de uitvoering van de elkaar opvolgende instrukties regelt. Adressen van data als grondstof, halffabrikaat of eindprodukt van het programma volgen direkt of via berekening uit het programma.

Mag ik even storen?

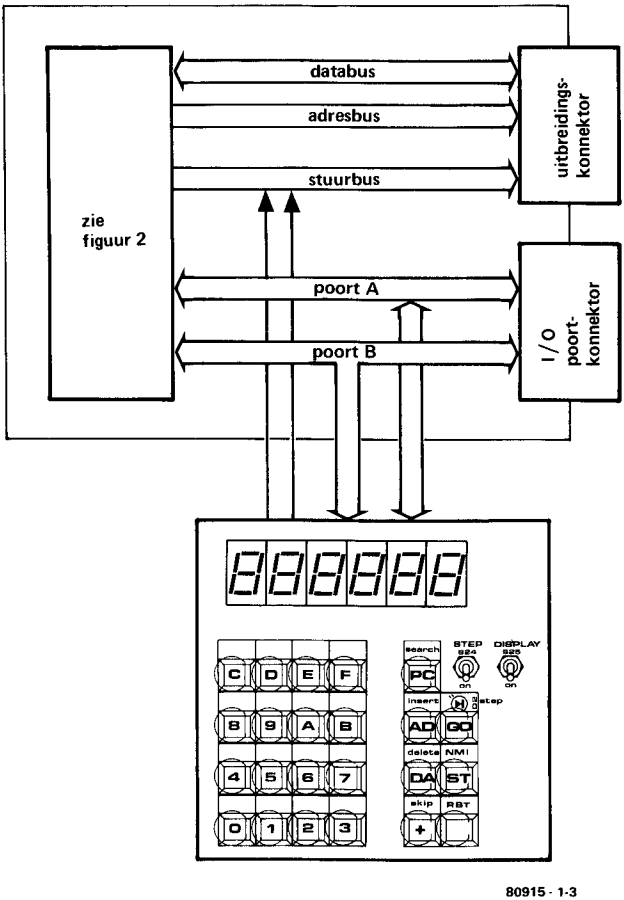
We moeten het nog hebben over drie signalen in figuur 2. Over RES, IRQ en NMI. Over RES, van Reset kunnen we kort zijn: met dit signaal wordt de junior-computer na inschakelen gestart. De signalen IRQ, van Interrupt ReQuest, en NMI, van Non-Maskable Interrupt zijn er om de computer op konstruktieve wijze te storen in de uitvoering van het lopende programma. Te storen voor min of meer dringende berichten. Hetgeen leidt tot een werkonderbreking.
Initiatief daartoe komt van buiten. De aandacht van de computer wordt gevraagd voor het feit dat er een toets van bijvoorbeeld een ASCII-toetsenbord is ingedrukt, hetgeen erop duidt dat de gebruiker hem iets te vertellen heeft, of een bepaald perifeer apparaat, bijvoorbeeld een printer vraagt om snelle bediening. Zodra zo'n "stoorpuls" wordt ontvangen breekt de computer het programma af waar ie mee bezig was en handelt het interrupt-programma af dat hoort bij de aard van de stoorpulsverwekker. Na gedane zaken wordt het onderbroken programma hervat.
Meestal zijn er meerdere onderbrekingsaanvragers. In dat geval krijgt elke onderbreker een bepaalde prioriteit toegekend. Hoe hoger de prioriteit des te sneller men wordt behandeld. Een belangrijk verschil tussen IRQ en NMI is dat IRQ via en door het programma kan worden genegeerd (denk aan het bordje "niet storen"), een NMI niet ("brandmelding").

En verder nog . . .

In het blok PIA van figuur 2 zit ook nog een programmeerbare timer. Deze komt in een apart hoofdstuk aan de orde, hoofdstuk 5. Het blok stuurlogica bevat een zogenaamde adresdekoder. Hiermee worden de diverse chip-selekt-signalen gemaakt, die een bepaald geheugendeel of de PIA klaar zetten voor gebruik, zodat dat men ermee kan lezen en schrijven.
Tot zover de bespreking in blokken van het kwa funktie inwendige deel van de junior-computer. Het wordt nu tijd om naar buiten te gaan.

Ogen en handen

De tussenstations tussen junior (gebruiker) en computer maken de computer kompleet. Dit is in figuur 3 in beeld gebracht: het blokschema van figuur 2 krijgt kop(toetsenbord) en staart(display), waardoor de gebruiker ogen en handen kan gebruiken in de omgang met de computer.
Er is nog meer. De drie bussen zijn via de "expansion connector" beschikbaar, de twee poorten A en B via de I/O-connector. Waarom? Wel, laten



80915 - 1-3

Figuur 3. Ten opzichte van het blokschema van figuur 2 een stap verder: kontakten met de buitenwereld zijn nu mogelijk geworden omdat een toetsenbord aanwezig is voor de invoer (de I = Input van I/O) van gegevens (data), en een display met zes hexadecimale tekens voor de uitvoer (de O = Output van I/O) van gegevens.

we reëel zijn. Het hexadecimale toetsenbord als standaard-invoerorgaan (Input, I) en het hexadecimale display als standaard-uitvoerorgaan (Output, O) zijn verreweg de simpelste (en goedkoopste!) mogelijkheden om "op een beetje nivo" met de computer van gedachten te wisselen. Er zijn nog veel meer I/O-mogelijkheden, die de kwaliteit van de konversatie verhogen, of het aantal konversatiemogelijkheden. Allemaal leuk en aardig, maar dat vereist ook meer geld en kennis. De meeste junioren hebben weinig geld en kennis. Maar junioren groeien uit tot senioren. We vinden dat de junior-computer kan en moet meegroeien.
De uitbreidingsconnector kan overigens ook worden gebruikt voor uit-

breiding van de geheugenkapaciteit, zodat men grotere programma's kan maken. Ook hier geldt dat dit voor de junior eigenlijk pas van later zorg is. Voor de volledigheid zij nog vermeld dat de uitbreidingsconnector 64-polig is en bovendien verenigbaar (compatibel) met de Elektuur-SC/MP-bus. De zestien leidingen van de poorten A en B zijn aangesloten op een 31-polige connector.
Toetsenbord en display hangen via de poorten A en B aan de eigenlijke computer; voor poort A is tweerichtingsverkeer mogelijk, poort B wordt slechts in één richting benut. Er worden twee signalen op de stuurbus gezet. Dat zijn de signalen RES en NMI, die horen bij de toetsen RST en ST, rechtsonder op het toetsenbord.
Bij het (leren) omgaan met de junior-computer zullen we het toetsenbord en het display uitgebreid leren kennen. Maar dat komt nog. Nu eerst een beschrijving van de komplete elektronica van de junior-computer. Van wat we noemen de *hardware*. Na het konkretiseren van deze hardware, na de bouw dus rest het (leren) konkretiseren van programma's, van de *software*. Je zou kunnen zeggen: de software is de geest van de computer en de hardware het vlees en bloed.

Zo zit ie in elkaar

Figuur 4 geeft aan hoe de komplete junior-computer elektronisch in elkaar zit; het principeschema zal nu worden besproken. Beginnen we met IC1. Dat is de microprocessor. Kenners kunnen u ervan verzekeren dat het gebruikte type, de 6502 niet de eerste de beste is. Is zeer populair om zijn grote opdrachtenrepertoire (in vaktaal: heeft een krachtige instruktieset) en om een hoop andere, nog te openbaren software-mogelijkheden.

Alles op z'n tijd

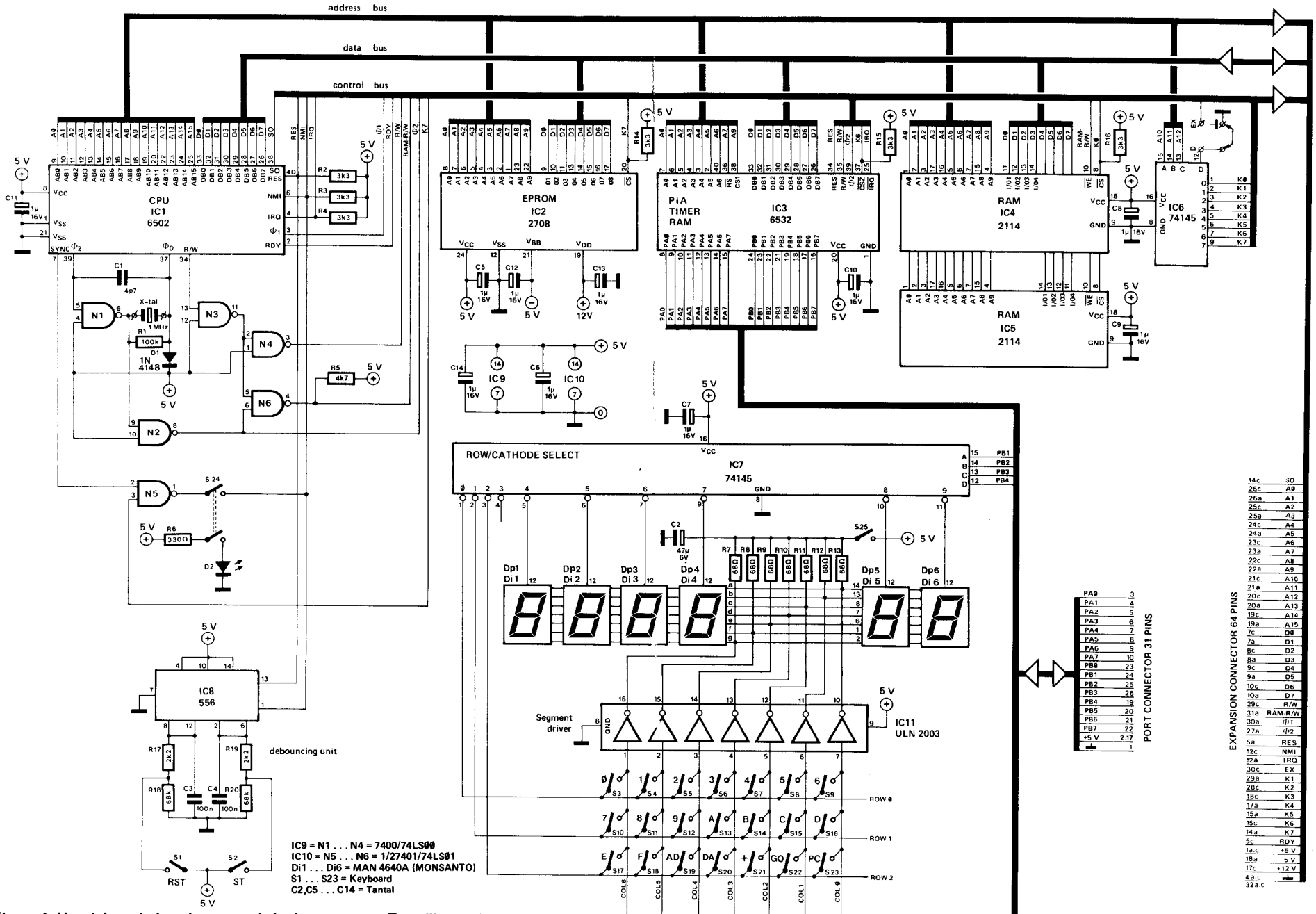
Voor de μP is een gangmaker nodig. Het elektronische veerwerk bestaat uit de klokgenerator met N1, R1, D1, C1 en een 1 MHz-kristal. Samen zorgen ze ervoor dat elke microsekonde een stel elektrische prikkels wordt geleverd: een Φ1 gevolgd door een Φ2, die van belang zijn voor het prepareren van resp. de adresbus en de databus. Dat zijn er nagenoeg een miljoen per sekonde.

Bussen vol enen en nullen

Vanuit de processor IC1 vertrekt de adresbus. Die bestaat uit de lijnen AØ . . . A15. Voor de databus, de lijnen DØ . . . D7 is IC1 vertrekpunt danwel eindpunt. In de elektrische toestand van adresbus en databus zit de kode van het adres resp. de data. Hoe? Wel, stel in gedachten van elke leiding in de volgorde A15 . . . AØ resp. D7 . . . DØ vast of deze spanning voert of niet. Schrijf in dezelfde volgorde van links naar rechts een "woord" van enen en nullen; een 1 als de leiding spanning voert, een 0 als dat niet het geval is. Zo krijg je een woord van zestien letters voor de kode van het adres en een van acht letters voor de kode van de data. In plaats van letters spreken we van bits, in plaats van woorden ook wel van bit-patronen. Dat het "alfabet" slechts uit twee "letters" (0 en 1) bestaat komt omdat met binaire (digitale) logika wordt gewerkt.

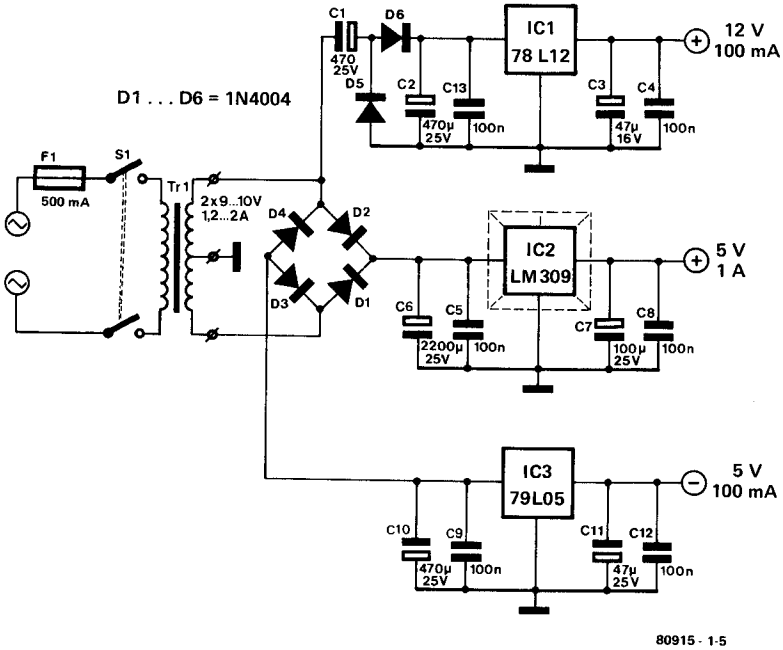
Bij data hebben we het vaak over *byte* in plaats van achtbits-woord of achtbits-patroon. Dat krijgt al meteen betekenis bij de bespreking van het geheugengedeelte van figuur 4. De EPROM IC2 vormt het (vaste) programmageheugen, terwijl de RAM's IC4 en IC5 als (variabel) werkgeheugen dienst doen. Datatransport gaat telkens met acht bits (1 byte) tegelijk. Daarom moet elke geheugenplaats acht bits kunnen bevatten oftewel de elektrische toestand van acht leidingen bevriezen (schrijven) of op een bepaald moment bepalen (lezen). De EPROM biedt plaats aan 1024 bytes. Het gebruikelijke afrondingsjargon spreekt van 1k, waarbij k duizend oftewel kilo betekend. Al die bytes moeten eenduidig kunnen worden opgespoord. Met tien adreslijnen, AØ . . . A9 gaat dat precies want daarmee zijn 2^{10} = 1024 verschillende adressen mogelijk.
De RAM's kunnen elk 1024 halve bytes van vier bits bevatten. Samen zijn ze goed voor 1024 bytes. In IC5 staan op een geheugenplaats met een bepaald adres de eerste, linkse vier bits van het woord (D7 . . . D4), in IC4 de rechtse vier op de plaats met hetzelfde adres.
De geheugens krijgen ook nog het een en ander van de stuurbus. Selektiesignalen bijvoorbeeld. De EPROM en de RAM's hebben dezelfde adresbusaansluitingen: AØ . . . A9. Hoe maakt men onderscheid tussen de twee geheugens? Dat gaat via de signalen $\overline{CS}$, van ChipSelect, die als leidingen K7 voor de EPROM resp. KØ voor de RAM's via de stuurbus afkomstig zijn van IC6, de adresdekoder. Staat er een 1, dus spanning op een $\overline{CS}$-lijn, dan zijn de data-ingangen van het betreffende geheugen van de databus ontkoppeld. Wordt $\overline{CS}$ 0 (volt), dan kan het geheugen aan het dataverkeer deelnemen.
Tien adreslijnen waren gemeenschappelijk voor beide geheugens. Blijven zes adreslijnen over, waarmee 2^6 = 64 verschillende geheugenblokken van 1k zijn te adresseren. Totaal kom je dan op 64k, hetgeen precies klopt als men rekening houdt met de kilo-afrondingsprocedure: 64k hoort bij 2^{16} = 65.536. In Elektuur-notatie: 65K536.
Met 64 chipselectlijnen zijn dus alle geheugenblokken met tien adresaansluitingen te selekteren. In de standaarduitvoering van de junior-computer worden er maar acht gemaakt. Namelijk de signalen KØ . . . K7 van de adresdekoder IC6, die de drie eerstvolgende adreslijnen A10 . . . A12 krijgt toegevoerd. Drie van de acht selektiesignalen worden intern gebruikt. Naast de al besproken twee is dat K6, die de RAM in de PIA, IC3 selekteert. Ter wille van het overzicht geven we nu een soort adresboek:

A15 . . . A13	A12	A11	A10	A9 . . . AØ	aktief:	geheugenblok:	
X	Ø	Ø	Ø	X	KØ	1k RAM (IC4, IC5)	
X	Ø	Ø	1	X	K1	1k extern	
X	Ø	1	Ø	X	K2	1k extern	
X	Ø	1	1	X	K3	1k extern	RAM, ROM
X	1	Ø	Ø	X	K4	1k extern	
X	1	Ø	1	X	K5	1k extern	
X	1	1	Ø	X	K6	RAM in PIA (IC3)	
X	1	1	1	X	K7	1k EPROM (IC2)	



Figuur 4. Het elektronische schema van de junior-computer. Ter wille van de overzichtelijkheid zijn de drie bussen als één leiding getekend.

80915 - 1 4



Figuur 5. De voeding van de junior-computer levert drie gestabiliseerde voedingsspanningen.

De X betekent "dontcare" of gewoon op z'n Hollands "doet er niet toe": de drieletterige "lettergrepen" uit de eerste kolom en de tienletterige van de vijfde kolom mogen bestaan uit willekeurige kombinaties van enen en nullen.
Van de 64k geheugenkapaciteit is dus 8k standaard-dekodeerbaar. Voor een groter dekodeerbereik is uitbreiding van de adresdekoder noodzakelijk. Extern. Het hoe heeft u nog van ons te goed. Overigens is acht "kaa" en zelfs de interne twee "kaa" meer dan voldoende voor een erg serieuze kennismaking. Dààr komt u nog wel achter.
Voor het gebruik van de RAM is nòg een stuursignaal nodig. Een signaal dat onderscheid maakt tussen lezen en schrijven. Een RAM-R/$\overline{W}$-signaal dus, dat 1 is bij lezen en 0 bij schrijven. Het signaal komt uit poort N6 en onstaat uit een kombinatie van het R/$\overline{W}$-signaal van de μP en de Φ2-klokpuls. Hierdoor is gegarandeerd dat geen data wordt geschreven zolang de databus nog niet is gestabiliseerd.

Stuurbusdiensten

Veel stuursignalen zijn al besproken. De rest volgt nu. Met het Resetsignaal RES worden de μP IC1 en de PIA IC3 in een bepaalde startpositie gedwongen. Dat gebeurt op de overgang van 1 naar 0 van de beide IC-aansluitingen RES. Normaal zijn deze punten 1 via de "pullup-" of hoog-

houdweerstand R2. Een reset komt tot stand na indrukken van de toets RST (S1). Een timer van IC8 is tussengeschakeld om de kontaktdender van de toets te temmen.
Ook de NMI-ingang van IC1 reageert op een 1/0-overgang en is in rust 1 onder invloed van R3. Afgezien van een eventuele externe mogelijkheid (via de uitbreidingsconnector) zijn er intern twee mogelijkheden om een NMI (onderbreking) te maken. Dat kan door het indrukken van de toets STOP (S2), die via de andere helft van IC8 een niet zo erg denderend, dus goed signaal levert. Of, indien de STEP-schakelaar S24 in de stand ON staat *en* de uitgang van poort N5 van hoog naar laag gaat. Dit is van belang om programma's stap voor stap te kunnen uitvoeren. Indien de adressen echter betrekking hebben op de EPROM, waarin het monitor-programma zit, moet de onderbrekingsmogelijkheid zijn geblokkeerd. Vandaar de aanwezigheid van K7 aan de ingang van N5. Is deze immers 0, dan kan de uitgang van N5 nooit een sprong van 1 naar 0 maken.
Maak je de aansluiting IRQ van IC1 en IC3 0, dan wordt het programma eveneens onderbroken. Tenminste: zolang deze ingang niet de mond wordt gesnoerd via het programma. Intern is een interruptrequest mogelijk via programmering van de timer binnen de PIA. We spreken dan van een software-interrupt. De NMI- en IRQ-aansluitingen zijn ook beschikbaar op de uitbreidingskonnektor.
Verder bevat de stuurbus de kloksignalen $\Phi 1$ en $\Phi 2$, waarvan de laatste nodig is voor het funktioneren van de PIA. En een R/$\overline{W}$-lijn die niet door $\Phi 2$ wordt beïnvloed, zoals de RAM-R/$\overline{W}$. Dit signaal bepaalt de data-transportrichting, beïnvloedt de toestand van de tweerichtingsbuffers achter de data-aansluitingen binnen IC1 en IC3. Tot slot de intern niet gebruikte signalen RDY en SO, die een rol spelen bij eventuele extern aan te sluiten dynamische RAM's, en de leiding EX (zie IC6), van belang bij de externe uitbreiding van het adresdekodeerbereik.

Het I/O-tussenstation

Via de PIA IC3, het bidirektionele data-doorgeefluik van de computer gaat de databus naar poort A of B. Bij elke poort hoort een I/O-register en een achtbits datarichtingsregister, waarin in de vorm van enen en nullen vastligt welke lijnen tot uitgang zijn verklaard en welke tot ingang. Een 1 levert een uitgang op, een 0 een ingang. De inhoud van het richtings-register wordt bepaald door de software, dus door het programma.
Het signaal R/W bepaalt de richting van het datatransport. Bij een 1 wordt data uit de PIA gelezen, een nul heeft schrijven tot gevolg.
Er is ook nog RAM aan boord, zij het niet zoveel: 128 bytes. Samen met de 1024 bytes in IC4 en 5 levert dit 1152 bytes op. Voor de adressering van de 128 geheugenplaatsen volstaan de zeven adreslijnen AØ . . . A6. Het signaal $\overline{RS}$ (= A7) dient als elektronische wissel: er wordt mee gekozen tussen datatransport van en naar de RAM en idem van en naar een van de poorten. Bij een 0 is de RAM aan bod; een 1 levert I/O-verkeer op. Adres-lijn A7 fungeert als wisselwachter $\overline{RS}$. Bij de RAM zijn twee chipselect-signalen betrokken: het uit de adresdekoder afkomstige signaal K6 dat als $\overline{CS2}$ dienst doet en CS1, waarover adreslijn A9 alles te vertellen heeft.
Bij de keuze tussen de poorten A en B en bij de datarichting zijn de adres-

lijnen AØ, A1 en A2 van belang. Over de PIA valt veel te vertellen. Zoveel dat er een apart hoofdstuk aan is gewijd in boek 2. Waarin ook de bijbehorende software uit de doeken wordt gedaan.

Het buitengebeuren

Toetsenbord en display vormen de ingebouwde buitenwereld van de junior-computer, het "dashbord" van wat we ooit de black box hebben genoemd. De navelstreng bestaat uit vier leidingen van poort B en zeven van poort A, en twee aansluitingen op de stuurbus. Om met de laatste te beginnen: dat zijn de al eerder besproken signalen RST en ST, die ontstaan door menselijk ingrijpen op de betreffende toetsen. Ook schakelaar S24 is al besproken. Daarmee kan worden gekozen tussen normale en stapvoorstap-afwikkeling van het programma.
De nog niet behandelde toetsen vinden we in figuur 4 terug als de schakelaars S3 . . . S23, die zijn opgesteld in een rechthoek (matrix) van drie rijen en zeven kolommen. Zestien van de toetsen zijn er voor het intoetsen van data in hexadecimale vorm (over wat hexadecimaal precies is meer in hoofdstuk 2). Data moeten we hier ruim opvatten; ook adresinformatie hoort erbij. De overige vijf toetsen zijn er voor allerlei besturingsfunkties, die bij het (leren) programmeren in hoofdstuk 3 uitgebreid zullen worden behandeld.
Data naar het display en data van het toetsenbord lopen beide via zeven lijnen van poort A er is dankbaar gebruik gemaakt van het feit dat een poort in twee richtingen kan werken. Via software van het monitorprogramma, dat wordt geaktiveerd door het indrukken van RST, worden de displays Di1 . . . Di6 periodiek bediend; Di1 . . . Di4 geven een adres hexadecimaal weer, Di5 en Di6 de bij het adres horende data, eveneens in hexadecimale vorm. Via diezelfde software worden ook de 21 toetsen periodiek afgevraagd: is er een toets ingedrukt en zoja welke? Het programma herkent de toets en handelt overeenkomstig de toetsfunktie. Het effekt van kontaktdender, dat is het herhaald kontakt-geen kontakt maken van een toets gedurende een tijdje na het indrukken, wordt door het programma genegeerd.
Het periodieke element van het berichtenverkeer schuilt in IC7. Die heeft tien uitgangen, die afhankelijk van de enen en nullen op de vier lijnen PB1 . . . PB4 van poort B om de beurt en periodiek laag (0) zijn. De uitgangen zijn verbonden met de drie rijen van S3 . . . S23 en met de zes gemeenschappelijke katoden van Di1 . . . Di6. Een uitgang van IC7 blijft dus ongebruikt. Is een bepaalde katode op een bepaald moment op 0 (volt) vastgepind omdat de aangesloten uitgang van IC7 aan bod is, dan kan op het display in kwestie iets worden weergegeven door het al dan niet laten oplichten van een of meerdere van de zeven segmenten van het display. Een segment licht op als de erop aangesloten bufferuitgang (IC11) niet laag is. En dat wordt weer bepaald door de enen en nullen op de lijnen PAØ . . . PA6. Die op hun beurt weer worden gedikteerd door de software.
Is een van de drie rijen van S3 . . . S23 aan bod dan herkent het programma welke toets van de rij is ingedrukt. Al met al levert de kommunikatie met toetsenbord en display in de vorm van "ieder op zijn beurt" (*multiplexing*) een elegante metode tot materiaalbesparing op. Minder hardware

dus ten koste van iets meer software. Want bezette geheugenplaatsen zijn goedkoper dan bezette IC-plaatsen.
Rest nog S25, waarmee het display kan worden uitgeschakeld als deze niet nodig is. Bijvoorbeeld wanneer men via de I/O-connector extern met de junior-computer wil kommuniceren. Het doelloze geflikker van Di1 . . . Di6 wordt daardoor de kop ingedrukt.

Eten en drinken

De junior-computer doet het niet zonder *voeding*. Vandaar de noodzaak van figuur 5. Het is een standaardschakeling die zorgt voor drie spanningen, te weten +5 volt voor alle IC's en displays en +12 en −5 volt voor de EPROM IC2. Elk IC is ter plaatse ontkoppeld met een tantaalelko (C5 . . . C13 in figuur 4); voedselvergiftiging is hierdoor uit den boze.

Tot zover de schemabeschrijving van de junior-computer. Waarin is getracht om niet aan de oppervlakte te blijven maar ook niet diep in de details te duiken. Het wordt nu hoog tijd om de soldeerbout aan te zetten.

Aan de slag

De bouw van de junior-computer

Wat moet er zo al gebeuren? De junior-computer zit niet bepaald in een vloek en een zucht in elkaar, maar om nou te zeggen dat het een onmogelijke zaak is gaat ons te ver. Wèl is het belangrijk dat u het nu volgende tekstgedeelte zeer goed leest voordat er ook maar één handeling wordt verricht.
De bouw bestaat uit drie klussen. Eerst moeten de drie printen van komponenten worden voorzien. Misschien moeten zelfs eerst nog zelf de printen worden gemaakt, hoewel u in dat geval wel eens op problemen zou kunnen stuiten. Na het volbouwen van de printen en het maken van een aantal onderlinge verbindingen volgt de testfase. Dat is de tweede klus, die betrekkelijk snel af kan zijn (een zeer kleine kans op pech met de kwaliteit van de onderdelen daargelaten). De derde klus komt neer op een dosis mechanische handvaardigheid: de junior-computer gaat de kast in. Zijn eigen kast wel te verstaan.
En dan bent u toe aan de vierde en allerbelangrijkste klus en dat is de omgang met de junior-computer. Maar dan is de soldeerbout al lang afgekoeld.

De print: het draagvlak van de elektronica

De junior-computer is een wat zo mooi heet "single board computer". Dat wil zeggen dat alle elektronische onderdelen op een print zitten. Alles is "aan boord". Nu zijn er drie printen. Hoe zit dat met dat "single board"? Heel simpel. Een van de drie printen is de voedingsprint; die telt niet mee. Houden we twee printen over. Dat zijn de hoofdprint en een klein printje, de display-print. En de laatste is een klein opsteekprintje. De displays hadden best nog op de hoofdprint gekund. Die dan wel wat groter zou zijn

geworden. De nu gekozen oplossing, waarbij de display-print een hoek maakt van ca. vijfenveertig graden met de hoofdprint is gekozen om praktische redenen; het vergroot de afleesbaarheid van de displays bij het gebruik van de junior-computer. Dus toch "single board".
Voordat we gaan solderen eerst nog iets over de hoofdprint. Dat is een dubbelzijdige print. Dat wil zeggen dat op beide zijden van de print koperbanen zijn aangebracht en in dit geval ook aan beide zijden komponenten. Sommige elektrische verbindingen beginnen als koperbaan op een kant en gaan op een gegeven punt aan de andere kant van de print verder; de doorverbinding van beide kopertrajekten gaat via *doorgemetalliseerde gaten,* gaten waarvan de wanden zijn voorzien van een laagje metaal. (Op het eerste gezicht is een geleidend gat misschien een gek idee, maar op gatengeleiding berust de hele halfgeleidertechniek!)

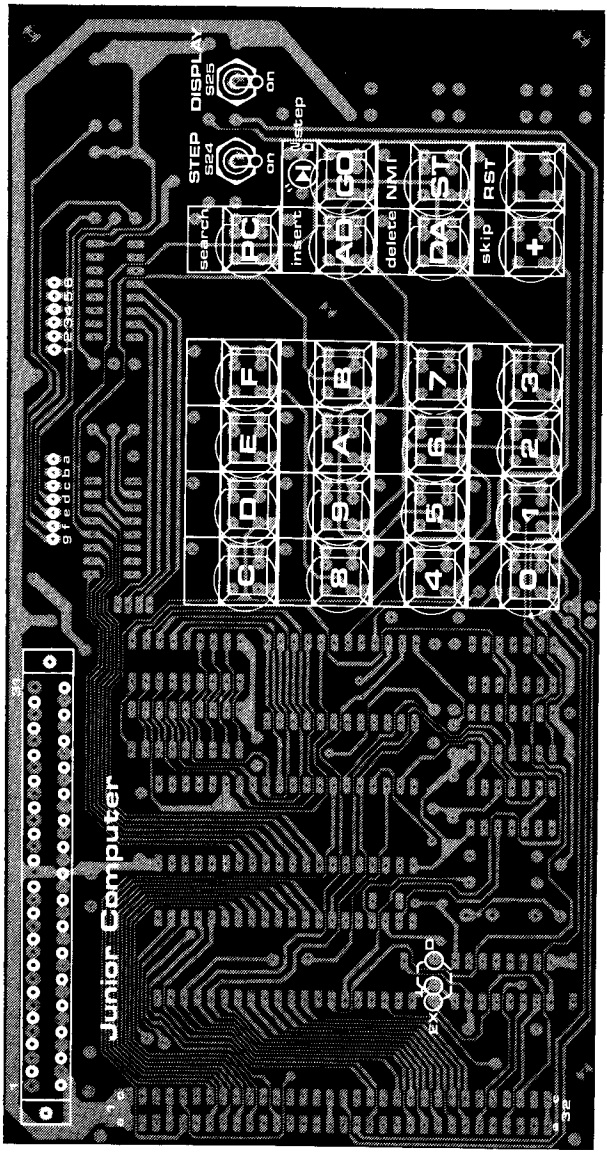
De hoofdprint

Maakt u de hoofdprint zelf[1], dan moeten de doorverbindingen zelf worden gemaakt; dat zijn er een dikke zeshonderd... Maakt u gebruik van de EPS-hoofdprint (EPS 80089-1) dan hoeft dat uiteraard niet te gebeuren. Toch is het zeer verstandig om vóór montage van de onderdelen alle vertikale doorverbindingen te kontroleren. Dat kan met een multimeter (ohmmeting). Of akoestisch, met de goedkope metode van figuur 14a. Een laagspanningsaansluiting van een beltransformator wordt verbonden met een aansluiting van de bel. De overige twee aansluitingen zijn elk verbonden met een "elektrode"; dat kan een star stukje montagedraad zijn. Is de verbinding okay, dan luidt de bel. Deze metode heeft als voordeel dat men een paar honderd keer opkijken naar een al dan niet bewegende meternaald bespaart. Een eventuele ontbrekende verbinding kan worden hersteld via een vertikale draadverbinding (zie figuur 14b) of via de aansluitdraad van een komponent. N.B. Zet men de print vast in een bankschroefje, dan heeft men beide handen vrij.
Van de dubbelzijdige hoofdprint noemen we één bepaalde kant de bovenkant; voor de onderkant blijven er dan niet zo veel mogelijkheden meer over. De onderdelenplattegrond van de bovenkant staat in figuur 6 en die van de onderkant in figuur 7. Het koperbanenpatroon voor de bovenkant staat in figuur 8 en dat voor de onderkant in figuur 9. We zien dat op de bovenkant komen te zitten de 23 toetsen, twee wipschakelaars en (eventueel) de 31-polige poortkonnektor. De onderkant is bestemd voor de montage van de weerstanden, kondensatoren, de IC's en nog een paar andere zaken, plus (eventueel) de 64-polige uitbreidingskonnektor. U ziet het: ten opzichte van het normale gebruik van "komponenten boven" is er hier sprake van de omgekeerde wereld, maar in elektronisch opzicht is er geen bezwaar tegen als de komponenten "hangen" in plaats van "zitten".

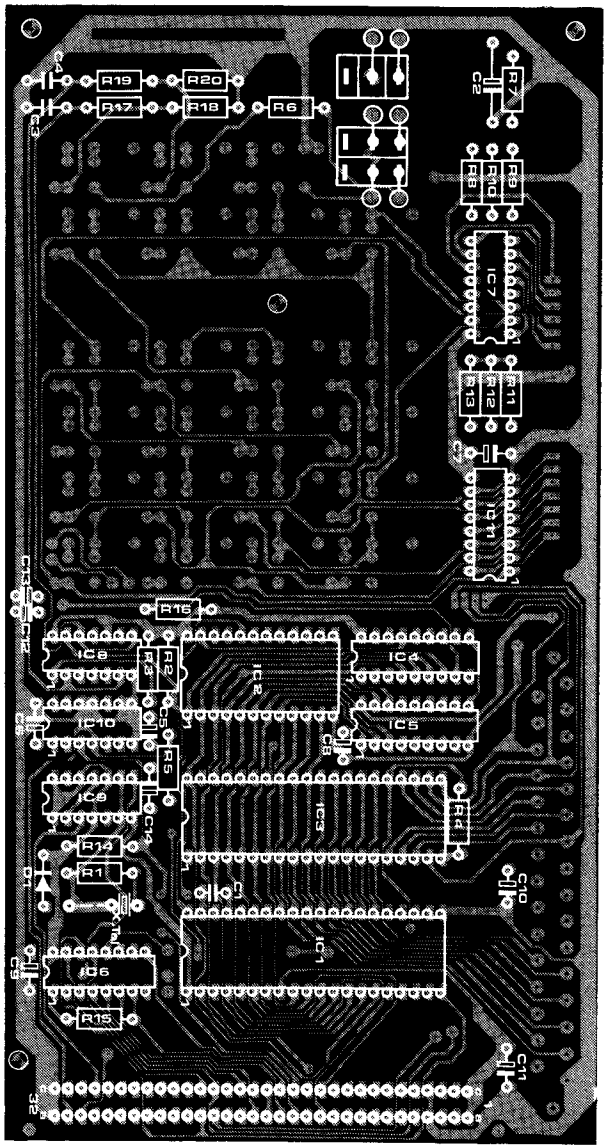
[1] *In verband met de beperkte afmetingen van dit boek is de hoofdprint verkleind afgedrukt. Voor het zelf maken van printen zijn afdrukken op ware grootte onontbeerlijk. Deze kan men vinden in Elektuur, maart 1980, bij het kennismakingsartikel over de junior-computer.*

Hè, hè, eindelijk zijn we dan toe aan het solderen. Gebruik een soldeerbout van 20 à 25 watt met een "potlood"-stift. De komponenten worden in een bepaalde volgorde op de print gemonteerd. En wel als volgt (zie figuur 7 en de onderdelenlijst voor de hoofdprint):

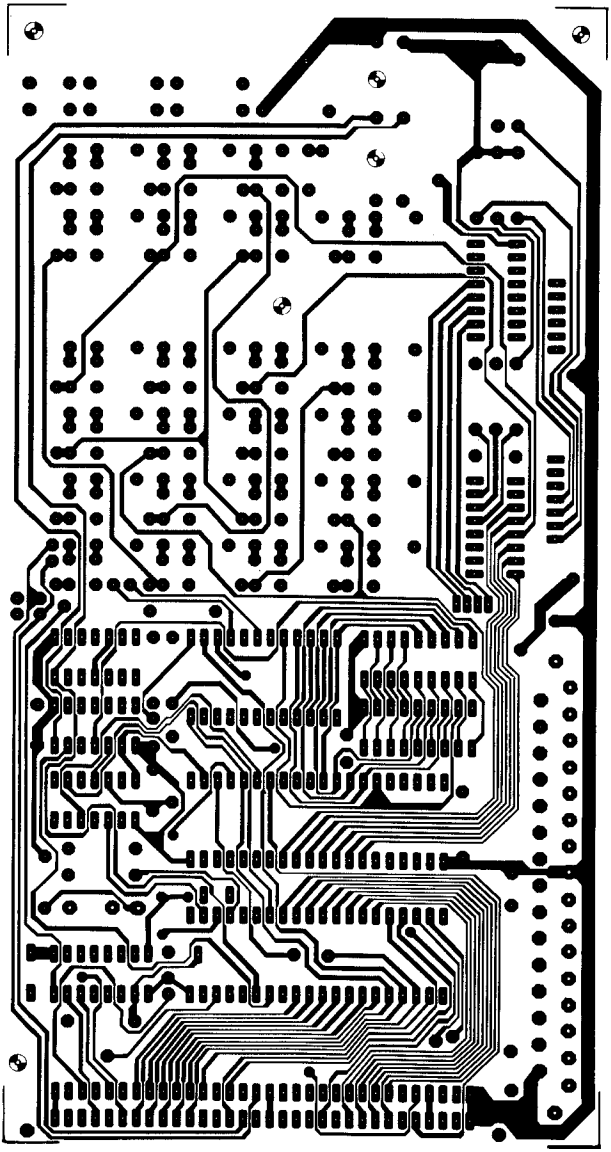
Een. De weerstanden R1 . . . R20 worden op de print gemonteerd. Na het solderen meteen de overtollige draadeindjes zo dicht mogelijk bij de las wegknippen. Dit voorkomt narigheden zoals sluitingen met andere soldeerpunten. Voor degenen die niet dagelijks omgaan met de kleurkode van de weerstanden volgt hier een opgave:
100 k: bruin-zwart-geel-(goud)
3k3: oranje-oranje-rood-(goud)
4k7: geel-violet-rood-(goud)
330 Ω: oranje-oranje-bruin-(goud)
68 Ω: blauw-grijs-zwart-(goud)
2k2: rood-rood-rood-(goud)
68 k: blauw-grijs-oranje-(goud)
De eerste ring bevindt zich daarbij het dichtst bij een uiteinde van de weerstand. De vierde ring geeft de tolerantie aan, dus de mate waarin de werkelijke waarde van de weerstand afwijkt van de nominale, gewenste waarde. Meestal krijgt u te maken met 5%-weerstanden (goud); een doodenkele keer met 1% (bruin) of 2% (rood).
Twee. Bij de montage van de diode D1 moeten we goed letten op de polariteit. Meestal, zeg maar altijd is de katode-aansluiting (de punt van de driehoek) aangegeven door een ring. In geval van twijfel is het laatste woord aan de universeelmeter (ohm-meting uitvoeren met twee aansluitmogelijkheden van de meetpennen; bij de meeste multimeters is de minpen de plus).
Drie. De kondensatoren C1, C3 en C4 worden gemonteerd.
Vier. De elko's C2 en C5 . . . C14 gaan de print op. Kun je normaal gesproken een weerstand of kondensator op twee manieren monteren, hier is het donders goed opletten geblazen. Elko's hebben een plus-aansluiting en een min-aansluiting. Op de komponenten-plattegrond is de plus aangegeven met een niet opgevulde balk (□) en de min-aansluiting met een opgevulde balk (■). Bij tantaalelko's, waarvan de plusaansluiting niet met een + is aangegeven geldt dat de + rechts zit als men de zijde die is voorzien van een gekleurd merkteken voor zich heeft.
Vijf. We zijn toe aan de *montage van de IC's*. Het is gebruikelijk om IC-voetjes op de print te monteren en om dan in die voetjes de IC's te prikken. Dat heeft als voordeel dat men in een wip een IC van de print kan afhalen, maar als nadeel dat er voor elk IC 14, 16, 18, 24 of 40 kontakten bij komen waarvan de betrouwbaarheid een punt *kan* zijn. Hier is de vergulde middenweg gekozen. Drie IC's krijgen een voetje, de overige worden direkt op de print gesoldeerd. Voor IC1 en IC3 is een 40-pens voetje nodig en voor IC2 een 24-pens voetje. Het spreekt vanzelf dat we de voetjes van goede kwaliteit gebruiken.
Nog eens vijf. Een IC kan op twee manieren op de print worden gesoldeerd of in een voetje worden gestoken. *Slechts één manier is de juiste.* Op de onderdelenplattegrond is een IC weergegeven met een rechthoek; een van de korte zijden vertoont een "putje". Een soortgelijk putje of markering van aansluitpen 1 toont het IC. De beide oriëntatiepunten moeten aan de



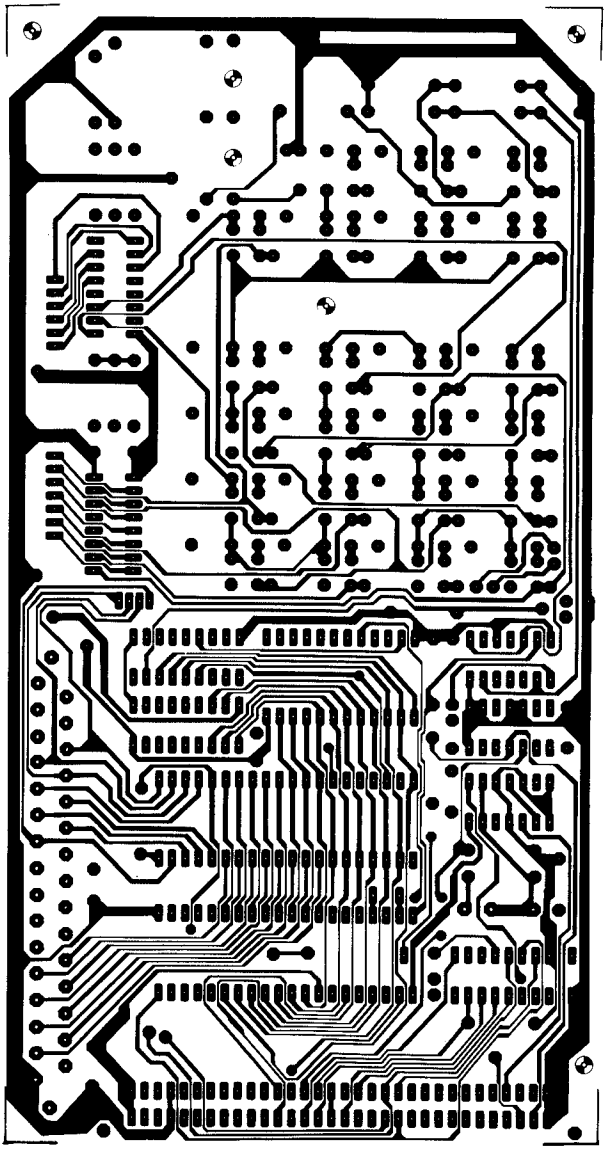
Figuur 6. Onderdelenplattegrond met koperbanenpatroon van de bovenkant van de dubbelzijdige hoofdprint (EPS 80089-1).



Figuur 7. Onderdelenplattegrond met koperbanenpatroon van de onderkant van de hoofdprint. Aan deze kant zitten alle weerstanden, kondensatoren, IC's, een diode en het kristal.



Figuur 8. Het koperbanenpatroon van de bovenkant van de hoofdprint.



Figuur 9. Het koperbanenpatroon van de onderkant van de hoofdprint.

Onderdelenlijst bij de hoofdprint van de junior-computer.

Het principeschema staat in figuur 4, de beide komponenten-layouts in de figuren 6 en 7 en de beide koper-layouts in de figuren 8 en 9.

Weerstanden:

R1 = 100 k
R2,R3,R4,R14,R15,R16 = 3k3
R5 = 4k7
R6 = 330 Ω
R7,R8,R9,R10,R11,R12,
R13 = 68 Ω
R17,R19 = 2k2
R18,R20 = 68 k

Kondensatoren:

C1 = 10 p keramisch
C2 = 47 μ/6 V tantaal
C3,C4 = 100 n MKM of MKH
C5,C6,C7,C8,C9,C10,C11,
C12,C13,C14 = 1 μ/35 V tantaal

Halfgeleiders:

IC1 = 6502 (Rockwell, MOS)
IC2 = 2708
IC3 = 6532 (Rockwell, MOS)
IC4,IC5 = 2114
IC6,IC7 = 74145
IC8 = 556
IC9 = 74LS00, 7400, 74LS132
IC10 = 74LS01, 7401
IC11 = ULN2003 (Sprague)
D1 = 1N4148
D2 = LED rood

Diversen:

X-TAL = kristal 1 MHz
S1 . . . S23 = digitast (Shadow), waarvan één met LED-gat.
S24 = schakelaar, dubbelpolig aan/uit
S25 = schakelaar, enkelpolig aan/uit
1 24-pens IC-voetje (voor IC2)
2 40-pens IC-voetjes (voor IC1 en IC3)
1 konnektor female 64-polig haaks; DIN 41612 (optie) (uitbreidingskonnektor)
1 konnektor male 31-polig; DIN 41617 (optie) (poortkonnektor)

(bij afwezigheid van uitbreidingskonnektor:) minstens vier aansluitpennetjes Ø = 0,8 mm, voor de voedingsaansluitingen
1 print EPS 80089-1

zelfde kant zitten. *Nooit* de door de fabrikant op het IC gestempelde tekst nemen als uitgangspunt voor de oriëntatie!
Overigens: het voordeel van het gebruik van een voetje voor de EPROM IC2 is dat men later gemakkelijk het monitorprogramma dat in IC2 zit kan uitwisselen voor een andere.
Zeven. Het kristal van 1 MHz wordt gemonteerd.
Acht. Is niet verplicht. Namelijk de montage van de 64-polige uitbreidingskonnektor. Heeft men voorlopig nog geen uitbreidingsplannen (wat zeer waarschijnlijk is bij een eerste kennismaking), dan laat men de montage ervan (voorlopig) achterwege. In dat geval moet men echter wel zorgen voor andere voedingsaansluitingen. De voedingsaansluitingen zijn:

+5 volt: pennen 1a of 1c
massa = 0: pennen 4a, 4c, 32a of 32c
–5 volt: pen 18a
+12 volt: pen 17c

Gebruikt men een konnektor, dan zijn bovenstaande aansluitpunten gemakkelijk terug te vinden omdat de nummers bij de aansluitingen staan vermeld. Gebruikt men geen konnektor, dan is men aangewezen op vijf printpennen, die op de betreffende aansluitpunten worden gemonteerd.

Op de print zijn alleen de uiterste nummers 1a/1c en 32a/32c aangegeven. Bij de bepaling van de juiste plaats van een voedingsaansluiting moeten er dus gaatjes worden geteld. Vergissingen daarbij komen de gezondheid van de junior-computer niet ten goede.

Nu zijn alle onderdelen op de onderkant van de hoofdprint gemonteerd. We zijn toe aan de bovenkant.

Negen. De enige draadbrug van de hoofdprint wordt aangebracht, en wel tussen de aangegeven punten ⊥ en D.
Tien. De twee wipschakelaars S24 en S25 worden op de print bevestigd (schakelaar-huis onder). Aan de onderkant komen zes draadjes tussen de schakelaars en de print. Het staat allemaal netjes aangegeven op de onderdelenplattegrond van de onderkant.
Elf. Indien men daar (al) behoefte aan heeft wordt de 31-polige poortkonnektor gemonteerd.
Twaalf. Blijft over de montage van de 23 toetsen en van D2. Deze klus moet zorgvuldig gebeuren omdat ter plaatse van de toetsen een behoorlijke mechanische druk wordt uitgeoefend op de hoofdprint. Verzeker er u voor de montage van een toets van dat er een goed kontakt is tussen het toetshuis en de print. Diode D2 wordt gemonteerd vóór de toets GO: let daarbij goed op de polariteit van de aansluitingen: de katode-aansluiting van D2 zit het dichtst bij de zijkant van de hoofdprint.

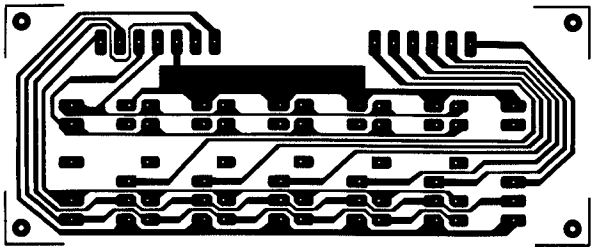
Zo, naar de hoofdprint hebben we voorlopig geen omkijken meer. Of toch niet? Het kan geen kwaad om de print na te kijken op rondslingerend soldeertin, dat kan zorgen voor onbedoelde elektrische verbindingen. Ook de positie van de IC's en elko's nog eens goed nakijken!

De display-print

Daar zijn we eigenlijk gauw mee klaar. De onderdelenplattegrond staat in figuur 10, het koperbanenwerk in figuur 11. Eigenlijk gaat het maar om twee dingen: het monteren van de zes zevensegment-displays Di1 . . . Di6 en het monteren van deze print op de hoofdprint. De displays leveren geen problemen op omdat deze maar op een manier kunnen worden bevestigd, vanwege het asymmetrische aansluitpennenpatroon.
De bevestiging op en elektrische doorverbinding met de hoofdprint gebeurt met zeven draadjes a . . . g en zes draadjes 1 . . . 6; deze aansluitingen ziet men zitten op de hoofdprint, boven de toetsen. Eerst maakt men 13 draadjes van 0,8 mm-montagedraad en van 1½ à 2 cm en soldeert die in de 13 overblijvende gaatjes van de displayprint, aan de koperkant uitstekend; eventuele aan de onderdelenkant uitstekende draadjes knipt men zo kort mogelijk af. Dan volgt de eigenlijke bevestiging op de hoofdprint. De 13 draadjes moeten worden gebogen en gericht en wel zodanig dat ze alle door de aansluitgaten van de hoofdprint steken. De display-print moet een hoek van ca. 45 graden maken met de hoofdprint. Als dat allemaal in orde is volgt het vast solderen van de 13 draadjes, aan de onderkant van de hoofdprint. Overblijvende stukjes draad knippen we uiteraard af.

N.B. Het is niet persé noodzakelijk dat de display-print op de hoofdprint wordt gemonteerd. Komt het bijvoorbeeld door de keuze van een bepaalde kast beter uit dat de displays zich op enige afstand bevinden van de hoofdprint, dan kan de verbinding worden gemaakt met een stuk zogenaamde "flat cable", dat is een platte bundel soepel en onderling geïsoleerd montagedraad in verschillende kleuren. Het spreekt haast vanzelf dat er nu extra moet worden gelet op de juistheid van de diverse aansluitingen.

Figuur 10. Onderdelenplattegrond van de display-print (EPS 80089-2).



Figuur 11. Koperbanenpatroon van de display-print.

Onderdelenlijst bij de display-print van de junior-computer. De komponenten-layout staat in figuur 10 en de koperlayout in figuur 11.

Halfgeleiders:

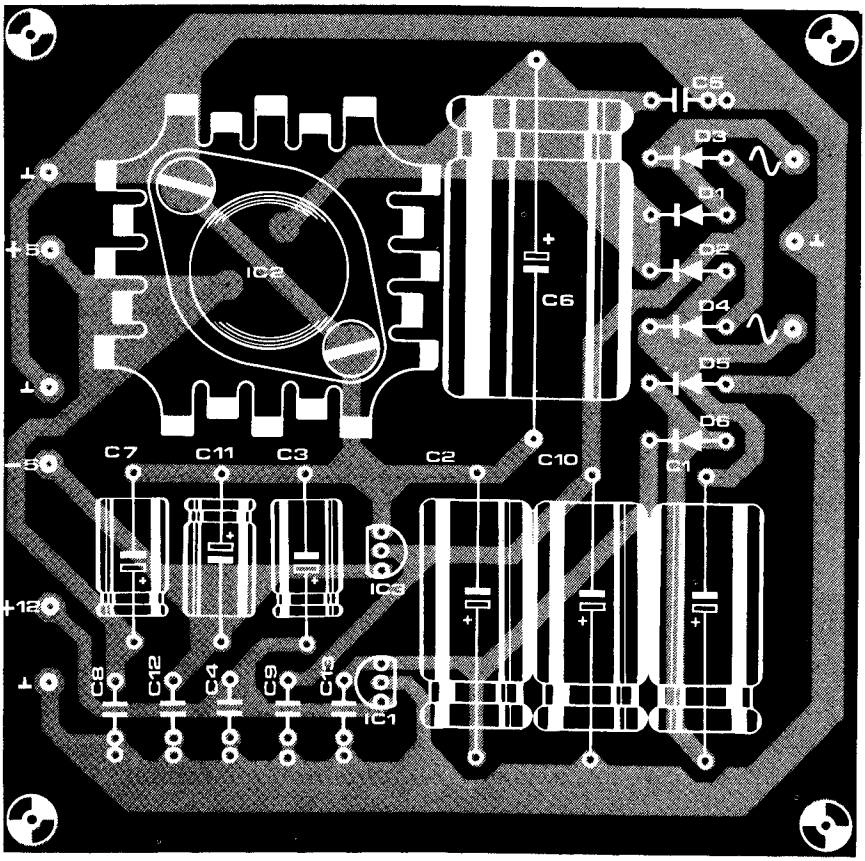
Di1,Di2,Di3,Di4,Di5, Di6 = MAN4640A (gemeenschappelijke katode) (Monsanto)

Diversen:

1 print EPS 80089-2

De voedingsprint

De onderdelenplattegrond van de voedingsprint staat in figuur 12 en de bijbehorende koper-layout in figuur 13. Eigenlijk zijn er bij het volbouwen van deze print geen problemen te verwachten. Wel is het zaak goed te letten op de juiste polariteit van de dioden en de elko's. Bij de montage van de LM309K (IC2) maken we gebruik van een vinger-koelplaat.



Figuur 12. Onderdelenplattegrond van de voedingsprint (EPS 80089-3).

Onderdelenlijst bij de voedingsprint van de junior-computer.

Het principeschema staat in figuur figuur 5; de printgegevens in figuur 12 en figuur 13.

Kondensatoren:

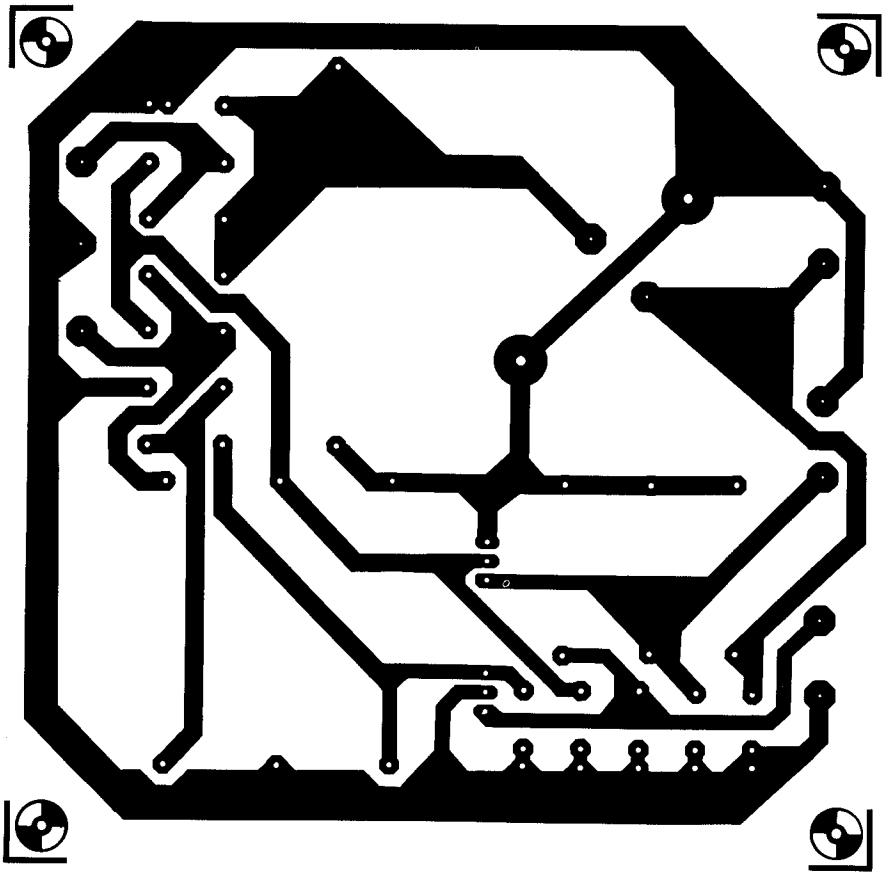
C1,C2,C10 = 470 μ/25 V
C3,C11 = 47 μ/25 V
C4,C5,C8,C9,C12,C13 = 100 n MKM of MKH
C6 = 2200 μ/25 V
C7 = 100 μ/25 V

Halfgeleiders:

IC1 = 78L12ACP (5%)
IC2 = LM309K
IC3 = 79L05ACP (5%)
D1,D2,D3,D4,D5,D6 = 1N4004

Diversen:

Tr1 = transformator 2 x 9 à 10 V/1,2 à 2 A
S1 = schakelaar, dubbelpolig aan/uit
F1 = zekering 500 mA, met zekeringhouder
1 print EPS 80089-3
1 vinger-koelplaat voor IC2



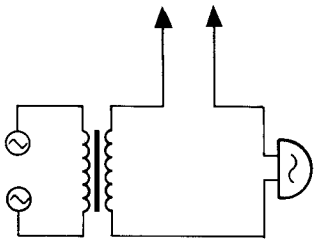
Figuur 13. Koperbanenpatroon van de voedingsprint.

En dan nog . . .

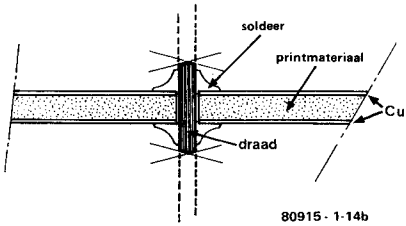
De drie printen zijn nu alle volgebouwd. Rest nog de aansluiting van de voedingstransformator Tr1 op de voedingsprint en, via de netschakelaar S1 en de zekering F1, op het lichtnet (netsteker nog niet in de kontaktdoos steken!) In afwachting van het definitieve inkasten kunnen de verbindingen provisorisch zijn, maar dat betekent niet dat ze niet degelijk zouden moeten zijn. Vooral het checken van de voedingsverbindingen is van levensbelang voor de junior-computer. Ze worden overigens pas gemaakt nadat de voeding op de goede werking is getest.

Zal ie het doen?

Het wordt nu echt spannend. We gaan de junior-computer namelijk testen op de juiste werking.



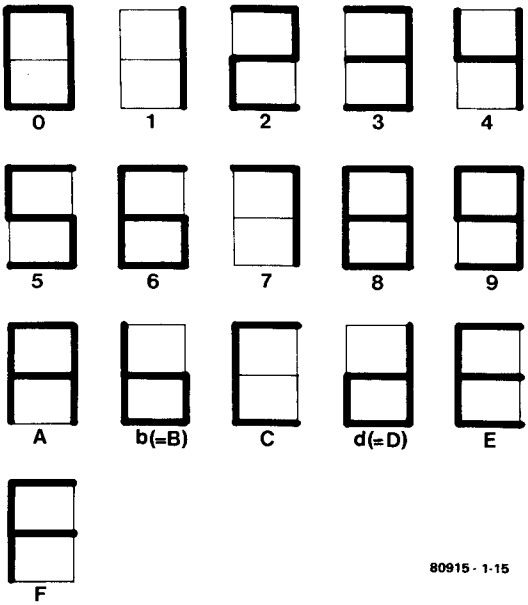
80915 - 1-14a



80915 - 1-14b

Figuur 14. Het testen van elektrische verbindingen hoeft niet persé te gebeuren met een multimeter (ohm-meting); het kan ook akoestisch, onder gebruikmaking van een goedkope bel + beltransformator. Let wel: deze metode mag men alleen toepassen indien de print nog niet is voorzien van onderdelen! In b is aangegeven hoe men in het geval van een zelf gemaakte hoofdprint een doorgemetalliseerd gat kan vervangen door een vertikale draadbrug.

Eerst is het nogmaals grondig kontroleren van alle drie printen een zeer verstandige zaak. Laat er desnoods ook eens een ander naar kijken. Dan gaat de netsteker de muur in. *Let wel: de voeding is nog niet aangesloten op de hoofdprint!* De voeding wordt ingeschakeld met de netschakelaar S1 en we meten met een multimeter (DC-spanningsmeting) de drie voedingsspanningen ten opzichte van massa (= nul), op de aansluitpennen van de voedingsprint. De spanningen moeten binnen ±5% gelijk zijn aan +5, −5 danwel +12 volt. Is dit niet het geval, dan moet de hele voeding nog eens apart worden onderzocht. Maar normaal gesproken kan er bij dit type voeding niets mis gaan.
Is de voeding okay, dan schakelen we hem uit en worden de vier verbindingen tussen de voedingsprint en de hoofdprint aangebracht. En nogmaals: let erop dat de juiste aansluitingen op de hoofdprint worden gebruikt (zie "hoofdprint", punt 8)!
Nadat de hoofdprint met vier draden is gevoed plaatsen we de schakelaar STEP (S24) in de stand "uit" en DISPLAY (S25) in de stand "on". De netspanning wordt weer ingeschakeld en . . .
Het display geeft totale duisternis weer. Geen paniek! Dat hoort namelijk zo. Eerst moet de toets RST worden ingedrukt. Als alles goed is geeft elk display nu een willekeurig hexadecimaal getal weer. Moet u natuurlijk wel weten wat een hexadecimaal getal is. Dat komt allemaal nog, in hoofdstuk 2. Nu is het voldoende om te weten dat de in figuur 15 aangegeven kombinaties van oplichtende segmenten kunnen voorkomen.
Hier komen we op een punt, waar, als we hoofdstuk 3 al hebben gelezen, sprake is van een soort voorwaardelijke spronginstruktie: zit het goed met die oplichtende displays, dan kunt u het nu volgende tekstgedeelte overslaan en bent u toe aan het inkasten en afwerken van de junior-computer. We zien u straks terug. De anderen moeten nog even "nablijven". Maar eigenlijk zijn we van mening dat het nu volgende gedeelte voor niemand nodig is.



Figuur 15. De test op de goede werking van de junior-computer levert als alles goed is na het indrukken van de toets RST zes hexadecimale tekens op het display op. Hier zijn alle mogelijkheden aangegeven.

De zure appel

Aangenomen dat de voeding okay is – wat kan er zo al zijn misgegaan? Er zijn een paar algemene mogelijkheden. Zoals:

- kortsluitingen, onstaan door een overmaat aan soldeertin. Die overmaat hoeft niet altijd meteen in het oog te lopen. Kleine soldeer-pareltjes van minder dan 1 mm doorsnede zijn berucht.
- slechte soldeerverbindingen ("koude lassen"). In geval van twijfel beslist de waarheid: de verdachte soldeerpunten opnieuw verhitten met de soldeerbout. Tin naar smaak toevoegen.
- slechte kontakten tussen het konnektor-"mannetje" (male) en het "vrouwtje", en tussen IC en IC-voetje. Een grondige inspektie van met elkaar kontakt te maken delen is geboden. Eventueel reinigen met alkohol o.i.d. Zorg ervoor dat de beide delen ook mechanisch kontakt met elkaar hebben (aandrukken)
- (bij zelf gemaakte printen) Valse kopersporen (lees: kortsluitingen) als gevolg van onvolledige etsing of haren op de print-layout. Alles kontroleren met een loep en eventuele spontane koperbruggen verwijderen met een mesje; laat daarbij de als zodanig bedoelde verbindingen ongemoeid!
- (eigenlijk al bij herhaling eerder genoemd) Foutief aangesloten dioden en elko's en foutief gemonteerde of in het voetje geplaatste IC's; er

is iets mis gegaan met de voedingsaansluitingen op de hoofdprint.
Tot zover een paar algemene aanknopingspunten. Nu nog een paar speciale tips:

- meet met een multimeter (DC 6-, 10- of 12-volts bereik volle schaal) de spanning tussen de punten 13 en 7 van IC8. Deze moet ca. 5 volt bedragen. Drukt men de toets RST in, dan moet de spanning ca. 0,5 volt zijn. Is dat niet het geval, dan kan dat liggen aan de volgende onderdelen: IC8 (de dubbele timer), R2 (pull-up- oftewel hooghoudweerstand) of aan de toets RST.
- is dat allemaal in orde, meet dan eens met een ohm-meting of pen 12 van IC6 aan massa ligt. Is dat niet het geval, dan kon best wel eens de enige draadbrug op de print verkeerd zitten.
- ook de klokgenerator kan apart op juiste werking worden getest. Immers: zonder hart geen leven. Met een oscilloskoop bekijken we de signalen op de aansluitingen 30a en 27a van de uitbreidingskonnektor. Als het goed is zien we dan kloksignalen Ø1 en Ø2, blokvormige signalen met een herhalingsfrekwentie van 1 MHz en een amplitude van 3 à 5 volt. Met een dubbelstraals-oscilloskoop moet te zien zijn dat de kloksignalen elkaar niet overlappen (niet tegelijkertijd +3 à +5 volt zijn), zoals het een goede tweefasenklok betaamt. Toont het skoopscherm doodse stilte dan staat de klok stil. Mogelijke verdachte komponenten zijn C1, IC9 en D1.

Het lijkt ons sterk als een of meerdere van de genoemde algemene of specifieke aanknopingspunten niet zullen leiden tot de juiste diagnose van de fout of fouten, en het herstel daarvan. En als het wèl sterk is? U kunt dan kontakt opnemen met de redaktie van Elektuur door middel van schriftelijke technische vragen of via de "telefonische hulpdienst", elke maandagmiddag.

De kast in

Eigenlijk valt over de behuizing van de junior-computer niet zo veel te zeggen. Of het moet zijn dat deze echt wel wenselijk is om het dure binnenwerk te beschermen tegen allerlei invloeden van buiten af. Het is wel zaak om de kast "op de groei" te kopen zodat er ruimte is voor toekomstige uitbreidingen. Die zullen komen, wacht maar af.
Er zijn twee hoofdtypen behuizing denkbaar. Populair gezegd: het "sigarenkist-model" en het "lessenaar-model". Monteert men de display-print bij een lessenaar-model tegen de schuine zijde, dan verhoogt dat het bedieningsgemak van de junior-computer.
In de kast komen gaten danwel uitsparingen voor de displays, de toetsenrechthoek, de zekeringhouder, netschakelaar en voor de doorvoer van het netsnoer. De hoofdprint is naast de vier gebruikelijke voorzien van een vijfde gat in de buurt van de toetsen. Dit extra bevestigingspunt (de print wordt gemonteerd onder gebruikmaking van op maat gezaagde afstandsbussen) voorkomt doorbuigen van de print als gevolg van het bespelen van het toetsenbord.
Rest de afwerking. Voorzie de toetsen van duurzame en passende opschriften (figuur 6 geeft hierover de nodige informatie). Dat is belangrijk om de verschillende toetsen uit elkaar te kunnen houden.
Dat was het. Nu begint het pas leuk te worden.

Als u IC2, de EPROM zo uit de winkel krijgt is deze niet zonder meer geschikt voor gebruik in de junior-computer. Eerst moet deze nog worden voorzien van 1024 x 8 enen en nullen, en wel zodanig dat daarmee de kode van het monitorprogramma voor de junior-computer vastligt. De EPROM moet dus nog worden geprogrammeerd. Dat gaat met een EPROM-programmer. Heeft u zo'n apparaat tot uw beschikking (met een zogenaamde 2708-personality module), dan kunt u dit onder gebruikmaking van de hexadecimale "monitor dump", achterin dit boek zelf (laten) doen. Of uw elektronica-handelaar kan het doen. Of wij kunnen het doen. Hoe dat laatste precies in zijn werk gaat (Elektuur Software Service) kunt u voorin dit boek lezen. Er wordt op gewezen dat zonder een goed geprogrammeerde EPROM IC2 de junior-computer niet werkt en dat ook de bedrijfstest (zes oplichtende hexadecimale tekens) nooit lukt, hetgeen tot gevolg zou hebben dat er naar een fout wordt gezocht die er misschien helemaal niet is!!

2

Digitale vingeroefeningen

Digi-taal en digi-rekenen

Aan het feit dat de mens tien vingers bezit hebben we het decimale of tientallige stelsel te danken. Met als gevolg dat we met decimale getallen rekenen en decimale getallen gebruiken voor onze (geheim-)taal van kodes.
De (junior-)computer krijgt ook te maken met rekenkundige bewerkingen en met allerlei gekodeerde gegevens (data). Hij heeft daarbij niet de beschikking over twee handen, maar over twee vingers: hij werkt met het tweetallige of binaire talstelsel. Over dat laatste gaat dit hoofdstuk.

Neem een 1, een 9, een 8 en een 1 en schrijf ze van links naar rechts achter elkaar:

1981

We denken dan meteen aan een getal. Bijvoorbeeld aan een jaartal, of de prijs van een kleuren-tv in guldens.
Maar het kan meer betekenen. Te denken valt aan een (omslachtige) kode van het gegeven "volgend jaar". Of een telefoonnummer. Dat laatste is ook een kode, namelijk een kode die samen met het netnummer de standen van een karrevracht relais en schakelaars vastlegt, zodanig dat meneer Jansen kan opbellen of opgebeld worden.
Stel het telefoonnummer 1981 behoort tot de invoergegevens van een computer. Intern werkt de computer met iets heel anders dan met 1981:

1111Ø111ØØ1

Dàt ziet er wel even anders uit. O ja: die rondjes met schuine strepen erdoor zijn nullen. Het streepje is er om nullen te onderscheiden van hoofdletters O. U zult deze gestreepte nullen nog vaak tegenkomen. Terug naar dat getal van enen en nullen. Het zal duidelijk zijn dat hier niet wordt bedoeld: 11.110.111.101, het "gewone" getal. Eigenlijk opvallend dat er uitsluitend enen en nullen voorkomen. Bij een normaal decimaal getal van 11 cijfers is de kans dat er uitsluitend enen en nullen voorkomen erg klein.

We hebben het even voor u berekend: ruim twee miljoenste procent. Inderdaad niet zo veel.
Laten we die kans eens op nul % stellen. Dus dan bestaat het getal uitsluitend uit enen en nullen. In het voorbeeld van een getal van elf cijfers zijn er $2x2x2x2x2x2x2x2x2x2x2 = 2^{11}$ verschillende mogelijkheden, als we uitgaan van twee mogelijkheden (Ø of 1) per cijfer. Vergelijk dit met een "gewoon" getal van 11 cijfers. Zoals bekend is er voor elk cijfer keuze uit tien mogelijkheden: 0 . . . 9.
Dit levert voor het getal $10x10x10x10x10x10x10x10x10x10x10 = 10^{11}$ verschillende mogelijkheden op.

Hoe zit een getal in elkaar?

Elk getal kun je zien als het resultaat van een optelling van meerdere andere getallen. Er zijn meerdere mogelijkheden. En wel meer naarmate het getal hoger is. Als er nu ook nog het aantal getallen dat, bij elkaar opgeteld het getal in kwestie levert, vrij te kiezen is zal het u duidelijk zijn dat het een onoverzichtelijke toestand gaat worden waarin geen enkel systeem schuilt. Daar moet iets aan worden gedaan. En daar is ook iets aan gedaan, namelijk via het systeem dat bekend staat als *talstelsel*. Er is een talstelsel dat u erg goed kent, hoewel u zich daarvan misschien helemaal niet bewust bent. Dat is het decimale oftewel tientallige stelsel. Een getal is daarbij samengesteld uit eenheden, tientallen, honderdtallen, duizendtallen, enzovoorts. Eenheden en -tientallen zijn daarbij machten van tien. De -tallen kunnen elk voor een bepaald getal 0x, 1x, 2x, 3x, 4x, 5x, 6x, 7x, 8x danwel 9x nodig zijn van alle hoeveelheden -tallen te komen tot het weer te geven getal. Het aantal benodigde -tallen schrijft men van rechts naar links op, in de volgorde van toenemend getal.
Een voorbeeld. Het getal 1981 bestaat uit:

1000 = 1 duizendtallen	$= 1x10^3$	=	1000 x 1	
900 = 9 honderdtallen	$= 9x10^2$	=	100 x 9	
80 = 8 tientallen	$= 8x10^1$	=	10 x 8	
+ 1 = 1 eenheden	$= 1x10^0$	=	1 x 1	
1981				1981

Het getal 1981 kunnen we ook op een andere manier samenstellen:

1024 = 1	1024-tal	$= 1 \times 2^{10}$	=	1024 x 1
512 = 1	512-tal	$= 1 \times 2^9$	=	512 x 1
256 = 1	256-tal	$= 1 \times 2^8$	=	256 x 1
128 = 1	128-tal	$= 1 \times 2^7$	=	128 x 1
0 = geen	64-tal	$= Ø \times 2^6$	=	64 x Ø
32 = 1	32-tal	$= 1 \times 2^5$	=	32 x 1
16 = 1	16-tal	$= 1 \times 2^4$	=	16 x 1
8 = 1	8-tal	$= 1 \times 2^3$	=	8 x 1
4 = 1	4-tal	$= 1 \times 2^2$	=	4 x 1
0 = geen	2-tal	$= Ø \times 2^1$	=	2 x Ø
1 = 1 eenheid		$= 1 \times 2^0$	=	1 x 1
+ 1981				1 = wel een bijdrage van die macht van 2 Ø = geen bijdrage van die macht van 2 11110111101

U ziet: nu geen -tallen die een bepaalde macht van tien zijn, maar -tallen die een bepaalde macht van twee zijn. Bij het tientallige stelsel is tien ook het aantal symbolen dat op de plaats van elk cijfer kan voorkomen (namelijk 0, 1, 2, 3, 4, 5, 6, 7, 8 en 9). U ziet ook dat als we uitgaan van machten van twee, dus van 2,2x2,2x2x2,2x2x2x2, enzovoorts, dat er maar twee symbolen mogelijk zijn voor elk cijfer van het getal, namelijk Ø en 1. En als derde punt: U ziet dat ook nu de aantallen -tallen van rechts naar links zijn opgeschreven met toenemend -tal. Het zal u misschien niet verwonderen dat de herhaalde overgang van tien naar twee bij de genoemde voorbeelden best wel eens te maken kan hebben met de overgang van het tientallige stelsel naar het *binaire of tweetallige stelsel.* Let wel: het gaat nog steeds om hetzelfde getal (1981), maar dan weergegeven met een ander talstelsel.

Waarom eigenlijk?

Men kan zich afvragen of het allemaal zo nodig is, die overgang van het vertrouwde tientallige stelsel naar het tweetallige. Er zijn twee antwoorden mogelijk op deze vraag: ja (1) en nee (Ø). Het antwoord: ja en nee, maar "ja" heeft toch zo zijn voordelen:

Kijken we nog eens goed naar het voorbeeld waarbij het getal 1981 binair is weergegeven als een getal met enen en nullen. We zien dat er wel of geen bijdrage is van een bepaalde macht van twee aan het getal 1981. "Wel" en "geen" hebben veel te maken met "ja" resp. "nee". Een ja levert een 1 voor de betreffende cijferpositie, een nee een Ø. Twee mogelijkheden dus.

De getallen en de als getallen gekodeerde data (gegevens) dienen in de computer op de een of andere manier te worden vastgelegd in een stukje grofstoffelijke elektronica. Bij elk cijfer van een getal hoort een elektronische deelschakeling. Werkt nu de computer intern met het tweetallige stelsel, dan is voor elk cijfer een tweetal toestanden mogelijk. Dus hoeft ook dat stukje elektronica van daarnet slechts twee toestanden te kunnen aannemen. Er zijn vele soorten flipflops voor dit doel beschikbaar. De twee toestanden ofwel logische nivo's zijn:

1 = logisch een: spanning aanwezig, "hoog" en
Ø = logisch nul: spanning afwezig, nul volt, "laag".

(N.B. Het gaat hier om zogenaamde "positieve logica". Van de nauwelijks in zwang zijnde negatieve logica is sprake, als men de 1 en dus de Ø tegenovergesteld defineert)

Het voordeel van intern tweetallig werken zit hem in het feit dat de elektronica zeer eenvoudig is: het is veel eenvoudiger om vier flipflops te maken met elk twee mogelijke toestanden (totaal 16 mogelijkheden), dan een element met tien verschillende toestanden, wat nodig zou zijn voor de weergave van een decimaal cijfer. Het kàn wel, maar ligt niet erg voor de hand.

Er is nog een ander groot voordeel van binair werken. Later in dit hoofdstuk komt de "flowchart" (stroomdiagram) ter sprake. Daarbij gaat het vaak om het nemen van beslissingen. Zo in de trant van: als er dit en dit aan de hand is moet dat en dat gebeuren. Stel er is op een bepaald moment sprake van een beslissing, waarbij afhankelijk van bepaalde voorwaarden een keuze moet worden gemaakt uit tien mogelijke akties. Het is veel handiger om een tiensprong te vervangen door tien tweesprongen; dus om

een gekompliceerde keuze uit tien te vervangen door tien achtereenvolgende ja/nee-beslissingen, waarbij alle tien mogelijkheden worden gecheckt.
Waar het om gaat is dat een gekompliceerd iets is vervangen door een aantal ongekompliceerde "ietsen". Dat geldt niet alleen voor die beslissingen van daarnet; het geldt net zo goed voor het getallenvoorbeeld: het getal 1981 vergt decimaal vier cijfers, het daaraan gelijk getal 1111Ø111Ø1 elf cijfers, welke laatste echter wél minder gekompliceerd zijn (twee in plaats van tien mogelijkheden).
Juist in de keuze voor een grotere lengte in plaats van breedte, die het gevolg is van de keuze voor het binaire talstelsel en in feite ook de keuze voor de ja/nee-logica[1], schuilt het enorme vermogen van de computer om moeilijke problemen of handelingen te vertalen in een min of meer groot aantal doodsimpele problemen of handelingen.

Over bits en bytes en zo

Het is hoogst ongebruikelijk om bij het binaire stelsel en dus eigenlijk bij het hele computer-gebeuren te spreken van de cijfers van een getal. Een getal bestaat hier niet uit cijfers maar uit *bits*. Een bit is de amerikaanse afkorting van "binary digit". Een bit kan dus twee waarden aannemen: Ø of 1 en heeft wat men noemt de laagste informatie-inhoud. De bit is de eenheid van informatie; lagere informatie dan een bit bestaat niet.
Bits komen vaak in groepen voor. Of, anders gezegd: er bestaat zoiets als bitpatronen. Denk daarbij aan een binair getal of aan de een of andere kode (waarover later meer). Nu spreekt men vaak niet van bitpatronen, maar van *woorden*. Net zoals de woorden van deze tekst bestaan uit letters bestaan woorden in de hier bedoelde betekenis uit bits.
Voor woorden met een bepaald aantal bits bestaat weer een aparte kreet. Zo is het gebruikelijk om woorden van acht bits lang *bytes* te noemen (In andere systemen bestaat een byte soms uit 16 bits).
Woorden van vier bits heten ook wel *nibbles*.
U herinnert zich ongetwijfeld nog van hoofdstuk 1 dat de databus van de junior-computer 8 bits breed is. Dit betekent dat de databus op elk moment is weer te geven met een byte. Voor de 16-bits adresbus zijn twee bytes nodig.
Bits en de bijbehorende bytes hebben niet uitsluitend betrekking op de binaire weergave van een getal. Ze hebben ook betrekking op allerlei andere *digitale kodes*. Daarbij zijn er nogal wat mogelijkheden. Bijvoorbeeld:

- de kode van een bepaalde soort opdracht (instruktie), die de computer (op ons verzoek) kan uitvoeren. Deze zogenaamde *opcodes* liggen voor een bepaald type microprocessor (in ons geval de 6502) vast. We leren ze kennen in hoofdstuk 3
- de vier-bits kode waarmee de decimale cijfers 0 . . . 9 worden vastgelegd. Over deze *BCD-kode* spreken we later nog.

[1] *Het is niet de bedoeling om in dit boek volledig in te gaan op de ja/nee-logica en de daaraan verwante zogenaamde Boole-algebra. Temeer daar er een andere Elektuur-uitgave, het Digiboek, deel 1 voor een groot deel aan is gewijd.*

- de kode van een adres, ofwel de *adreskode*. Dit is in feite een 16-bits getal (twee bytes lang) dat aangeeft welke geheugenplaats (gerekend vanaf plaats nummer nul) op een bepaald moment betrokken is bij de uitvoering van een instruktie.
- dan heb je de zogenaamde ASCII-kode van zeven bits voor het koderen van *tekens* oftewel *"characters"*. Daaronder wordt verstaan hoofdletters en kleine letters, decimale cijfers, leestekens en nog een aantal huishoudelijke tekens, die met name van het telex-gebeuren afkomstig zijn.
- kodes voor andere gegevens (data).

Bij al deze digitale kodes gaat het om een aantal bits, die elk de waarde Ø of 1 kunnen aannemen.

Hexadecimaal: bits in steno

Goed, men is zo leuk geweest om de oude vertrouwde decimale cijfers te vervangen door veel grotere reeksen die bestaan uit enen en nog eens nullen. Is dat nou allemaal wel zo praktisch voor degene die met de computer moet omgaan? Okay, dat het handiger is voor de computer zal best zo zijn, maar je moet je wel eens verplaatsen in de gedachten van de computer en reeksen van enen en nullen opschrijven. Alleen al die adresbus met z'n 16 bits. Dat is toch ondoenlijk; een vergissing is zo gemaakt. Moet dat nou echt?

Nee.

Omdat we gebruik kunnen maken van de *hexadecimale notatie* van een binair getal of een andere digitale kode.

Hexadecimaal betekent zestientallig. Wat we in feite gaan doen is een binair woord weergeven via de notatie in het hexadecimale stelsel. We hebben gezien dat de overgang van het tientallige naar het binaire stelsel gepaard gaat met een sterke vergroting van het aantal cijfers (bits). Op dezelfde manier kan men verwachten dat de overgang van het tientallige naar het zestientallige stelsel gepaard zal gaan met een verkleining van het aantal cijfers. Deze twee gegevens kombinerend moet het duidelijk zijn dat de overgang van tweetallig naar zestientallig moet leiden tot een flinke reduktie van het aantal cijfers dat nodig is om een binair getal of digitale kode weer te geven. We zullen zien dat de reduktie in aantal cijfers maximaal een faktor 4 bedraagt.

Voor het tientallige stelsel zijn tien basis-symbolen nodig, namelijk de cijfers 0 . . . 9; het tweetallige stelsel heeft er twee nodig, waarbij is gekozen voor Ø en 1. Bij elk lager-dan-tientallig stelsel zijn er voldoende basis-symbolen voorhanden; je neemt gewoon het aantal benodigde cijfers vanaf nul. Voor het hexadecimale stelsel kunnen we echter niet zeggen: neem de getallen 0 . . . 15, omdat elke cijferpositie van een getal moet worden weergegeven door één symbool. Daarom is bij de hexadecimale weergave van een getal gekozen voor de cijfers 0 . . . 9, aangevuld met de hoofdletters A, B, C, D, E en F. De betekenis van de cijfers en letters is als volgt:

0=0=ØØØØ	4=4=Ø1ØØ	8= 8=1ØØØ	C=12=11ØØ
1=1=ØØØ1	5=5=Ø1Ø1	9= 9=1ØØ1	D=13=11Ø1
2=2=ØØ1Ø	6=6=Ø11Ø	A=10=1Ø1Ø	E=14=111Ø
3=3=ØØ11	7=7=Ø111	B=11=1Ø11	F=15=1111

Hoe gaat nu de overgang van binaire notatie (nullen en enen) naar de hexadecimale notatie precies in zijn werk? Heel eenvoudig. Het binaire getal wordt van rechts naar links opgedeeld in groepjes van vier bits. Links aangekomen blijven wellicht een, twee of drie bits over; dit aantal wordt met nullen aangevuld tot vier. Vervolgens vervangt men elke nibble van vier bits door het overeenkomende hexadecimale cijfer en plakt deze laatste keer aanelkaar. Bijvoorbeeld:

$$(\underbrace{\emptyset)11}_{6}\underbrace{\emptyset1}_{A}\underbrace{\emptyset1}_{B}\underbrace{\emptyset1}_{C}11\underbrace{\emptyset\emptyset}=6ABC_{16}$$

De index 16 achter 6ABC duidt erop dat het om de hexadecimale notatie gaat. Dat is alleen dan gebruikelijk als er meerdere getalstelsels in het geding zijn; meestal laat men de index weg.
Om misverstanden te voorkomen: de hexadecimale notatie geldt ook voor bitpatronen (digitale kodes) die niet als binair getal zijn bedoeld. En nog een ding: het gaat hier niet zo maar om een soort stenografische notatie, maar om een volwaardige notatie in het zestientallige stelsel. Met andere woorden: het getal 6ABC is gelijk aan:
$Cx16^0 + Bx16^1 + Ax16^2 + 6x16^3 = 12x1+11x16+10x256+6x4096$=decimaal 27.324.
U kunt dit kontroleren door het binaire getal decimaal uit te schrijven.
Dat het allemaal goed uitkomt met dat indelen in groepen van vier bits heeft te maken met het feit dat de *basis* van het nieuwe talstelsel (16) een bepaalde macht is van de basis (2) van het oorspronkelijke talstelsel. Bij de overgang van binair naar achttallig (oktaal) gaan we uit van groepen van drie bits en bij de overgang van binair naar 32-tallig van groepen van vijf bits.
Figuur 1 geeft een overzicht van de drie talstelsels die voor ons verhaal van belang zijn. Figuur 2 geeft een van de metoden aan om een decimaal getal om te zetten in een binair getal. Het getal wordt herhaald gedeeld door twee. Deelt men een oneven getal door twee, dan levert dit een 1 op; een even getal geeft een Ø. Het is mogelijk om decimale getallen direkt om te zetten in hexadecimale, maar het is handiger om eerst de binaire versie te maken en vervolgens via afsplitsing van groepen van vier bits hexadecimaal te gaan werken.

Binair rekenen

In wezen bestaat er een grote overeenkomst tussen het vertrouwde decimale rekenen en binair rekenen. Dat zullen we zien bij de vier rekenkundige bewerkingen optellen, aftrekken, vermenigvuldigen en delen.

Binair optellen

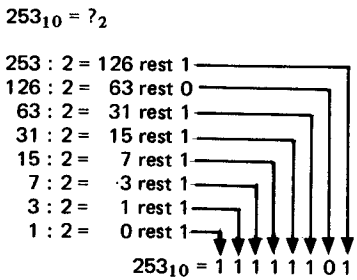
Bij decimaal optellen ontstaat een overdracht naar de volgende plaats, eentje verderop links, zodra het bedrag 10 is overschreden. Een voorbeeld:

```
  129   eerste getal
  243   tweede getal
    1   overdracht
+ ---
  372   resultaat = de som
```

In het binaire geval ontstaat zo'n overdracht zodra het bedrag 2 is over-

binair	decimaal	hexa-decimaal	nibble
Ø	Ø	Ø	ØØØØ
1	1	1	ØØØ1
1Ø	2	2	ØØ1Ø
11	3	3	ØØ11
1ØØ	4	4	Ø1ØØ
1Ø1	5	5	Ø1Ø1
11Ø	6	6	Ø11Ø
111	7	7	Ø111
1ØØØ	8	8	1ØØØ
1ØØ1	9	9	1ØØ1
1Ø1Ø	1Ø	A	1Ø1Ø
1Ø11	11	B	1Ø11
11ØØ	12	C	11ØØ
11Ø1	13	D	11Ø1
1110	14	E	1110
1111	15	F	1111
1ØØØØ	16	1Ø	
1ØØØ1	17	11	
.	.	.	
.	.	.	
.	.	.	
.	.	.	

Figuur 1. Overzicht van het binaire, decimale en hexadecimale talstelsel. Bij de binaire getallen zijn alle "overbodige" nullen, alle nullen links van de meest linkse 1, weggelaten. Immers, dat is ook bij decimale getallen gebruikelijk: we schrijven "9" in plaats van "09".



Figuur 2. De omzetting van een decimaal getal in het bijbehorende binaire getal kan ondermeer gebeuren door herhaald delen door 2. Is het door twee te delen getal oneven, dan noteert men een 1, anders een Ø.

schreden. Ook hier een voorbeeld:

```
  1Ø11Ø1    eerste getal
   1Ø1Ø1    tweede getal
1111 1      overdracht
+ ------
1ØØØØ1Ø     resultaat = de som
```

Het is gebruikelijk om bij binaire optelling niet te spreken van overdracht, maar van *carry*. In feite hoeft men voor het optellen van binaire getallen slechts de volgende regels te kennen:

1+Ø=1
Ø+1=1
1+1=Ø, met carry, dus Ø1+Ø1=1Ø
Ø+Ø=Ø

Deze regels kan men verifiëren door ze toe te passen op figuur 1; ieder volgend getal ontstaat uit het vorige door er 1 bij op te tellen.

Binair aftrekken

Bij decimaal aftrekken treedt het verschijnsel "eentje lenen" op zodra het verschil tussen twee cijfers negatief is. We illustreren dit met het volgende voorbeeld:

```
  1984   aftrektal
   199   aftrekker
    11   geleend
- ----
  1785   resultaat = het verschil
```

Binair gaat het zo:

```
  11ØØØØØ1   aftrektal
   1111111Ø  aftrekker
  111111     geleend
- --------
  Ø1ØØØØ11   verschil
```

We spreken nu niet van "eentje lenen", maar van *borrow*. Ook nu zijn er vier regeltjes waar alles om draait:

Ø–Ø=Ø
1–1=Ø
1–Ø=1
Ø–1=1 met borrow, dus 1Ø–Ø1=Ø1

Dus een carry leidt bij optellen ertoe dat er een 1 extra wordt opgeteld bij de bits, een plaatsje linkser; een borrow leidt ertoe dat er een 1 extra wordt afgetrokken van het verschil van de twee bits, een plaatsje linkser. Ook de vier aftrekregels kan men uitstekend verifiëren door ze toe te passen op figuur 1: ieder vorig getal ontstaat uit het volgend getal door er 1 van af te trekken.

Binair vermenigvuldigen

Laten we eens twee decimale getallen met elkaar vermenigvuldigen. Bijvoorbeeld de getallen 233 en 147:

```
   233
   147
  ---- x
  1631
  932
 233
+ ----
 34251
```

Eerst is het resultaat van 7x233 genoteerd; vervolgens het resultaat van 4x233, maar dan in zijn geheel een plaats naar links verschoven, zodat het in wezen gaat om 40x233. Tenslotte het resultaat van 1x233, twee plaatsen naar links verschoven; in feite dus 100x233.
Het vermenigvuldigen van binaire getallen gaat precies zo. Zelfs nog eenvoudiger, omdat slechts vermenigvuldigen met 1 of Ø voorkomt. Een voorbeeld, waarbij de getallen 1Ø11 en 1Ø1Ø met elkaar worden vermenigvuldigd:

```
   1Ø11      (kontrole:=11)
   1Ø1Ø      (kontrole:=10)
 ------- x
   ØØØØ
  1Ø11
 ØØØØ
1Ø11
 1          carry
+ -------
1ØØ111Ø     (kontrole:=110)
```

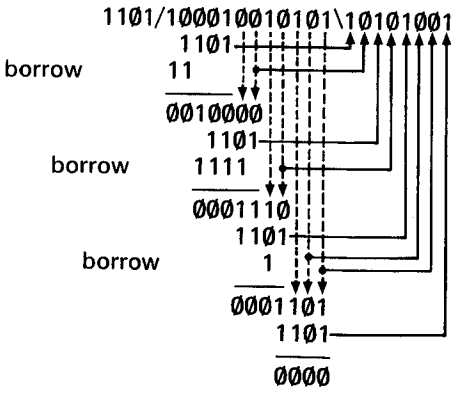
Moet men hexadecimale getallen met elkaar vermenigvuldigen (of bijelkaar optellen of van elkaar aftrekken), dan kan dit door ze eerst binair uit te schrijven, dus door de hexadecimale cijfers te vervangen door de bijbehorende groepen van vier bits. Het resultaat zet men weer om in een hexadecimaal getal.

Binair delen

Eerst maar weer een vertrouwd decimaal voorbeeld: de deling van 2091 door 17:

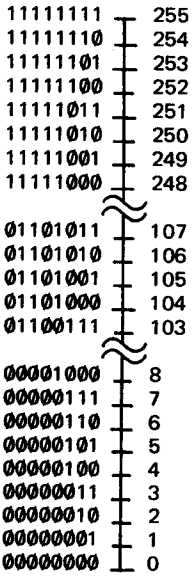
```
17/2091\123
   17|||
 - --|||
    39|
    34|
 - --|
     51
     51
 -   --
      0
```

Binair gaat het net zo. Simpeler zelfs, omdat het er slechts op neer komt dat wordt nagegaan of het verschil tussen de rest en de deler (= het getal waardoor wordt gedeeld) positief of negatief is. Is de rest groter dan de deler, dan noteert men een 1; er ontstaat een nieuwe rest uit de aftrekking van de deler van de oude rest en de nieuwe rest krijgt er rechts een bit bij, dat afkomstig is van het deeltal (= het getal dat wordt gedeeld).
Is daarentegen de rest kleiner dan de deler, dan noteert men een nul en plakt er aan de rechter kant het volgende bit van het deeltal aan.
We geven nu een voorbeeld.
Het deeltal is 1ØØØ1ØØ1Ø1, decimaal 2197
De deler is 11Ø1, decimaal 13
Het resultaat is 1Ø1Ø1ØØ1, decimaal 169



Negatieve getallen

Tot nu toe zijn we er eigenlijk stilzwijgend van uitgegaan dat het om positieve binaire getallen gaat. Net zo als we decimale negatieve getallen kennen bestaan er negatieve binaire getallen. Er zijn verschillende metoden om negatieve getallen weer te geven. We beperken ons tot de bespreking van de *twee-komplement-metode (two's complement).* Maar eerst even iets anders.



Figuur 3. Getallen-as (ook wel getallenstraal genoemd) van alle positieve achtbits binaire getallen.

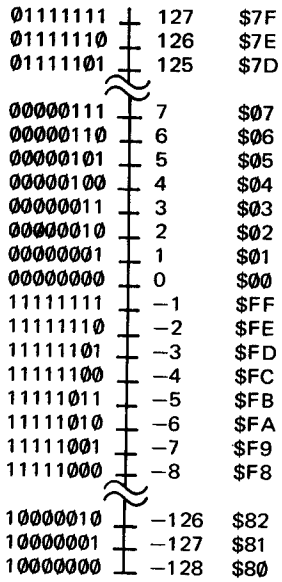
De junior-computer, dat wil zeggen de 6502-microprocessor heeft een woordlengte van 1 byte=8 bits. Zonder extra maatregelen kunnen dus getallen tussen 00000000 en 11111111 (hexadecimaal:00 . . . FF; decimaal:0 . . . 255) worden weergegeven. Voor grotere getallen moet men meerdere bytes aan elkaar plakken.
In figuur 3 zijn de getallen 00000000 . . . 11111111 weergegeven op een zogenaamde getallen-as. Denk daarbij aan een maatlat die is verdeeld in 256 lengte-eenheden. De afstand tussen twee opeenvolgende getallen is daarbij konstant, namelijk één lengte-eenheid. Het optellen van getallen komt overeen met het optellen van twee afstanden. De totale afstand kan daarbij de 256 lengte-eenheden overschrijden. De binaire som kan echter niet groter zijn dan 11111111, omdat er geen negende bit beschikbaar is. We zullen overigens in hoofdstuk 3 zien dat de overdracht naar dat negende bit dat er niet is toch wordt geregistreerd, namelijk via de zogenaamde carry-vlag. Zodat we via deze vlag een links aansluitend tweede byte 00000001 kunnen maken.
Stel we maken maar gebruik van 7 van de 8 bits van een byte. Dat betekent dat nu een getallenbereik mogelijk is van 00000000 tot en met 01111111, dus van nul tot en met 127. Dat zijn bij elkaar 128 van de 256 mogelijkheden van een achtbits-getal. Neem nu het getal nul en trek daar 1 van af:

```
          00000000
          00000001
borrow    11111111
          --------  -
          11111111      =nul min een = min een =-1
          00000001        (nog eens 1 aftrekken)
borrow    --------  -
          11111110      =-2
          00000001        (nog eens 1 aftrekken)
borrow          1
          --------  -
          11111101      =-3
```

Zo doorgaand ontstaat het negatieve deel van de getallen-as van figuur 4, met alle gehele getallen van +127 t/m −128. De negatieve getallen zijn weergegeven volgens de twee-komplement-metode, c.q. -notatie. Het is niet de bedoeling om u te vermoeien met allerlei getalteoriën die er achter zitten. We geven slechts het recept om snel de binaire weergave van een negatief getal te vinden.

1. Eerst bepalen we van het negatieve getal -a de positieve waarde a; binair.
2. Dan vervangen we de nullen in a door enen en de enen door nullen.
3. Nadat hierbij 1 is opgeteld hebben we de binaire weergave in twee-komplement van -a.

Een voorbeeld. Plus drie is gelijk aan 00000011. Enen en nullen verwisselen geeft: 11111100. Daarbij optellen: 00000001, en men krijgt 11111101, hetgeen −3 is (zie ook figuur 4).
Ten koste van één bit oftewel onder opoffering van het maximale aantal verschillende *positieve* mogelijkheden bij een bepaalde bitlengte is er voor gezorgd dat ook negatieve getallen kunnen worden weergegeven. De negatieve getallen beginnen altijd met een 1 (het meest linkse bit); positieve getallen met *dezelfde* woordlengte altijd met een nul, die men net zo goed



Figuur 4. Getallen-as voor alle achtbits positieve en negatieve getallen volgens de twee-komplement-metode. Met de bijbehorende hexadecimale notatie (dollarteken).

kan weglaten. In figuur 5 is in tabelvorm een aantal negatieve binaire getallen opgegeven met de bijbehorende hexadecimale kode.

BCD-kode

BCD is de afkorting van Binary Coded Decimal, hetgeen zoveel betekent als binair gekodeerde decimale getallen.
Het gaat hier *niet* om een getalstelsel, maar om een "gewone" kode. Elk cijfer van het decimale getal wordt vertaald in een groep van vier bits. Die groep van bits stelt de binaire weergave voor van het cijfer. Dus:

0=0000
1=0001
2=0010
3=0011
4=0100
5=0101
6=0110
7=0111
8=1000
9=1001

U ziet, het zijn dezelfde kodes als die voor de eerste negen cijfers van het hexadecimale talstelsel. De BCD-kode van een getal ontstaat nu door de groepen van vier bits aan elkaar te plakken in de volgorde van de cijfers van

binair	decimaal	hexa-decimaal
11111111	− 1	FF
1111111Ø	− 2	FE
111111Ø1	− 3	FD
111111ØØ	− 4	FC
11111Ø11	− 5	FB
11111Ø1Ø	− 6	FA
11111ØØ1	− 7	F9
11111ØØØ	− 8	F8
1111Ø111	− 9	F7
1111Ø11Ø	−1Ø	F6
1111Ø1Ø1	−11	F5
1111Ø1ØØ	−12	F4
1111ØØ11	−13	F3
1111ØØ1Ø	−14	F2
1111ØØØ1	−15	F1
1111ØØØØ	−16	FØ
111Ø1111	−17	EF
111Ø111Ø	−18	EE
111Ø11Ø1	−19	ED
111Ø11ØØ	−2Ø	EC
111Ø1Ø11	−21	EB
111Ø1Ø1Ø	−22	EA
111Ø1ØØ1	−23	E9
111Ø1ØØØ	−24	E8
111ØØ111	−25	E7
111ØØ11Ø	−26	E6
111ØØ1Ø1	−27	E5
111ØØ1ØØ	−28	E4
111ØØØ11	−29	E3
111ØØØ1Ø	−3Ø	E2
111ØØØØ1	−31	E1
111ØØØØØ	−32	EØ
11Ø11111	−33	DF
11Ø1111Ø	−34	DE
11Ø111Ø1	−35	DD
11Ø111ØØ	−36	DC
11Ø11Ø11	−37	DB
11Ø11Ø1Ø	−38	DA
11Ø11ØØ1	−39	D9
11Ø11ØØØ	−4Ø	D8
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.

Figuur 5. Overzichtstabel van negatieve binaire getallen, met de bijbehorende decimale en hexadecimale waarden.

het getal. Een voorbeeld. De BCD-kode van het getal 1981 is:

0001100110000001

1 9 8 1

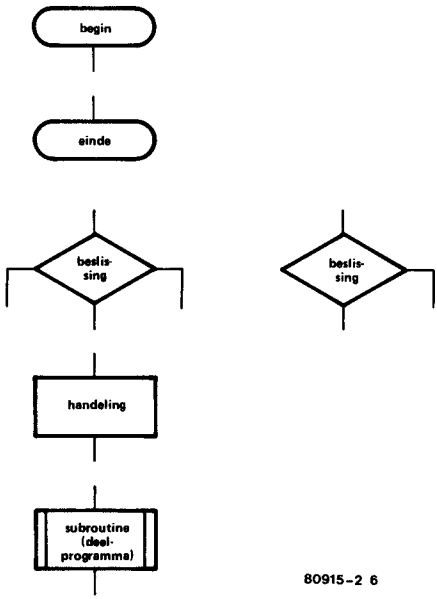
Dat lijkt in de verste verte niet op de binaire kode van het getal 1981. Zoals we al weten is die:

1111Ø111101

Bij de BCD-kode is het inzicht in de decimale waarde van het getal groter dan bij de binaire kode. Dat is een voordeel. Een nadeel is dat het rekenen met getallen in BCD-vorm omslachtiger is dan binair rekenen. Maar niet onmogelijk, getuige het feit dat de 6502-microprocessor van de junior-computer er toe in staat is.

Het probleem in kaart gebracht: het stroomdiagram

Dit gedeelte is misschien een beetje een vreemde eend in de byte, past niet geheel in het kader van dit hoofdstuk 2, maar aan de andere kant is het nuttig om vòòr hoofdstuk 3 te beschikken over bepaalde basiskennis die van belang is voor het programmeren en vòòr het programmeren.
Waarvoor gebruiken we een computer, in ons geval de junior-computer? Voor de oplossing van een probleem. En "probleem" moet men ruim opvatten. De oplossing van het probleem is een taak voor de gebruiker en niet voor de computer. Voor de computer is een andere taak weggelegd: De oplossing geven via de richtlijnen die hem door de gebruiker zijn medegedeeld in de vorm van opdrachten (instrukties). Het vertalen van de richtlijnen in konkrete instrukties in het programmeren, daarover gaat hoofdstuk 3. Dit gedeelte gaat over een belangrijk hulpmiddel bij het vaststellen van de richtlijnen (en dus van de oplossing): het *stroomdiagram,* ook wel bekend onder de naam *flowchart*.
Een stroomdiagram geeft schematisch de gang van zaken weer die hoort bij een bepaalde gekozen oplossing van het probleem. Heel vaak zijn er verschillende oplossingen mogelijk: er zijn meerdere wegen die naar Rome leiden.
Ieder probleem kent een begin en een einde; tussen begin en einde wikkelt zich een aantal akties af in de vorm van beslissingen en handelingen. In figuur 6 staan de bijbehorende symbolen, de onderdelen van het stroomdiagram. Beslissingen worden gesymboliseerd door een zogenaamde testruit; afhankelijk van de kondities zijn er twee of drie verschillende wegen die kunnen worden ingeslagen. De rechthoek met extra lijnen aan beide zijkanten geeft een subroutine oftewel deelprogramma weer. Zo'n deelprogramma heeft zelf dan ook weer een stroomdiagram. Trouwens, dat geldt eigenlijk voor alle onderdelen van een stroomdiagram. In eerste instantie zal men de probleemoplossing ruw schetsen in een stroomdiagram; dat betekent de opdeling in een aantal hoofdbewerkingen en beslissingen. Vervolgens vult men de rechthoeken en testruiten in; dat levert een uitgebreider stroomdiagram op. Naarmate men op deze manier verder doorgaat wordt het stroomdiagram steeds verfijnder. Tenslotte is het dan zo ver dat iedere rechthoek of testruit direkt kan worden vertaald in instrukties voor de (junior-)computer.



Figuur 6. De meest voorkomende symbolen van een stroomdiagram. Er zijn er nog veel meer. Men dient er rekening mee te houden dat er afwijkende symbolen met dezelfde funktie bestaan. N.B. De testruit met drie uitgangen is met name zinvol voor het globale stroomdiagram.

Een voorbeeld, dat nooit tot een computerprogramma zal leiden

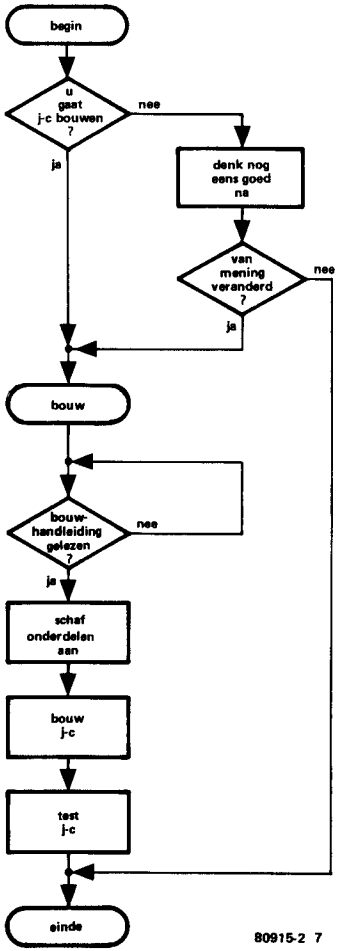
Ter illustratie van het begrip stroomdiagram geven we nu een min of meer uitgewerkt voorbeeld van zo'n stroomdiagram. Het staat in figuur 7 en gaat over de oplossing van een "probleem" dat u als u dit leest misschien al heeft "opgelost": de bouw van de junior-computer, zoals dat in hoofdstuk 1 is beschreven.

U ziet dat nadat de beslissing tot de bouw van de junior-computer is genomen en nadat de bouwhandleiding is gelezen achtereenvolgens de handelingen "schaf onderdelen aan", "bouw j-c" en "test j-c" worden verricht, waarna "einde" in zicht is. Er is een andere weg om daar terecht te komen, namelijk als men besluit de junior-computer niet te bouwen.

De verbinding tussen een uitgang en de ingang van de testruit is een voorbeeld van wat een *wachtlus* wordt genoemd. Pas als "ja" van toepassing is gaan we verder door in het stroomdiagram.

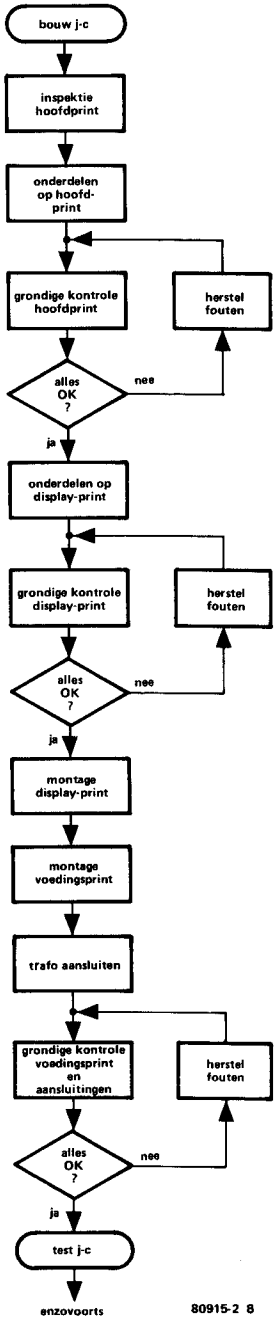
Gaat het in figuur 7 om een zeer globaal stroomdiagram, in figuur 8 is het verder uitgewerkte stroomdiagram gegeven van wat in figuur 7 in de handeling "bouw j-c" is samengevat. U vindt in figuur 8 in grote trekken datgene terug dat in hoofdstuk 1 uitgebreid is beschreven.

We gaan nog een stapje verder. Tot de handeling "onderdelen op hoofdprint" van figuur 8 hoort de montage van de elko C12. Bij deze laatste handeling hoort het stroomdiagram van figuur 9. Een drietal handelingen

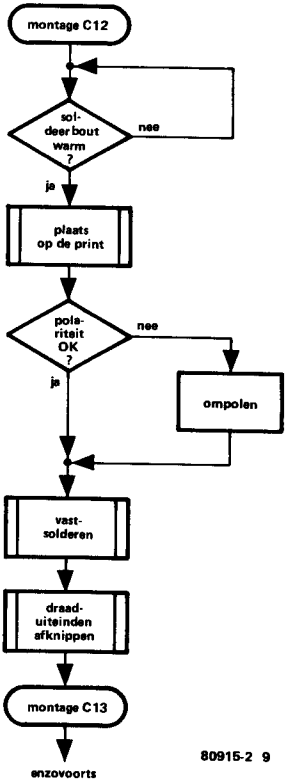


Figuur 7. Het globale stroomdiagram van de bouw van de junior-computer. Dit basis-stroomdiagram is het uitgangspunt voor verder uitgewerkte stroomdiagrammen.

van "montage C12" is in de vorm van een subroutine gegoten. Dat houdt verband met het feit dat de handelingen in kwestie ook van toepassing zijn op een groot aantal andere, op printen te monteren onderdelen. In dat geval hebben subroutines belangrijke voordelen boven normale routines. Het waarom staat in hoofdstuk 3.
Waar we trouwens nu aan toe zijn.



Figuur 8. De handeling "bouw j-c" van het stroomdiagram van figuur 7 is hier verder uitgewerkt in een stroomdiagram.



Figuur 9. Een deel van het stroomdiagram van de handeling "onderdelen op hoofdprint" van figuur 8 bestaat uit het stroomdiagram "montage C12". Drie handelingen van dit stroomdiagram zijn als subroutine opgezet omdat de handelingen ook elders in het uiteindelijke, zeer gedetailleerde stroomdiagram vaak zullen voorkomen; ze betreffen namelijk de montage van een onderdeel op een print.

Oefeningen (de eigenlijke digitale vingeroefeningen)

Eigenlijk was dit een heel vervelend hoofdstuk met al dat ge-nul en ge-een. Maar helaas erg belangrijk. Vandaar de nu volgende opgaven, die voor u een goede test kunnen zijn voor de vraag of u net zo kunt denken als de computer.

1). Wat is de hexadecimale notatie van 00101111?
2). Wat is de hexadecimale notatie van 11111?
3). Wat is de hexadecimale notatie van 101000111?
4). Wat is de hexadecimale notatie van 11010101?
5). Wat is de hexadecimale notatie van 010?
6). Wat is de hexadecimale notatie van 001011?
7). Hoe ziet het hexadecimale getal 132 er binair uit?
8). Hoe ziet het hexadecimale getal A014 er binair uit?
9). Hoe ziet het hexadecimale getal 0356 er binair uit?
10). Hoe ziet het hexadecimale getal A561 er binair uit?
11). Hoe ziet het hexadecimale getal ABBA er binair uit?
(niet te verwarren met het Zweedse export-artikel)
12). Hoeveel is 01001111 + 11000111?
13). Hoeveel is 1110011 +11111111?
14). Hoeveel is 11111111 + 1?
15). Hoeveel is 11110000 + 1111?
16). Hoeveel is 10101010 + 10101?
17). Hoeveel is 01110100 – 1101?
18). Hoeveel is 11110000 – 1111?
19). Hoeveel is 10111000 – 10000001?
20). Hoeveel is 10101111 – 10101111?
21). Hoeveel is 100 – 11111011?
22). Hoeveel is 11110001 x 01111?
23). Hoeveel is 101 x 11111111?
24). Hoeveel is 1010 x 1010?
25). Hoeveel is 11 x 11111111?
26). Hoe groot is het produkt van de hexadecimale getallen 4 en ABBA?
27). Hoeveel is 10000000101 : 111?
28). Hoeveel is 110100000000 : 1101?
29). Hoeveel is 100110111110110010 : 1001110?
30). Hoeveel is $B9A0 : $0B?

Antwoorden

1). 2F
2). 1F
3). 147
4). 1AA
5). 2
6). ØB
7). ØØØ1ØØ11ØØ1Ø
8). 1Ø1ØØØØØØØØ1Ø1ØØ
9). ØØØØØØ11Ø1Ø1Ø1Ø11Ø
10). 1Ø1ØØ1Ø1Ø11ØØØØ1
11). 1Ø1Ø1Ø111Ø111Ø1Ø
12). 1ØØØ1Ø11Ø
13). 1Ø111ØØ1Ø
14). 1ØØØØØØØØ
15). 11111111
16). 11111111
17). 11ØØ111
18). 111ØØØØ1
19). ØØ11Ø111
20). Ø
21). 1ØØØØ1ØØ1 (–247 in 9 bits twee-komplementnotatie)
22). 111ØØØØ11111
23). 1ØØ11111Ø11
24). 11ØØ1ØØ
25). 1Ø11111Ø1
26). 1Ø1Ø1Ø111Ø111Ø1ØØØ = $2AEE8
27). 1ØØ1ØØ11
28). 1ØØØØØØØ
29). 1111111111
30). 1ØØØØ111ØØØØØ = $1ØEØ

3

Programmeren

omgangsvormen

Nadat de junior-computer is gebouwd en nadat we in hoofdstuk 2 een hoop basisinformatie aan de weet zijn gekomen zijn we toe aan het gebruik ervan: wat kun je ermee doen en hoe gaat dat in zijn werk? Deze vragen zullen in dit hoofdstuk uitgebreid worden beantwoord.

Daar staat ie dan, gebouwd en wel. Stekker in kontaktdoos, netschakelaar in en hij is klaar voor gebruik. Eerst moeten we echter zelf klaar zijn voor gebruik. Het wordt nu ernst. Speelse ernst, want u leert spelenderwijs programmeren en programmerenderwijs spelen. Door de junior-computer tijdens het leren te gebruiken leert u hem te gebruiken.

Terreinverkenning

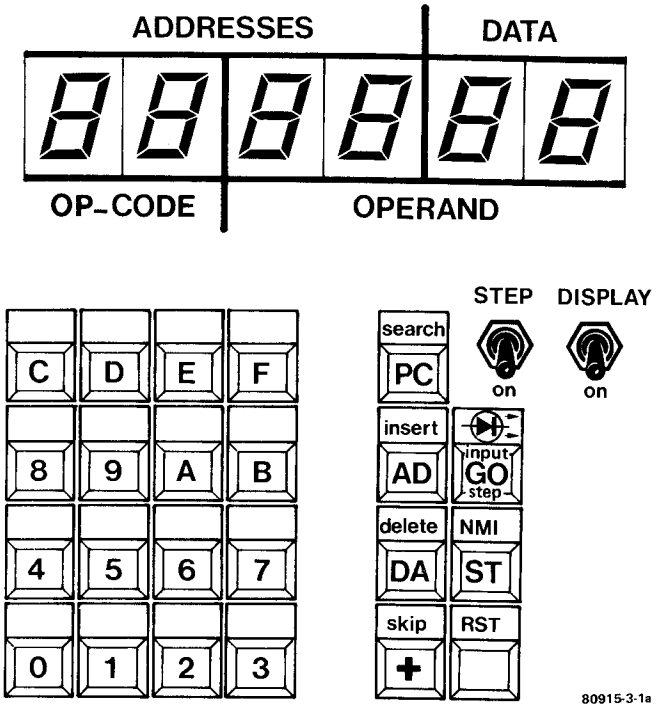
U krijgt te maken met zes zevensegment-displays, 23 toetsen en nog twee losse wipschakelaars. Het "dashboard" is getekend in figuur 1a. Na de "warming up", de aansluiting op het lichtnet en het aanzetten van de netschakelaar en het indrukken van toets RST lichten de displays op; ze geven willekeurige hexadecimale getallen weer. Indrukken van toets RST zorgt ervoor dat het monitorprogramma wordt geaktiveerd. Als gevolg daarvan wordt door de computer gekeken of een van de toetsen is ingedrukt. Gaat het om een van de toetsen Ø . . . F, dan wordt de toetswaarde in beeld gebracht.
De vier linker displays geven hexadecimaal de adressen weer. In principe zijn alle adressen van ØØØØ t/m FFFF mogelijk. De twee rechter displays geven de inhoud weer van de geheugenplaats die zich door het adres in kwestie "aangesproken voelt".
Stel we willen bepaalde geheugenplaatsen met data laden. Op de plaats met adres Ø2ØØ 18, op het volgende adres A9, enzovoorts. We gaan dan als volgt te werk:

toetsen				adres	data				
RST				xxxx	xx	01FF	X	X	
AD				idem	idem	0200	1	8	CLC
Ø	2	Ø	Ø	0200	xx	0201	A	9	LDA
DA				0200	idem	0202	Ø	3	
		1	8	0200	18	0203	6	9	ADC
+		A	9	0201	A9	0204	Ø	7	
+		Ø	3	0202	Ø3	0205	8	D	STA
+		6	9	0203	69	0206	Ø	A	
+		Ø	7	0204	Ø7	0207	Ø	2	
+		8	D	0205	8D	0208	Ø	Ø	BRK
+		Ø	A	0206	ØA	0209	X	X	
+		Ø	2	0207	Ø2	1A7D	X	X	
+		Ø	Ø	0208	ØØ	1A7E	Ø	Ø	
AD				0208	ØØ	1A7F	1	C	
1	A	7	E	1A7E	xx	1A80	X	X	
DA		Ø	Ø	1A7E	ØØ				
+		1	C	1A7F	1C				

Dat is een hele mondvol. Wat is er allemaal gebeurd? Van alles. Eerst is via het indrukken van RST de monitor aangeroepen. De adres- en data-displays geven dan geen kruisjes aan maar willekeurige hexadecimale[1] data (X = don't care, "doet er niet toe" of: "kan van alles zijn"). Door het indrukken van AD geven we een adres op waarop data moet worden geplaatst of worden bekeken. Dat adres is 0200. De toetsen 0, 2 en 0 (2x) worden ingedrukt en het adres waarop data besteld moet worden staat klaar. Na het intoetsen van DA en de data is de geheugenplaats met adres 0200 geladen met data 18. In wezen niet geladen, want die geheugenplaats *was* niet leeg. "Overschreven" is een betere vlag die het laden dekt.
Door de plustoets in te drukken wordt het adres met één opgehoogd ("geïncrementeerd", om het eens moeilijk te zeggen) en de vervolgens ingetoetste data komt op de bij dit nieuwe adres horende geheugenplaats. Het is dus niet nodig om opnieuw DA in te drukken. Dat is wèl nodig als men verderop of verder terug data wil ingeven, bijvoorbeeld op adres 1A7E. Dan moet eerst AD en de adresinformatie worden ingegeven.
Helemaal rechts in de "mondvol" van zoëven is een "plattegrond" gegeven van de stukjes geheugen waarop de laadoperaties betrekking hebben. In de vakjes de data, links daarvan het bijbehorende adres.
We hebben nu kennis gemaakt met:
- de toetsen 0 . . . F, voor het aangeven van adressen en te laden data.
- toets RST. Roept het monitorprogramma aan. Aktiveert displays.
- toets AD. Zorgt voor de adresmode, voor het ingeven van een of meedere adressen.

---

[1] *Zie figuur 15 op pagina 34. De B en de D worden als kleine letters weergegeven. Let op het verschil tussen b en 6 (zes)! De displays lichten alleen op als S25 (DISPLAY) "on" staat!*



Figuur 1a. Het "hardware-dashboard" van de junior-computer. Naast zestien hexadecimale toetsen zijn er zeven funktietoetsen en twee funktieschakelaars. De tekst in kleine letters bij de funktietoetsen heeft betrekking op funkties die de toetsen verrichten als men programma's wil redigeren ("editing"). Dat komt uitgebreid aan de orde in boek 2. Ook de teksten "op-code" en "operand" hebben daarop betrekking. In de normale situatie zijn de displays Di1 . . . Di4 voor het weergeven van adressen (op elk moment één adres) en Di5 en Di6 voor de weergave van data (op elk moment één byte). Adressen en data zijn hexadecimaal gekodeerd. N.B. Deze figuur bevat tevens alle informatie, nodig voor de toetsteksten (zie hoofdstuk 1).

– toets DA. Brengt de junior-computer in de datamode.
– toets +. Indrukken verhoogt het laatst ingegeven adres met één. Mode (adres- of data-) blijft ongewijzigd.

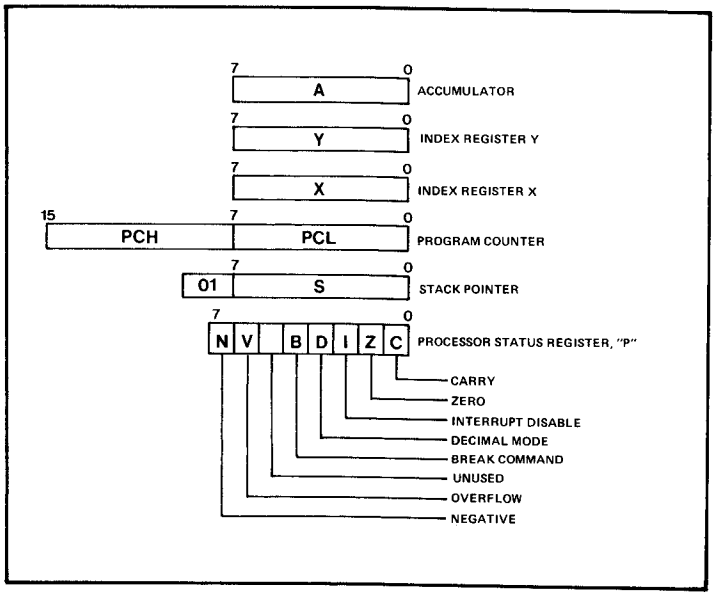
Het laden van data van daarnet betrof niet zo maar willekeurige data. Het ging om een programmaatje. Een programma dat twee hexadecimale getallen bijelkaar optelt. Zonder nu meteen al de hele sluier te verwijderen nu alvast een paar tipjes.

Op adres Ø2ØØ staat 18, kode voor de instruktie CLC, CLear Carry. Vervolgens gaat de computer te rade bij adres Ø2Ø1. Vindt daar A9, de kode voor LDA, LoaD Accumulator immediate. Laden waarmee? Dat staat op het volgende adres, Ø2Ø2. De accu wordt dus geladen met Ø3. Op adres Ø2Ø3 staat 69. Niet negenenzestig maar de hexadecimale kode voor de

ADC-instruktie, ADd memory to Accumulator with Carry oftewel tel de inhoud van de geheugenplaats met het eerstvolgende adres op bij de accu-inhoud. De inhoud van adres Ø2Ø4 is Ø7, de accu bevatte Ø3. De nieuwe accu-inhoud zou best wel eens ØA kunnen zijn.
We zijn zeer benieuwd wat er op adres Ø2Ø5 gebeurt. Achtdé, 8D staat er. Dit wordt herkend als STA, Store Accumulator in memory; berg de accu-inhoud op in het geheugen. En waar dan wel? Het bezorgadres staat op de volgende twee adressen, Ø2Ø6 en Ø2Ø7; het laatste adresbyte, ØA gevolgd door het eerste, Ø2. Het opbergadres is dus Ø2ØA. Op Ø2Ø8 komt ØØ te staan, kode van de instruktie BRK: breek het programma af. Ja, en dan? Geef op de plaatsen 1A7E en 1A7F het adres 1CØØ op waar het na de BRK naar toe moet. Dat is de monitor, zodat we na afloop wat te zien krijgen.
Het programma staat nu in het geheugen. Wat nu? Een programma is er toch om wat mee te doen? Gelijk heeft u. Druk maar eens de toetsen AD, Ø, 2, Ø, Ø en GO in en op het display verschijnt het adres Ø2ØA met de data ØA: resultaat van de optelling van twee getallen. Het ene getal staat op adres Ø2Ø2, het andere op Ø2Ø4. Neem andere getallen en de inhoud van Ø2ØA past zich aan volgens de hexadecimale getalregels.
Het "hoe" en "waarom eigenlijk" van dit programma is hier nog niet aan de orde. Waar het om gaat is dat we met allerlei soorten data te maken krijgen, die naast- en doorelkaar worden gebruikt. Bepaalde data vormt de kode van een bepaalde instruktie. Een instruktie heeft betrekking op bepaalde data. We spreken in dit verband van een *operand*: welke data bijvoorbeeld moet er worden geladen of opgeborgen, of: op welk adres?
Dat "alles doorelkaar" heeft te maken met het "stored program"-concept, daterend uit de veertiger jaren en bedacht door meneer Neumann: programma-instrukties en operanden zijn gelijksoortig gekodeerd en liggen beide in een geheugen opgeslagen. De bijbehorende kringloop(cyclus) is de volgende: kode voor de (volgende) instruktie ophalen uit het geheugen → instruktie dekoderen (wat moet er worden gedaan?) → operand halen, indien van toepassing (waar "laat je de instruktie op los"?) → instruktie uitvoeren → adres vaststellen waarop de volgende instruktie voor de computer ligt te wachten → deze instruktie ophalen → en ga zo maar door. De kreet "programma" zegt nu misschien iets meer: het gaat om een dwingende volgorde van aktiviteiten.

Programmeermodel: het "software-dashboard"

Voordat we uitgebreid duiken in de software-mogelijkheden van de junior-computer, zoals de instruktieset en de adresseermogelijkheden geven we eerst een overzicht van de interne registerstruktuur, de architektuur van de 6502-microprocessor. De inhoud van de registers is namelijk door de gebruiker via het programma te beïnvloeden. En voordat er gesproken kan worden over beïnvloeden moet je weten wat er zo al te beïnvloeden valt. Vandaar "software-dashboard". Vandaar figuur 1b.
De rechthoek A in figuur 1b is een acht-bitsregister, de *accumulator* of, lekker kort, de *accu*. Dit register wordt gebruikt om data uit het geheugen te halen of als vertrekpunt voor in het geheugen op te bergen data. Vaak ook wordt A als tussenstation gebruikt om data van de ene naar de andere geheugenplaats te transporteren.



80915-3-1b

Figuur 1b. Het "software-dashboard" van de junior-computer. Of juister: van de 6502-microprocessor. Het is een overzicht van alle interne registers, waarvan de inhoud is te beïnvloeden via software, dus via het programma. Het processor-statusregister P zou je ook het "vlaggenregister" kunnen noemen.

Zeven van de acht bits van het *statusregister* P (P-register) zijn "vlaggen". Wat zijn dat voor dingen? Bits die 1 of Ø zijn, afhankelijk van de toestand van de bij een bit horende flipflop. Vlaggen zijn "geset" (1) of "gereset" (Ø). Er zijn instrukties waarmee een bepaalde vlag direkt en onvoorwaardelijk 1 of Ø kan worden gemaakt. Bij andere instrukties worden bepaalde vlaggen voorwaardelijk beïnvloed, dus afhankelijk van het resultaat (de uitgangsdata) van de instruktie (zie de laatste kolom van Aanhangsel 2, p. 150 . . . 155). De vlaggen zijn nuttig om in een bepaalde programmafase het verdere verloop ervan te beïnvloeden.
Eén zo'n vlag is de carry, C. Hij wordt uitgehangen (logisch 1) zodra er, bijvoorbeeld bij een optelling van twee getallen een overdracht (carry) plaatsvindt van het achtste (het ultra-linkse bit van een byte) naar het negende bit. Of vlag N, die aangeeft of het resultaat van een bewerking negatief is. Of vlag Z, die het resultaat nul meldt. De overige vlaggen van het P-register komen later aan de orde.
De programmateller, PC, is een zestien-bitsregister. Hierin staat het adres van de geheugenplaats waar de volgende instruktie wacht (denk aan de Neumann-kringloop!). De teller is opgedeeld in register PCL (L = Low), dat het rechter adresbyte bevat, en PCH (H = High) voor het linker byte. Tussen twee instrukties door wijst de PC ook nog adressen aan waar benodigde data, zoals de operanden moeten worden opgehaald.

De X- en Y-indexregisters (8-bits) bevatten ook data. De registers worden geladen via instrukties. Ze zijn bij bepaalde adresseertechnieken (geïndexeerd en indirekt) nodig om bepaalde adressen te bewaren. We komen er nog uitgebreid op terug, evenals op de *stackpointer* S, die adressen aanwijst die bij sprongen vanuit het programma naar een deelprogramma (subroutine) van belang zijn voor de afwikkeling van de diverse aktiviteiten.

Adresseerrepertoire

Maar liefst dertien verschillende adresseermogelijkheden staan de gebruiker van de junior-computer ter beschikking. Dàt is het sterke punt van de 6502-μP. De instruktieset, het opdrachtenrepertoire is niet zo bijster groot: 56 stuks. De kombinatie van een beperkt aantal "krachtige" (slimme) instrukties met een scala aan adresseermogelijkheden zorgt voor een optimale programmeer-vriendelijkheid. Pas veel later zijn andere fabrikanten dan die van de 6502 daarachter gekomen.
Weinig instrukties betekent ook: weinig te onthouden voor de gebruiker. Alle instrukties zijn gekenmerkt door een (hexadecimale) machinekode en door drie hoofdletters, waarvoor het woord *mnemonics* van toepassing is (afkomstig van de Griekse mythologische figuur Mnemosyne, de personifikatie van het geheugen). Een soort instruktie-steno dus.

Immediate-addressing

Onmiddellijke adressering

Instrukties met onmiddellijke adressering worden "losgelaten" op data die in het werkgeheugen onmiddellijk aan bod is nadat de instruktie bekend is. Ze bestaan uit twee bytes; het eerste voor de instruktie-omschrijving (de *opcode*), het tweede bevat de operand-data. Het onmiddellijk-karakter wordt aangegeven door het teken # ("railroad crossing"). Een voorbeeld:
LDA # 7A betekent: laad de accu met data 7A. Of:
LDX # 3B, laad het X-indexregister met data 3B. In plaats van X had er ook Y kunnen staan. Nog een voorbeeld:
ADC # [byte] met als gevolg: A+M+C→A. Tel bij de accu-inhoud het byte M op, dat onmiddellijk volgt op de opcode en tel daarbij 1 of 0 op, afhankelijk van de carry-vlag. Bij een optelling moet deze vlag altijd worden gereset, nul gemaakt. Dat kan via een CLC-instruktie (Clear-Carry). Laten we eens zo'n optelprogramma bekijken:

```
CLC        clear carry (C = Ø)
LDA # 13   laad accu met 13
ADC # 08   tel daar Ø8 bij op
BRK        stop zodra optelling is uitgevoerd
```

Het laatste is nodig om het programma daadwerkelijk te laten lopen en te onderbreken als gedaan is wat moest worden gedaan. Hoe voeren we nu dit programma op de junior-computer uit? Eerst zoeken we de instruktiekodes op (zie tabel achterin het boek). We vinden: CLC = 18; LDA # = A9; ADC # = 69; BRK = ØØ en kiezen als startadres Ø1ØØ. We gaan daarna

als volgt te werk: (inschakelen, displayschakelaar "on", STEP-schakelaar off)

				adres:	display:	
RST	AD			xxxx	xx	
Ø	1	Ø	Ø	Ø1ØØ	xx	
DA		1	8	Ø1ØØ	18	CLC
+		A	9	Ø1Ø1	A9	LDA #
+		1	3	Ø1Ø2	13	
+		6	9	Ø1Ø3	69	ADC #
+		Ø	8	Ø1Ø4	Ø8	
+		Ø	Ø	Ø1Ø5	ØØ	BRK
AD				Ø1Ø5	ØØ	
1	A	7	E	1A7E	xx	
DA		0	0	1A7E	ØØ	
+		1	C	1A7F	1C	
AD						
Ø	1	Ø	Ø	Ø1ØØ	18	
GO				Ø1Ø7	xx	programmastart
AD				Ø1Ø7	xx	
Ø	Ø	F	3	ØØF3	1B	resultaat

Een paar dingen zullen u vast nog niet duidelijk zijn. Waarom die plotselinge sprong van adres Ø1Ø5 naar 1A7E? En waarom staat het resultaat op adres ØØF3? Na de BRK op Ø1Ø5 is het eigenlijke programma voltooid. Wat volgt is een aantal huishoudelijke zaken dat te maken heeft met het monitorprogramma. Na afwikkeling van de BRK-instruktie worden de adressen 1A7E en 1A7F behandeld. In de bijbehorende geheugenplaatsen staat het startadres (1CØØ) van een deelprogramma van de monitor, de *save-subroutine*. Die zorgt ervoor dat de inhoud van elk intern μP-register, dus de inhoud van het "software-dashboard" op bepaalde geheugenplaatsen wordt opgeborgen. En wel als volgt:
Op adres ØØEF de inhoud van PCL,
op adres ØØFØ de inhoud van PCH,
op adres ØØF1 de inhoud van P,
op adres ØØF2 de inhoud van S,
op adres ØØF3 de inhoud van A,
op adres ØØF4 de inhoud van Y en
op adres ØØF5 de inhoud van X.
Op ØØF3 staat de inhoud van A, in ons geval het resultaat, 1B, van de optelling van de twee getallen. En dat was de laatste programmaregel. De voorlaatste betrof het ingeven van het startadres, gevolgd door het indrukken van de GO-toets, waardoor het programma daadwerkelijk is gaan lopen.
Waren we zojuist aan het optellen, aftrekken kan ook. Een nuttige instruktie daarvoor is:
SBC # [byte] met als gevolg: A-M-$\overline{C}$→A. Trek van de accu-inhoud het byte M af, dat onmiddellijk volgt op de opcode. Er wordt in twee-komple-

ment gewerkt (zie hoofdstuk 2). Ook de inhoud $\overline{C}$ (non-C) van de carry-vlag moet worden afgetrokken. Voor het juiste resultaat bij het aftrekken van twee getallen moet de carry-vlag worden gezet, dus $\overline{C}$=Ø, C=1 worden gemaakt. Dat doet de instruktie SEC. Het aftrekprogramma ziet er als volgt uit:

SEC	SEC→kode 38	SBC#Ø8	SBC→kode E9
LDA#13	LDA→kode A9	BRK	BRK→kode ØØ

Ook nu kiezen we startadres Ø1ØØ. We toetsen als volgt:

				adres	data	
RST	AD			xxxx	xx	
Ø	1	Ø	Ø	Ø1ØØ	xx	
DA		3	8	Ø1ØØ	38	SEC
+		A	9	Ø1Ø1	A9	LDA #
+		1	3	Ø1Ø2	13	
+		E	9	Ø1Ø3	E9	SBC #
+		Ø	8	Ø1Ø4	Ø8	
+		Ø	Ø	Ø1Ø5	ØØ	BRK
AD				Ø1Ø5	ØØ	
1	A	7	E	1A7E	xx	zie opm.
DA		Ø	Ø	1A7E	ØØ	
+		1	C	1A7F	1C	
AD						
Ø	1	Ø	Ø	Ø1ØØ	38	programmastart
GO				Ø1Ø7	xx	
AD				Ø1Ø7	xx	
Ø	Ø	F	3	ØØF3	ØB	resultaat

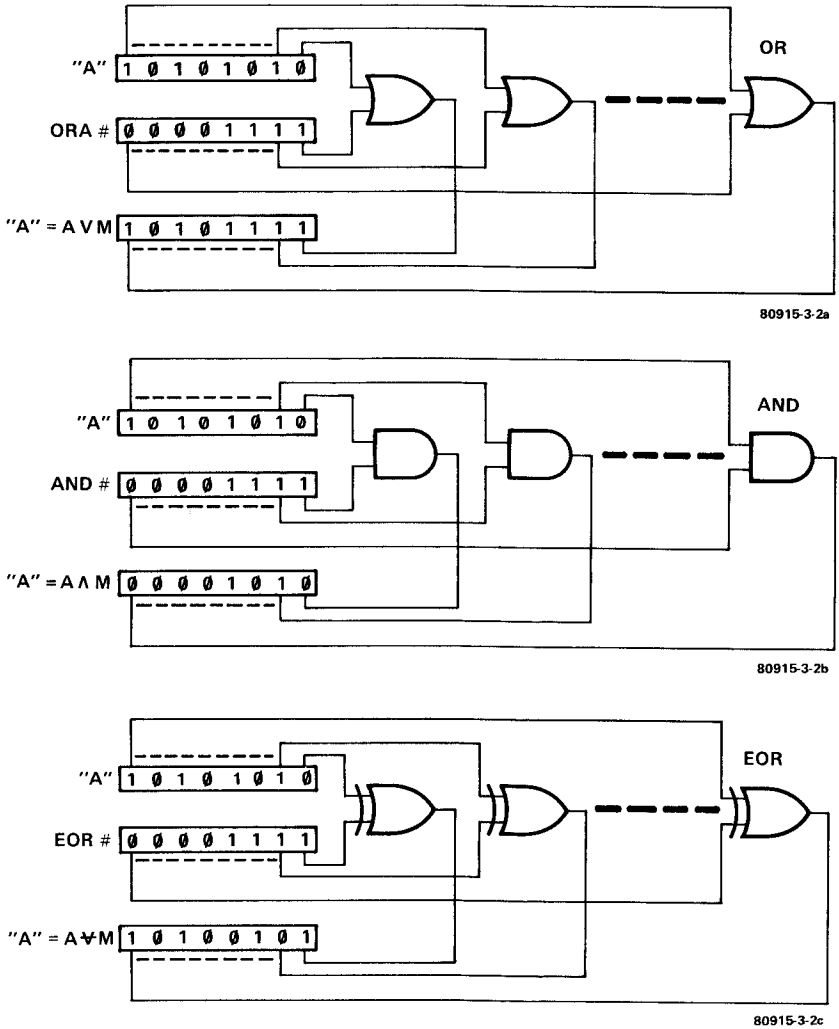
Opmerking. Het laden van de adressen 1A7E en 1A7F hoeft alleen te gebeuren als dat al niet eerder na het inschakelen van de junior-computer is gebeurd, bijvoorbeeld bij het al behandelde optelprogramma.
We zien op adres ØØF3 het resultaat van het programma: Ø8, afgetrokken van 13 (decimaal:19) levert ØB (decimaal: 11).

Logische instrukties

We behandelen in het kader van immediate adresseren nog een aantal logische instrukties.
Het OR-ren is een bekende logische operatie. De instruktie ORA # [byte] heeft tot gevolg: A V M → A. De kode is Ø9. Laten we aan de hand van een programmaatje eens bekijken wat er precies gebeurt:

```
LDA#AA  laad de accu met AA
ORA#ØF  voer bit voor bit de OR-funktie uit met ØF
BRK     stop
```

AA is binair: 1Ø1Ø1Ø1Ø
ØF is binair: ØØØØ1111
resultaat: 1Ø1Ø1111 (AF)
Op bits met gelijke positie wordt de OR-funktie losgelaten. Het resultaat



Figuur 2. De logische bewerking van elk der instrukties ORA, AND en EOR is hier symbolisch voorgesteld met hardware: poorten van een passend type.

komt in de accu te staan. Figuur 2a geeft met hardware (OR-poorten) weer wat er gebeurt.
Een andere logische operatie: AND. De instruktie AND# heeft als gevolg A∧M→A. De kode is 29. Ook hier eerst een programma:

```
LDA#AA   laad accu met AA
AND#ØF   voer bit voor bit de AND-funktie uit met ØF
BRK      stop
```

AA is binair: 1Ø1Ø1Ø1Ø
ØF is binair: ØØØØ1111
resultaat: ØØØØ1Ø1Ø (ØA)

Op bits met gelijke positie wordt de AND-funktie losgelaten; het resultaat staat in de accu. Figuur 2b geeft aan hoe het symbolisch met AND-poorten gaat.
Dan als derde de Exclusive OR (EOR-)funktie. De instruktie EOR # heeft als gevolg: A ∀ M → A. De kode is 49. Een programma:

LDA#AA	laad accu met AA
EOR#ØF	voer bit voor bit de EOR-funktie uit met ØF
BRK	stop

AA is binair: 1Ø1Ø1Ø1Ø
ØF is binair: ØØØØ1111
resultaat: 1Ø1ØØ1Ø1 (A5)
(nul als bits gelijk, één indien ongelijk)
Op bits met gelijke positie wordt de EOR-operatie uitgevoerd. Ook nu staat het resultaat in de accu.
De drie gegeven voorbeelden werken we uit in één programma. Toetsen we als volgt (1A7E: ØØ; 1A7F: 1C):

RST	AD			xxxx	xx		
Ø	1	Ø	Ø	Ø1ØØ	xx		
DA		A	9	Ø1ØØ	A9	LDA#	begin 1
+		A	A	Ø1Ø1	AA		
+		Ø	9	Ø1Ø2	Ø9	ORA#	
+		Ø	F	Ø1Ø3	ØF		
+		Ø	Ø	Ø1Ø4	ØØ	BRK	einde 1
+		A	9	Ø1Ø5	A9	LDA#	begin 2
+		A	A	Ø1Ø6	AA		
+		2	9	Ø1Ø7	29	AND#	
+		Ø	F	Ø1Ø8	ØF		
+		Ø	Ø	Ø1Ø9	ØØ	BRK	einde 2
+		A	9	Ø1ØA	A9	LDA#	begin 3
+		A	A	Ø1ØB	AA		
+		4	9	Ø1ØC	49	EOR#	
+		Ø	F	Ø1ØD	ØF		
+		Ø	Ø	Ø1ØE	ØØ	BRK	einde 3
AD							
Ø	1	Ø	Ø	Ø1ØØ	A9	startadres 1	
GO				Ø1Ø6	AA	stop 1	
AD							
Ø	Ø	F	3	ØØF3	AF	resultaat 1	
AD							
Ø	1	Ø	5	Ø1Ø5	A9	startadres 2	
GO				Ø1ØB	AA	stop 2	

AD						
Ø	Ø	F	3	ØØF3	ØA	resultaat 2
AD						
Ø	1	Ø	A	Ø1ØA	A9	startadres 3
GO				Ø11Ø	xx	stop 3
AD						
Ø	Ø	F	3	ØØF3	A5	resultaat 3

Door het inbouwen van drie onderbrekingen (BRK) kunnen we de drie programmadelen achtereenvolgens uitvoeren. Er wel op letten dat telkens vòòr GO het juiste startadres wordt ingegeven!
De drie behandelde logische instrukties hebben een belangrijke betekenis. Ze worden gebruikt om bepaalde bits van een byte te wijzigen, waarbij de overige bits ongewijzigd blijven. Daartoe wordt het byte in kwestie in de accu gezet en wordt met een geschikt maskeer-byte ge-ORd, ge-AND of ge-EORd. Met als gevolg dat het onderhanden genomen bit wordt geset (1 gemaakt) resp. gereset (0 gemaakt) resp. geïnverteerd (1 wordt 0 of omgekeerd).

Tot zover de immediate-instrukties. Acht stuks hebben we er leren kennen: LDA#, LDX#, LDY#, ADC#, SBC#, ORA#, AND# en EOR#. Daarnaast de instrukties CLC, SEC en BRK, die bij het implied adresseren nog uitgebreid zullen worden behandeld.

Absolute adressering[1]

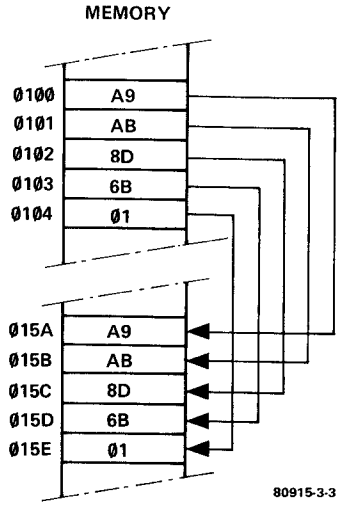
Bij onmiddellijke of immediate adressering volgt de operand-data direkt op de opcode van de instruktie. Bij absolute adressering is de operand-data een adres. Het gaat om het adres danwel de inhoud ervan. Direkt na de opcode volgt dan in twee stappen (bytes) dat adres. Laten we gewoon eens een programmaatje bekijken:

LDA-Ø1ØØ laad accu met de inhoud van plaats Ø1ØØ
STA-Ø15A berg accu-inhoud op op plaats Ø15A
LDA-Ø1Ø1 laad accu met de inhoud van plaats Ø1Ø1
STA-Ø15B berg accu-inhoud op op plaats Ø15B
LDA-Ø1Ø2 laad accu met de inhoud van plaats Ø1Ø2
STA-Ø15C berg accu-inhoud op op plaats Ø15C
LDA-Ø1Ø3 laad accu met de inhoud van plaats Ø1Ø3
STA-Ø15D berg accu-inhoud op op plaats Ø15D
LDA-Ø1Ø4 laad accu met de inhoud van plaats Ø1Ø4
STA-Ø15E berg accu-inhoud op op plaats Ø15E

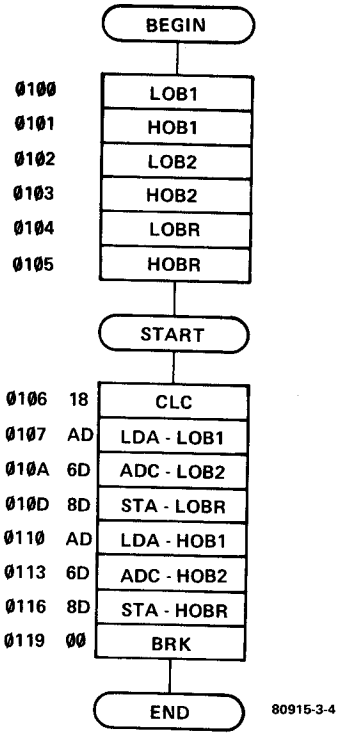
N.B. Het streepje tussen de mnemonics en het adres geeft het absolute adresseringskarakter aan.
Wat is er gebeurd? De inhoud van de plaatsen Ø1ØØ . . . Ø1Ø4 is gekopieerd op de plaatsen Ø15A . . . Ø15E. Dat ging met de accu A als tussenstation. De oorspronkelijke vier geheugenplaatsen blijven hun informatie behouden. Vandaar: kopiëren. In figuur 3 zijn de stukjes "geheugenplattegrond" waar het hier om gaat te zien.

---

[1] *Ook wel direkte adressering genoemd.*



Figuur 3. Geheugenkaart van het kopieer-programmaatje, dat is behandeld in "absolute adressering".



Figuur 4. Stroomdiagram ("flowchart") van het programma dat twee 16-bits getallen bij elkaar optelt.

De instruktie LDA is een oude bekende, zij het nu zonder #. De instruktie STA- is nieuw (het streepje duidt op absolute adressering). Hierdoor gebeurt: A → M; de accu-inhoud wordt opgeborgen in het geheugen (M). Bij absolute adressering beslaan alle instrukties drie bytes. Het eerste byte is de kode voor de instruktie, dus de opcode, Het tweede byte is het rechter adresbyte (ADL; L = Low), het derde byte het linker adresbyte (ADH; H = High).
Laten we eens een programma maken waarmee twee 16-bits getallen bij elkaar worden opgeteld en waarbij instrukties met absolute adressering worden gebruikt. Het eerste getal bestaat uit een linker (HOB1; HOB=High Order Byte) en een rechter (LOB1; LOB=Low Order Byte) byte. Zo ook het tweede getal (HOB2 resp. LOB2). Het resultaat is weer een 16-bits getal, eventueel met carry (Ø1 links toegevoegd aan het getal), dat is opgedeeld in een rechter byte LOBR en een linker byte HOBR.
Voordat we het eigenlijke programma geven en bespreken eerst in figuur 4 het "programma van het programma": de flow-chart oftewel het stroomdiagram. Daaruit maken we op dat de zes geheugenplaatsen vanaf ØlØØ zijn gereserveerd voor de zo juist genoemde bytes, als grondstof of eindprodukt van het programma. Het eigenlijke programma speelt zich af van Ø1Ø6 t/m Ø119. Voordat het programma zich toets voor toets en display voor display voor u ontrolt geven we nog even de bij elkaar op te tellen getallen. Getal 1 is Ø4EF, getal 2 23AB. Daar gaat ie:

(STEP: OFF; DISPLAY: ON)

RST	AD				xxxx	XX	
1	A	7	A		1A7A	XX	
DA		Ø	Ø		1A7A	ØØ	STEP-voorbereiding
+		1	C		1A7B	1C	
+ +					1A7D	XX	
+		Ø	Ø		1A7E	ØØ	BRK-voorbereiding
+		1	C		1A7F	1C	
AD					1A7F	1C	
Ø	1	Ø	Ø		Ø1ØØ	XX	
DA		E	F		Ø1ØØ	EF	LOB1
+		Ø	4		Ø1Ø1	Ø4	HOB1
+		A	B		Ø1Ø2	AB	LOB2
+		2	3		Ø1Ø3	23	HOB2
+					Ø1Ø4	XX	plaats voor LOBR
+					Ø1Ø5	XX	plaats voor HOBR
+		1	8	CLC	Ø1Ø6	18	clear carry-vlag
+		A	D	LDA-	Ø1Ø7	AD	LOB1 in A
+		Ø	Ø		Ø1Ø8	ØØ	
+		Ø	1		Ø1Ø9	Ø1	
+		6	D	ADC-	Ø1ØA	6D	A+LOB2→A
+		Ø	2		Ø1ØB	Ø2	(carry-vlag telt mee)
+		Ø	1		Ø1ØC	Ø1	

```
+          8  D  STA-  Ø1ØD  8D  }
+          Ø  4        Ø1ØE  Ø4  }  resultaat (=LOBR)
+          Ø  1        Ø1ØF  Ø1  }  naar adres Ø1Ø4
+          A  D  LDA-  Ø11Ø  AD  }
+          Ø  1        Ø111  Ø1  }  HOB1 in A
+          Ø  1        Ø112  Ø1  }
+          6  D  ADC-  Ø113  6D  }
+          Ø  3        Ø114  Ø3  }  A+HOB2→A
+          Ø  1        Ø115  Ø1  }  (carry-vlag telt mee)
+          8  D  STA-  Ø116  8D  }
+          Ø  5        Ø117  Ø5  }  resultaat (=HOBR)
+          Ø  1        Ø118  Ø1  }  naar adres Ø1Ø5
+          Ø  Ø  BRK   Ø119  ØØ     eind programma
AD
 Ø  1  Ø  6        Ø1Ø6  18     startadres
GO                Ø11B  xx     terug naar monitor
AD
 Ø  1  Ø  4        Ø1Ø4  9A     resultaat: LOBR
+                  Ø1Ø5  28     resultaat: HOBR
```

Uw vingers zullen wel pijn doen van al dat toetsen. Vandaar nu enige rust, gebruikt om te denken. Namelijk over de achtergronden van het programma. De op te tellen getallen waren:
Getal 1: Ø4EF. Binair: ØØØØØ1ØØ 111Ø1111
Getal 2: 23AB. Binair: ØØ1ØØØ11 1Ø1Ø1Ø11
←HOB→ ←LOB→
Eerst wordt op adres Ø1Ø6 de carry-vlag binnen gehaald, nul gemaakt. Na het laden van de accu met LOB1 en het vervolgens optellen daarvan bij LOB2 wordt de carry-vlag weer een. Kijk maar:

```
          11101111  LOB1
          10101011  LOB2
        111 1111    carry
      + ----------
        [1]10011010 LOBR
carry     ←9→←A→
```

Bij het opbergen van LOBR (op Ø1Ø4) en het binnenhalen van HOB1 in de accu blijft de carry-vlag zoals hij was. We gaan weer optellen:

```
          ØØØØØ1ØØ  HOB1
          ØØ1ØØØ11  HOB2
                1    carry van LOBR
                     (nu helemaal rechts!)
        +     111    carry
      + ----------
        [Ø]ØØ1Ø1ØØØ  HOBR
carry     2    8
```

Het resultaat zien we op Ø1Ø4 en Ø1Ø5. Door het aanroepen van

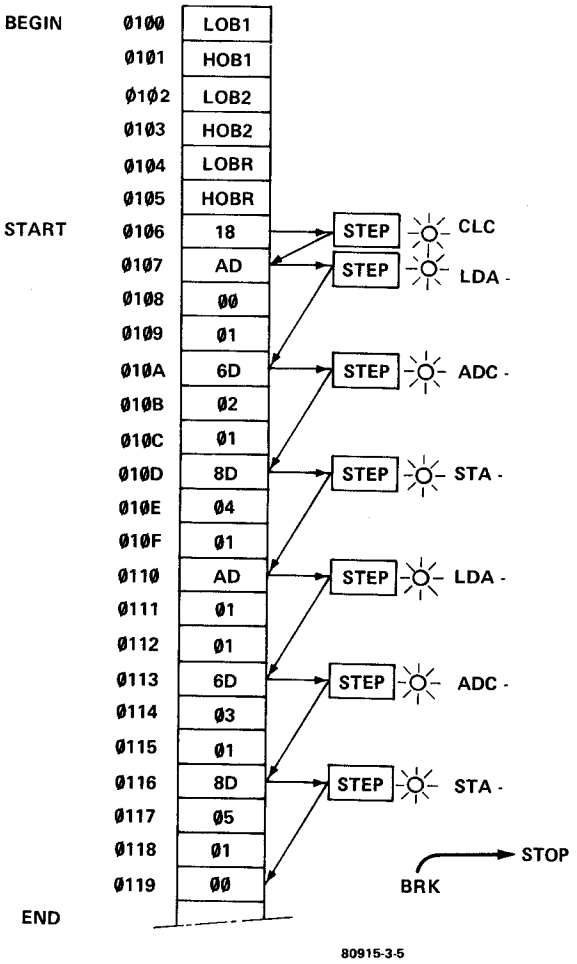
de adressen Ø1Ø4 en Ø1Ø5 krijgen we het resultaat van de optelling in beeld.
Overigens: de voorbereiding – telkens na het inschakelen van de junior-computer – houdt in dat de adressen 1A7E+1A7F en 1A7A+1A7B geladen worden met een zogenaamde vector, een verwijsadres (1CØØ). Meer hierover later.
We hadden het over absoluut adresseren, weet u nog wel? Het optelprogramma van daarnet gebruikte bijna uitsluitend instrukties met absolute adressering. Heel mooi heeft u de bijbehorende afwikkeling van een instruktie in drie fasen kunnen zien: opcode – rechter adresbyte – linker adresbyte.
Er zijn nog veel meer instrukties met absolute adressering. Een overzicht:

Laden en opbergen:
LDA- kode AD (M→A) load accu with memory
LDX- kode AE (M→X) load index X with memory
LDY- kode AC (M→Y) load index Y with memory
STA- kode 8D (A→M) store accumulator in memory
STX- kode 8E (X→M) store index X in memory
STY- kode 8C (Y→M) store index Y in memory

rekenkundige instrukties:
ADC- kode 6D (A+M+C→A) add memory to accumulator with carry
SBC- kode ED (A–M–C→A) subtract memory from accumulator with carry
INC- kode EE (M+1→M) increment memory by one
DEC- kode CE (M–1→M) decrement memory by one
Bij de laatste twee instrukties wordt de geheugeninhoud van M met 1 (ØØØØØØØ1) verhoogd (INC-) of verlaagd (DEC-).
logische instrukties:
ORA- kode ØD (A V M → A) OR memory with accumulator
AND- kode 2D (A Λ M → A) AND memory with accumulator
EOR- kode 4D (A ∀ M → A) Exclusive OR memory with accumulator
(dit zijn drie oude bekenden in een nieuw jasje: – in plaats van #)
Voordat we aan weer een nieuwe adresseermogelijkheid beginnen eerst een intermezzo: een nieuwe programmeermogelijkheid.

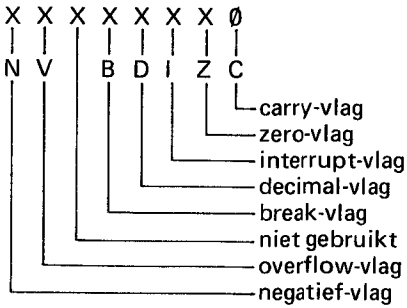
Stap voor stap programmeren

Het indrukken van de toets GO ("ga je gang maar") na het intoetsen van het programma en van het startadres heeft tot gevolg dat het programma wordt afgewikkeld tot en met de eerste BRK-instruktie die de computer op zijn weg tegenkomt. Dat is het geval als de schakelaar STEP in de positie OFF staat. Deze schakelaar heeft ook nog een positie ON. Zetten we hem ON, dan gaat het in de STEP/GO-toets ingebouwde ledje oplichten.
Er gebeurt meer dan dat. Tenminste als we ervoor hebben gezorgd dat adres 1A7A is geladen met ØØ en 1A7B met 1C – een puur huishoudelijke kwestie. Stel we hebben net zo als eerder het programma ingetoetst, inklusief als laatste het startadres. Vroeger drukten we dan GO in, nu STEP. In beide gevallen gaat het om dezelfde toets: de STEP/GO-toets.
We werken nu in de stap-voor-stap-mode. Telkens na het indrukken van



Figuur 5. Het programma van figuur 4, in (geheugen)kaart gebracht.

STEP wordt één instruktie afgewerkt. Figuur 5 illustreert dit. We zien een stukje geheugen met het optelprogramma uit "absoluut adresseren".
Goed, we zijn op Ø1Ø6, drukken STEP in. Dan wordt de carry-vlag nul gemaakt (CLC). Het programma gaat niet verder. We willen wel eens weten of die vlag inderdaad nul is. Herinneren ons dat de toestand van alle vlaggen in het P-register vastligt en dat de inhoud van dit statusregister is "gered" op adres ØØF1. Wat ligt er nou meer voor de hand als daar eens te gaan kijken? Dus toets AD Ø Ø F1 in. Wat zien we op de displays? De adresdisplays wijzen ØØF1 aan. En de data-displays?
Even geduld a.u.b. Eerst iets anders. Het byte in het P-register en dus ook op adres ØØF1 zit als volgt inelkaar:



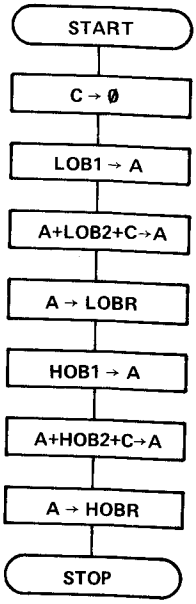
De X-en duiden erop dat de bits 1 of nul kunnen zijn. Dat hangt helemaal af van de voorgeschiedenis vanaf de laatste keer dat de junior-computer is ingeschakeld. In ieder geval één ding: Het minstwaardige bit is het carry-bit. Is dit nul (na CLC), dan moet het rechter hexadecimale cijfer op het data-display decimaal gedekodeerd een even getal opleveren. Is de carry-vlag 1, dan een oneven getal. Zie daar het antwoord op de vraag wat het data-display "doet". We kunnen dus checken of de CLC-instruktie is uitgevoerd.
Er is via AD etc. een uitstapje gemaakt en we willen weer terug naar het adres waar de volgende instruktie (na STEP) op uitvoering wacht. Door indrukken van de PC-toets gebeurt dat; PC staat voor Program Counter en u weet, in de programateller werd het adres bewaard waar de volgende instruktie, althans de instruktie- of opcode ligt opgeslagen.
Na het indrukken van PC verschijnt er Ø1Ø7 AD in het display; AD is de opcode van LDA-. Drukken we STEP. We zien nu: Ø1ØA 6D. Inmiddels is dan de voorgaande LDA-instruktie uitgevoerd, dus LOB1=EF moet in de accu staan. Even checken: de inhoud van A is gered op adres ØØF3. Toets: AD, Ø, Ø, F, 3, en ja hoor, het display toont netjes 00F3 EF. Even indrukken van PC en we kunnen weer verder gaan, hetzij in stappen, hetzij ineens (STEP:OFF), omdat we geen check-uitstapjes meer hoeven of willen maken.
Al met al is de stap-voor-stap-procedure een machtig wapen in de strijd tegen programmeerfouten (het "debuggen", foutloos maken), maar óók een machtig edukatief gereedschap.

N.B. Bij het stap voor stap doorlopen van een programma wordt telkens na het afwerken van een instruktie teruggesprongen naar de monitor, via de NMI-sprongvektor (p. 122), die is vastgelegd op de adressen 1A7A en 1A7B. Het is niet mogelijk om het monitorprogramma zelf, of delen daarvan (subroutines, zie p. 82ff) stap voor stap, dus instruktie voor instruktie te doorlopen. Tijdens het verblijf in de monitor kan de inhoud van allerlei geheugenplaatsen worden bekeken. Bijvoorbeeld ØØEF . . . ØØF5 (zie p. 63).

Zero page addressing

Adresseren op pagina nul

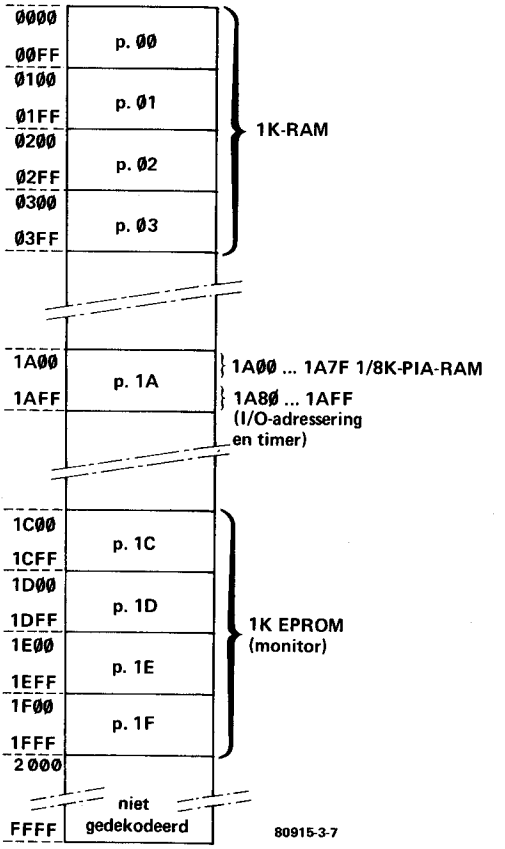
Het gaat om een bijzondere vorm van absoluut adresseren. Instrukties beslaan bij absoluut adresseren drie bytes: opcode – rechter adresbyte ADL – linker adresbyte ADH. Pagina-nul-instrukties onderscheiden zich



Figuur 6. Het "ruwe" (globale) stroomdiagram van het programma van figuur 4.

hiervan doordat het byte ADH altijd nul is (ØØ). Het adresbereik doorloopt dus de mogelijkheden: ØØØØ . . . ØØFF. Deze 256 verschillende adressen horen bij wat we noemen pagina nul. De volgende 256 adressen horen bij pagina 1: Ø1ØØ . . . Ø1FF. Dat gaat zo door tot pagina FF, met de adressen FFØØ . . . FFFF, waarmee alle 256^2 = 65.536 adressen zijn bestreken in 256 pagina's van elk 256 adressen.
Die paginering leidt tot figuur 7, waarin een geheugenkaart (memory map) is getekend. We zien welk geheugendeel (pagina's) voor wat is. De pagina's Ø t/m 3 bestrijken de RAM, het werkgeheugen. Pagina 1A hoort bij de PIA en betreft zowel de interne RAM als de I/O adressering en de timer. Pagina's 1C . . . 1F zijn gereserveerd voor het monitorprogramma. In de standaard-uitvoering van de junior-computer is vanwege de onvolledige adresdekodering (zie hoofdstuk 1) 1FFF het hoogst bereikbare adres.
Terug naar het adresseren. Als de computer automatisch weet (uit de opcode) dat bij een pagina-nul-instruktie het linker byte ØØ is hoeven we dat niet meer op te geven. Met als gevolg twee bytes in plaats van drie bytes per instruktie. En dàt spaart geheugenplaatsen! Pagina-nul-instrukties bestaan dus uit twee bytes: de opcode, gevolgd door het rechter adresbyte ADL. De mnemonics van de opcode wordt vaak aangevuld met een Z, van Zero, nul (is niet "verplicht").

Laden en opbergen:
LDAZ kode A5 (M → A) load accumulator with memory
LDXZ kode A6 (M → X) load index X with memory
LDYZ kode A4 (M → Y) load index Y with memory



Figuur 7. Het geheugen is opgedeeld in pagina's van elk 256 bytes. Binnen een pagina zijn de linker twee nibbles gelijk; de rechter twee nibbles variëren elk van Ø tot en met F.

STAZ kode 85 (A→M) store accumulator in memory
STXZ kode 86 (X→M) store index X in memory
STYZ kode 84 (Y→M) store index Y in memory

rekenkundige (arithmetic) instrukties:
ADCZ kode 65 (A+M+C→A) add memory to accumulator with carry
SBCZ kode E5 (A−M−C→A) subtract memory from accu w. carry
INCZ kode E6 (M+1→M) increment memory by one
DECZ kode C6 (M−1→M) decrement memory by one

logische instrukties:
ORAZ kode Ø5 (A V M → A) OR memory with accumulator
ANDZ kode 25 (A Λ M → A) AND memory with accumulator
EORZ kode 45 (A ∀ M → A) Exclusive OR memory with accumulator

Relative addressing

relatieve adressering

Deze adresseringsmetode wordt toegepast bij vertakkingsinstrukties. De Engelsen en Amerikanen spreken van branch-instructions. Dat zijn voorwaardelijke (conditional) spronginstrukties. Bij onvoorwaardelijke spronginstrukties wordt altijd gesprongen, bij voorwaardelijke uitsluitend onder bepaalde voorwaarden. Die voorwaarden horen bij de test-ruitjes van het stroomdiagram (zie hoofdstuk 2). De te testen grootheden van het programma-stroomdiagram zijn vertaald in te testen vlaggen in het werkelijke programma: is een vlag nul? Prima, doe dan dit; is ie 1? Doe dan dat.

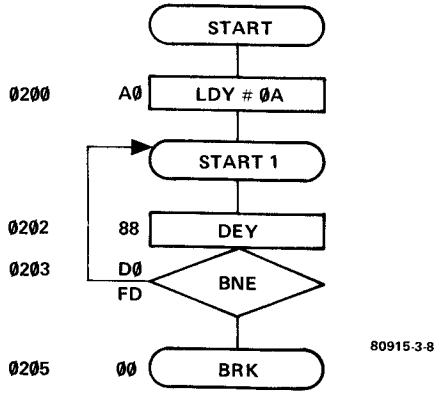
De volgende voorwaardelijke spronginstrukties zijn er:

1. BCC opcode 9Ø Branch on Carry Clear. Spring als C=Ø (gereset)
   BCS opcode BØ Branch on Carry Set. Spring als C=1 (geset)
2. BNE opcode DØ Branch Not Equal. Spring als Z=Ø (gereset)
   BEQ opcode FØ Branch on EQual. Spring als Z=1 (geset)
3. BPL opcode 1Ø Branch on Plus. Spring als N=Ø (gereset)
   BMI opcode 3Ø Branch on MInus. Spring als N=1 (geset)
4. BVC opcode 5Ø Branch on oVerflow Clear. Spring als V=Ø (gereset)
   BVS opcode 7Ø Branch on oVerflow Set. Spring als V=1 (geset)

Met deze instrukties zullen we programma's gaan maken. De onvoorwaardelijke instrukties komen later in dit hoofdstuk aan bod.
Om te beginnen het programma waarvan het stroomdiagram in figuur 8 staat. Op het startadres, Ø2ØØ wordt het Y-register geladen met ØA en via de DEY-instruktie wordt de inhoud van Y met 1 verlaagd. Dus ØA wordt Ø9. Dan komt BNE. Afhankelijk van de toestand van de nulvlag Z wordt teruggegaan naar DEY. Het Y-register bevat bij de eerste BNE Ø9, is dus niet nul en dus is Z nul. Aan de voorwaarde voor springen naar DEY is voldaan. Dit programma doet niets anders dan het Y-register na laden net zolang met 1 te verlagen totdat de geheugeninhoud nul is geworden. Zodra dat het geval is is aan de voorwaarde voor springen niet meer voldaan en stopt het programma (BRK).
In figuur 8 staat bij de linker punt van de testruit de opcode van BNE(DØ) en FD. Dat is het hexadecimale getal dat aangeeft hoeveel geheugenplaatsen er vooruit (als het getal positief is) of terug (negatief) gegaan moet worden om het adres te bereiken dat hoort bij de punt van de pijl die uit de bijbehorende testruit van het stroomdiagram vertrekt.
Nu is FD gelijk aan 11111101 en dat is, rekening houdend met de door de computer gebruikte twee-komplementnotatie −3. Drie plaatsen gaan we terug (het lijkt wel ganzeborden). Dat klopt precies. Kijk maar even mee:

```
Ø2ØØ  AØ  LDY #
Ø2Ø1  ØA
Ø2Ø2  88  DEY
Ø2Ø3  DØ  BNE
Ø2Ø4  FD      aantal stappen
Ø2Ø5  ØØ  BRK
```

(tussen neus en lippen door hebben we een nieuwe instruktie leren kennen: de één-byte-instruktie DEY, verlaag de inhoud van Y met 1)



Figuur 8. Stroomdiagram van een programma met een voorwaardelijke spronginstruktie. Opcodes en offset ("adresverplaatsing") zijn aangegeven.

Direkt na het afwerken van adres Ø2Ø4 staat de programmateller PC al op het volgende adres Ø2Ø5 gericht. En het is de programmateller, waarop de stappen van Ø2Ø4 betrekking hebben. Het na de opcode van een voorwaardelijke instruktie gespecificeerde aantal stappen terug of vooruit heet de *offset*. Een goed Nederlands woord hiervoor is er eigenlijk niet, hoewel wij enigszins trots zijn op de vondst "adresverplaatsing". Maar laten we het voortaan toch maar houden op offset.
Het offsetbereik varieert van maximaal 127 stappen vooruit (+127; hexadecimaal ØØ . . . 7F) tot maximaal 128 stappen terug (—128; hexadecimaal FF . . . 8Ø). Dus totaal: de 256 mogelijkheden van 1 byte. Bij de berekening van een offset moeten we uitgaan van de stand van de programmateller of — praktischer — van het adres van de instruktie die volgt op de voorwaardelijke spronginstruktie. Het berekenen is zonder hulpmiddelen overigens geen eenvoudig karwei. Zonder hulpmiddelen, zeiden we.

Offset berekenen met de monitor

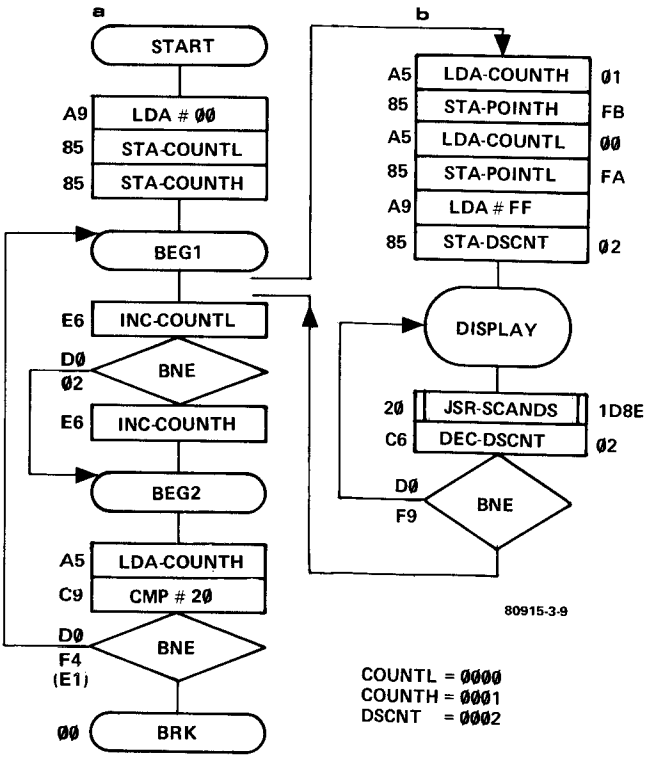
Twee zaken zijn er van belang bij een spronginstruktie. Je moet weten waar de sprong begint (de oorsprong) en waar hij eindigt. Dat is al bekend bij het opstellen van een stroomdiagram zoals figuur 8, moet in ieder geval bekend zijn wil men het offset-rekenwerk laten doen — door de computer. Dat gaat als volgt (met het programma van figuur 8 als voorbeeld): Eerst wordt adres 1FD5 ingetoetst, het startadres van het monitor-offset-routine BRANCH. Vervolgens wordt GO ingedrukt, het rechter byte ingetoetst van het adres waarop de opcode van de spronginstrukties staat (Ø3) en meteen daarna het rechter byte van het doeladres (Ø2). De twee adresbytes verschijnen in het adresdisplay en het datadisplay laat de berekende offset (FD) zien:

AD				XXXX	XX
1	F	D	5	1FD5	D8
GO				ØØØØ	ØØ
Ø	3	Ø	2	Ø3Ø2	(FD) — offset noteren!
RST					

Men kan tussen twee berekeningen in het display ØØØØØØ maken door een willekeurige kommandotoets in te drukken.
Na het indrukken van RST heeft de computer dit rekenprogramma verlaten. Nu kunnen we het programma van figuur 8 intoetsen:

```
AD
 Ø    2     Ø    Ø      Ø2ØØ   XX
DA         A    Ø      Ø2ØØ   AØ    LDY #
 +         Ø    A      Ø2Ø1   ØA
 +         8    8      Ø2Ø2   88    DEY
 +         D    Ø      Ø2Ø3   DØ    BNE    (1A7E: ØØ)
 +         F    D      Ø2Ø4   FD    offset (1A7F: 1C)
 +         Ø    Ø      Ø2Ø5   ØØ    BRK    (terug naar monitor)
```

Waarna het startadres wordt ingegeven en het programma via GO of STEP in beweging wordt gezet. Dat we alleen het rechter byte van oorsprongadres en doeladres hoeven op te geven ligt aan het feit dat het totale aantal offset-mogelijkheden 256 bedraagt. Dat is de informatie-inhoud van een byte=twee hexadecimale karakters.



Figuur 9. Stroomdiagram van een software-teller (9a). Het programma van figuur 9b kan worden tussengevoegd om de tellerstanden zichtbaar te maken. Voor een bespreking, zie pagina 91.

Een software-teller

Als verdere illustratie van voorwaardelijke spronginstrukties nu een programma waarmee een teller wordt gerealiseerd. Zonder solderen! We maken daarbij tevens kennis met een nieuwe instruktie, de CMP-instruktie.

Het gaat om een teller die hexadecimaal van ØØØØ tot 2ØØØ telt en als de maat vol is stopt. Het stroomdiagram staat in figuur 9a. Het getal 2ØØØ beslaat 16 bits; twee geheugenplaatsen zijn er nodig om de tellerstand vast te leggen. Op plaats COUNTH (adres ØØØ1) staat het linker byte, op plaats COUNTL (adres ØØØØ) het rechter byte.
Begonnen wordt met het laden van ØØ in de accu en het via A nul maken van de tellerstand: de teller is dan gereset. Vervolgens wordt de inhoud van COUNTL met één (en meteen) opgehoogd en komen we de eerste voorwaardelijke spronginstruktie, BNE tegen. Daarmee wordt de toestand van de nulvlag Z getest. De geheugeninhoud is ongelijk aan nul, dus wordt er gesprongen naar BEG2 en volgende.
De accu wordt nu geladen met de inhoud van COUNTH en via de instruktie CMP vergeleken met de eindwaarde, 2Ø. In het kort alvast iets over deze instruktie, later meer. De instruktie (opcode C9) beïnvloedt ondermeer de nulvlag. Deze wordt 1 als de inhoud van A (dus van COUNTH) gelijk is aan 2Ø en blijft nul zolang dat niet het geval is. Is de inhoud van COUNTH gelijk aan 2Ø, dan is aan de voorwaarde voor springen bij de tweede BNE niet voldaan en stopt het programma.
Met andere woorden: COUNTL wordt na elke 256 ophogingen ØØ. Dat heeft ophogen van COUNTH tot gevolg (bij de bovenste BNE wordt dan niet sprongen) enwel zolang COUNTH lager dan 2Ø is. Het toetsen van het programma gaat zo (eerst RST; 1A7E: ØØ; 1A7F: 1C):

AD					xxxx		xx	
1	F	D	5		1FD5		D8	startadres offset-berekening
GO					ØØØØ		ØØ	
1	8	1	C		181C		Ø2	offset eerste BNE
2	Ø	1	6		2Ø16		F4	offset tweede BNE
RST	AD							
0	2	1	Ø		021Ø		xx	startadres programma
DA		A	9		ØØ1Ø		A9	LDA#
+		Ø	Ø		ØØ11		ØØ	
+		8	5		ØØ12		85	STA-
+		Ø	Ø		ØØ13		ØØ	
+		8	5		ØØ14		85	STA-
+		Ø	1		ØØ15		Ø1	
+		E	6		ØØ16		E6	INC-
+		Ø	Ø	12	ØØ17		ØØ	
+		D	Ø	11	ØØ18		DØ	BNE
+		Ø	2	10	ØØ19		Ø2	offset
+		E	6	9	ØØ1A		E6	INC-
+		Ø	1	8	ØØ1B	1	Ø1	

```
+        A  5  7  Ø21C  2  A5  LDA-
+        Ø  1  6  Ø21D     Ø1
+        C  9  5  Ø21E     C9  CMP#
+        2  Ø  4  Ø21F     2Ø  operand CMP#
+        D  Ø  3  Ø22Ø     DØ  BNE
+        F  4  2  Ø221     F4  offset
+        Ø  Ø  1  Ø222     ØØ  BRK
AD                Ø222     ØØ
Ø  2  1  Ø        Ø21Ø     A9  startadres
GO                Ø224     xx  programmastart
```

In het programma zijn de eventueel na een BNE te maken sprongen (van de programmateller) in beeld gebracht; twaalf sprongen terug of twee vooruit, F4 of Ø2. Na de BRK (terug naar monitor) kan de tellereindstand worden gekontroleerd: COUNTH bevat 2Ø en COUNTL ØØ. N.B. Zie ook pagina 91!

Voelen we nu de CMP-instruktie nader aan de tand. Hij zorgt voor het vergelijken van de accu-inhoud met bepaalde data. In de immediate-versie (CMP #; opcode C9) is het eerstvolgend byte de data waarmee wordt vergeleken, de referentiedata. Er wordt A-M gedaan: van de accu-inhoud wordt de referentiedata (in M) afgetrokken zonder rekening te houden met de carry-vlag. De accu-inhoud en M zijn na afloop ongewijzigd. Drie vlaggen worden beïnvloed door het resultaat A min M:

a) de N-vlag wordt: geset (1) als A-M 8Ø of hoger is;
gereset (Ø) als A-M 7F of lager is (de N-vlag is gelijk aan het linker bit van A-M)
b) de nulvlag Z wordt: geset (1) als A = M; gereset (Ø) als A ≠ M
c) de carryvlag C wordt: geset (1) als A ≥ M; gereset (Ø) als A < M

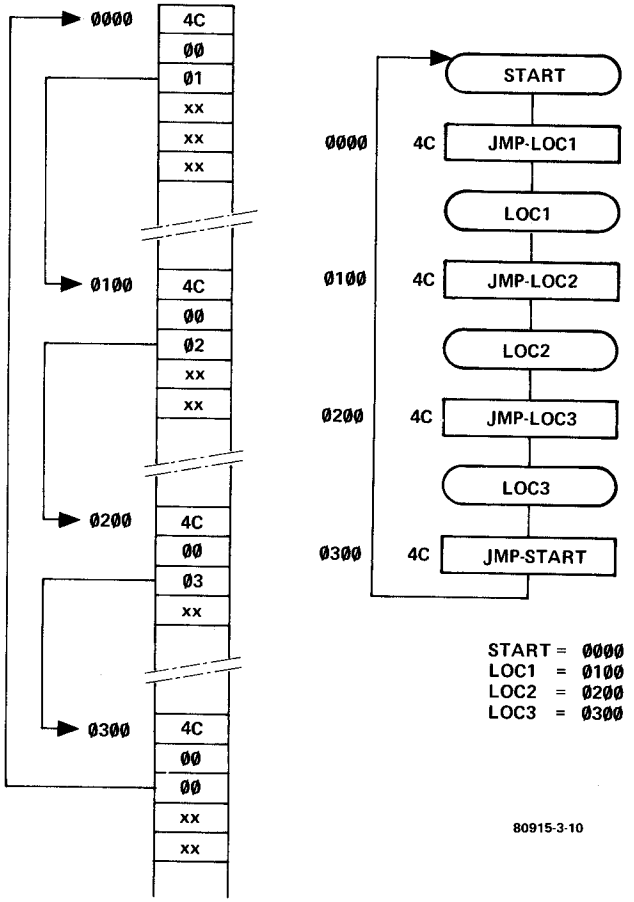
Halen we het voorwaardelijke instruktierepertoire voor de geest, dan weten we nu drie dingen:

a) Met een CMP-instruktie kan worden vastgesteld of A-M in twee-komplementnotatie positief of negatief is; met een aansluitende BMI- resp. BPL-instruktie kan worden gekeken of er afhankelijk daarvan wordt gesprongen of niet.
b) Na een CMP weet men of de accu-inhoud gelijk is aan bepaalde data of niet, met een aansluitende BEQ- resp. BNE-instruktie kan worden gekeken of er afhankelijk daarvan wordt gesprongen of niet.
c) Na een CMP-instruktie weet men of A ≥ bepaalde data is of niet; met een aansluitende BCS- resp. BCC-instruktie kan worden gekeken of er afhankelijk daarvan wordt gesprongen of niet.

De instrukties CPX en CPY doen hetzelfde als CMP. Niet A, maar X resp. Y wordt vergeleken met de inhoud van M.

Onvoorwaardelijke spronginstrukties

Doorliepen de programma's tot nu toe geheugenplaatsen met netjes op elkaar aansluitende adressen (met uitzondering van de voorwaardelijke spronginstrukties, maar dan nog mits:), dat is niet altijd zo. 't Is geen wet



Figuur 10. Een in principe oneindig lang programma, dat bestaat uit vier onvoorwaardelijke spronginstrukties.

van Meden en Perzen. Je kunt een sprong maken naar een ander adres en vandaar gewoon verder gaan. Voor het maken van die sprong heeft de 6502 een handige instruktie in huis:
JMP-. Het is een onvoorwaardelijke spronginstruktie (JuMP=spring) met absolute (let op het liggend streepje achter de mnemonics) of (nog te bespreken) indirekte adressering. Voor absolute adressering is de opcode 4C, voor indirekte 6C. Een 3-byte-instruktie dus, waarvan de laatste twee het sprongadres bevatten. Figuur 10 geeft stroomdiagram en geheugenkaartjes van een programmaatje met uitsluitend JMP-instrukties. Vanuit START worden in een bepaalde volgorde achtereenvolgens en periodiek vier adressen doorlopen: "vlooien springen". Flauw eigenlijk. Maar nu het serieuze werk.

JSR en RTS — een verhaal apart

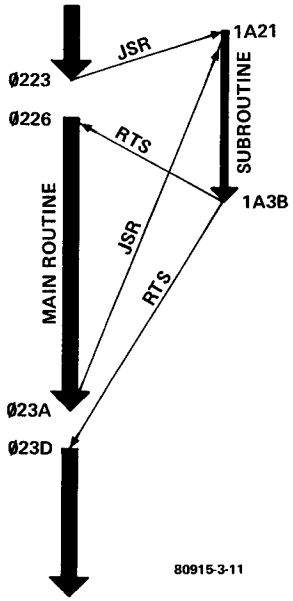
Springen naar en van subroutines

Stel een scholier vraagt de leraar om iets uit te leggen. Die doet dat mondeling. De scholier is echter vergeetachtig en een tijdje later doet hij weer een beroep op de leraar. Stel dat dat zo oneindig lang doorgaat. Dit veronderstelt een oneindig geduldige leraar — en een oneindig domme leerling. Maar dàt kost tijd en energie! Had de leraar nou maar één keer de uitleg op papier gezet, dan kon de leerling elke keer dat het weer zo ver is de schriftelijke uitleg raadplegen! In het programma "leren" is die uitleg op papier een "subroutine" of "deelprogramma"; het pakken van het papier is te vergelijken met JSR, Jump to SubRoutine oftewel spring naar het deelprogramma, en het wegleggen ervan met RTS, ReTurn from Subroutine ofwel verlaat het deelprogramma en ga door met het hoofdprogramma.
De computer is nog veel dommer dan die leerling van daarnet; álles moet je hem uitleggen! Geef hem die dan één keer en verwijs hem ernaar als dat nodig is. Konkreet: komt een bepaalde handeling op verschillende plaatsen in het programma bij herhaling voor, dan is het zinvol om de bij zo'n handeling horende instrukties plus operanden (de subroutine, het deelprogramma) eenmalig in een bepaald stuk geheugen te laden en als de handeling is vereist via een onvoorwaardelijke spronginstruktie JSR te springen naar het startadres van de subroutine. Om na het afwikkelen ervan via een onvoorwaardelijke spronginstruktie RTS terug te gaan naar het hoofdprogramma van waaruit werd gesprongen.
Zinvol omdat er veel geheugenplaatsen worden uitgespaard. Hoeveel dat hangt af van het aantal keren dat een subroutine wordt aangeroepen. En dat is in principe onbegrensd. Zinvol ook omdat het programma kwa struktuur overzichtelijker wordt.
Een programmavoorbeeld. Enwel dat van figuur 11. We zien een subroutine op de geheugenplaatsen 1A21 . . . 1A3B, die twee keer wordt aangeroepen. Op Ø223 staat de opcode van een JSR, op Ø224 (ADL) en Ø225 (ADH) het startadres van de subroutine. De computer is op de hoogte van het adres (Ø225) dat als laatste voor de sprong is afgehandeld; dat adres staat in wat heet de stack of stapel, waarover we het nog zullen hebben. Op Ø23A vindt de tweede JSR plaats; op de adressen Ø23B en Ø23C staat dezelfde adresdata als op Ø224 en Ø225. Een RTS leidt ertoe dat de draad van het hoofdprogramma weer wordt opgenomen op adres Ø23D. Het adres op de stapel is dus 1 lager dan het terugkeeradres.

Stack en stackpointer

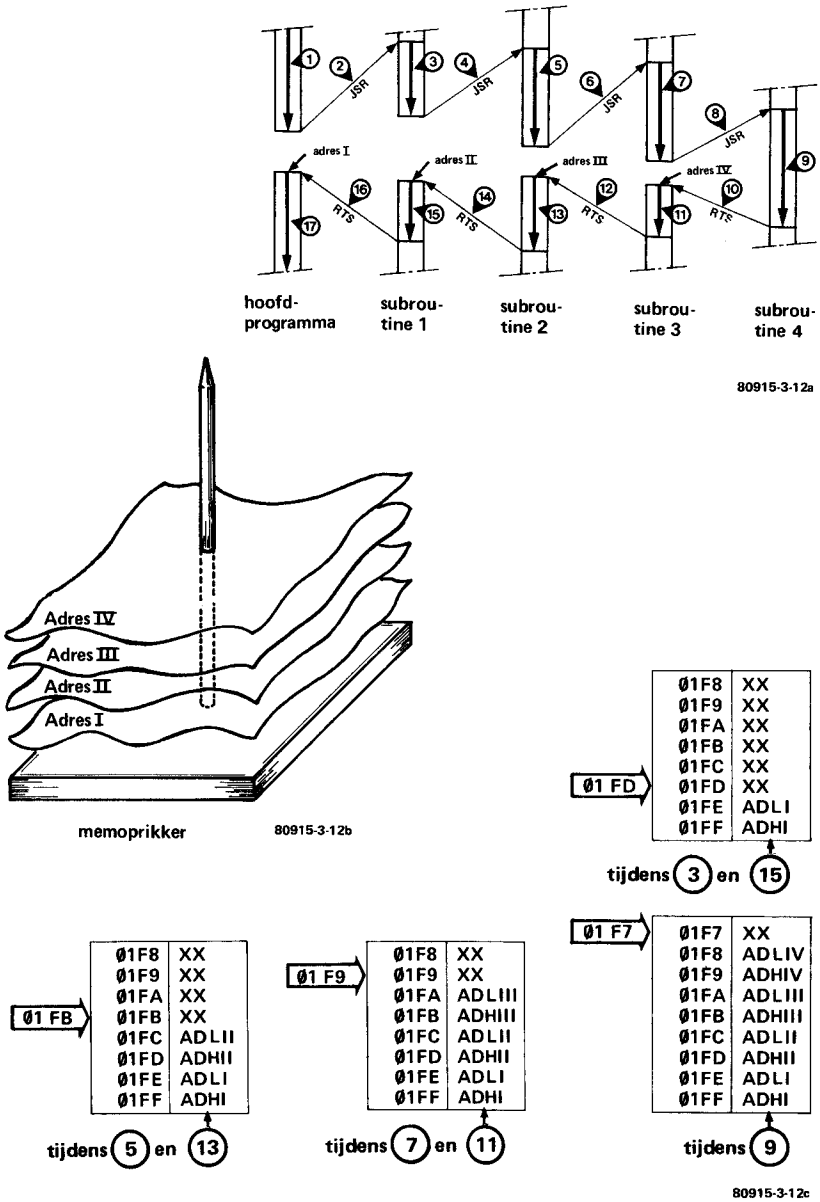
Stapel en stapelwijzer

Bekijk het Droste-cacaobusje van een paar generaties geleden eens goed. Op het busje staat een plaatje van een verpleegster die een cacaobusje in de hand houdt waarop een plaatje staat van een verpleegster die een cacaobusje in de hand houdt waarop . . . en zo zouden we nog uren door kunnen gaan.
Men beschouwe een subroutine waarbinnen wordt gesprongen naar een



Figuur 11. Hoofdprogramma ↔subroutine. Vice versa, dus een retour en nooit een enkele reis!

andere subroutine waarbinnen wordt gesprongen naar weer een andere subroutine waarbinnen . . . enzovoorts. Dit proces heet *nesting* of *nesten*. Denk aan een plastic kubus, één zijde open, die past in een iets grotere die past . . . prachtig kinderspeelgoed.
Het aantal verpleegsters met cacaobusjes is beperkt (ze worden steeds kleiner), het aantal kubussen is beperkt (denk aan de afmetingen van het kind), en ook het aantal keren dat men een JSR kan loslaten, de *nestdiepte* is beperkt. Iedere JSR moet een keer worden gevolgd door een RTS; is het aantal JSR's groter dan het aantal RTS-en, dan "gaat het programma af door de zijdeur", stopt aan het eind van een subroutine.
Bij elke sprong naar een subroutine hoort een terugkeer en bij elk sprongadres een terugkeeradres. Van de terugkeeradressen houdt de computer een administratie bij; dat gebeurt in de *stack(stapel)*. Dat is een stukje geheugen waarin de terugkeeradressen (twee bytes per adres) in een bepaalde volgorde liggen opgeslagen. In wèlke volgorde? Wel, de laatste subroutine waarnaar werd gesprongen wordt als eerste in één keer tegelijk afgewerkt. Met andere woorden: de eerste keer dat na een RTS wordt teruggesprongen wordt de programmadraad opgenomen op het adres dat volgt op de laatste spronginstruktie. Dat bewuste adres is als laatste op de stapel gekomen, ligt dus "boven op" de stapel.
Dus wat het laatste binnenkwam wordt het eerst afgewerkt. Vergelijk het met de afwas: eerst worden borden afgespoeld, dan volgt het eigenlijke afwassen. Na het afspoelen staan de borden op een stapel. Het bovenste bord is als laatste afgespoeld en wordt als eerste afgewassen (tenminste: bij

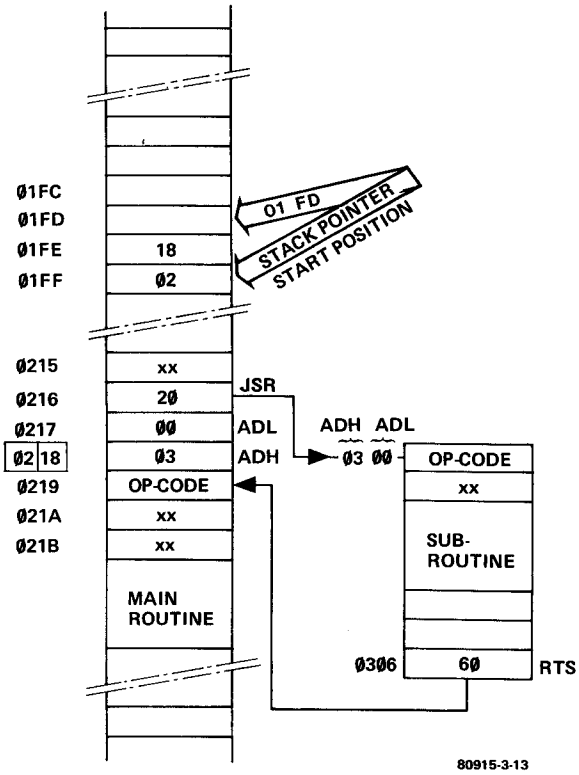


Figuur 12. Het nesten van subroutines (a). De toestand van de stapel (stack) doet daarbij sterk denken aan een memoprikker (b). De toestand van de stapel en van de stapelwijzer (stack pointer) op acht tijdstippen, die ook in a zijn aangegeven (c). Tijdens de programmafasen 1 en 17 staat de stapelwijzer gericht op Ø1FF, de laagste plaats van de stapel, met het hoogste adres van pagina Ø1.

ons thuis). Deze LIFO-procedure (Last In First Out) bij het afwassen en het afhandelen van subroutines is de tegenhanger van FIFO: First In First Out, zoals men dat op zaterdagen bij de bakker of slager tegenkomt ("nummertjes trekken").
In figuur 12 is het nesten van subroutines in beeld gebracht. De pijlen worden afgewerkt in de volgorde [1] ... [17] en bestaan uit afwisselend stukjes programma (geheugenplaatsen) afwerken en springen. In figuur 12b is het stapelmechanisme aanschouwelijk gemaakt met een stapel adresbriefjes op een memoprikker. Het opprikken van een blaadje is het gevolg van een sprong naar een subroutine; is deze afgehandeld, dan wordt het blaadje verwijderd en komt het vervolgens aan bod zijnde adresblaadje boven te liggen. Overigens is het pijlenverloop van figuur 12a een van de ontzettend veel mogelijkheden.
Als stapel kunnen de 256 geheugenplaatsen van pagina 1 worden gebruikt, met adressen Ø1FF ... Ø1ØØ. Er is dus plaats voor maximaal 128 terugkeeradressen. Een geheugenkaart loopt zoals gebruikelijk van boven naar beneden met toenemend adres. De stapelplaatsen worden van onder naar boven met terugkeeradresinformatie geladen (vergelijk figuur 12b!), te beginnen met de eerste adressen Ø1FF en Ø1FE, voor het eerste terugkeeradres.
In figuur 12c is de situatie aangegeven van de stapel en van de stapelwijzer (stack en stackpointer) tijdens acht momentopnamen. We zien dat altijd eerst het linker byte ADH en vervolgens het rechter byte ADL van het terugkeeradres wordt geladen. We zien ook dat de stapelwijzer staat gericht op de laagste lege plaats van de stapel. Wat we niet zien in figuur 12c is dat als terugkeeradres wordt genomen het adres in het hoofdprogramma waarop het linker startadresbyte ADH staat van de subroutine in kwestie. Dat laatste is wèl te zien in het programmavoorbeeld van figuur 13.
Dit programma heeft een voorgeschiedenis die loopt tot en met adres Ø215. Dan, op Ø216 gebeurt het: daar staat 2Ø, de opcode van JSR. Springen waarheen? Op de adressen Ø217 en Ø218 staat waarheen gesprongen wordt; op Ø217 het rechter byte (ØØ) en op Ø218 het linker byte (Ø3) van het startadres van de subroutine. Op de stapel in pagina 1 van het geheugen staat op Ø1FF Ø2 (linker byte) en op Ø1FE 18 (rechter byte), het adres waarop de laatste handeling staat die geheel is afgewerkt alvorens te springen (namelijk een deel van de operand van JSR). Eén plaatsje na dit adres loopt het (hoofd)programma weer verder als is teruggesprongen. De subroutine begint op Ø3ØØ met de opcode van de eerste instruktie. Daarop volgen er meer; welke precies is hier niet interessant. In ieder geval komt op Ø3Ø6 6Ø, opcode van de instruktie RTS, een een-byte-instruktie met (nog te bespreken) implied of ingebouwde adressering. Het "ingebouwde" schuilt hier in het feit dat de operand van RTS (waarnaar moet worden teruggesprongen) (boven) op de stapel ligt. Er wordt teruggesprongen naar het hoofdprogramma en op Ø219 gaat de opcode van een nieuwe instruktie de molen in. Dus: het terugkeeradres is gelijk aan het stapeladres plus één!

Het betere toetsenwerk

Hoogste tijd voor een programma waarin veel onlangs besproken sprong-instrukties aan bod komen. We leren meteen een paar kunstjes kennen uit het rijke repertoire van het monitorprogramma.

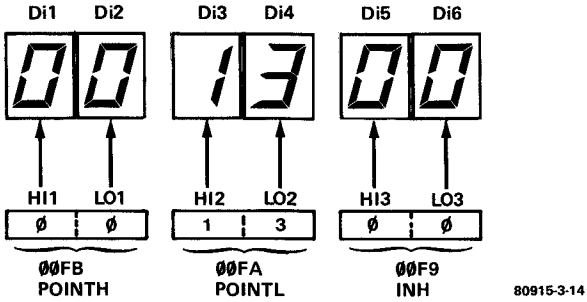


Figuur 13. Programmavoorbeeld met een subroutine.

We willen een programma maken dat het mogelijk maakt de aan een toets toegekende hexadecimale toetswaarde op het display te zetten. Bij een toets hoort de volgende toetswaarde:

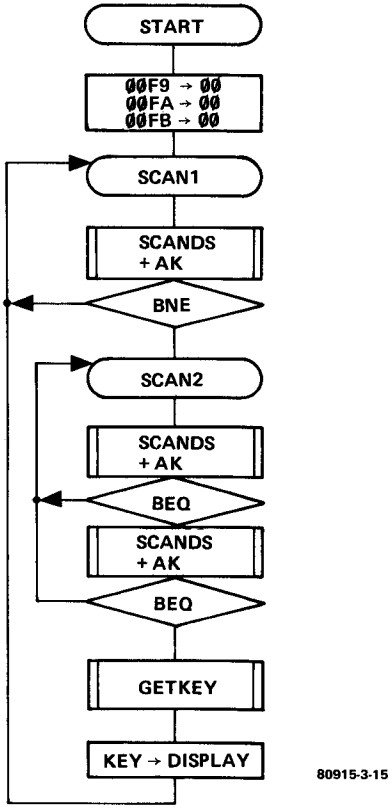
0 : 00	5 : 05	A : 0A	F : 0F	PC : 14
1 : 01	6 : 06	B : 0B	AD : 10	
2 : 02	7 : 07	C : 0C	DA : 11	
3 : 03	8 : 08	D : 0D	+ : 12	
4 : 04	9 : 09	E : 0E	GO : 13	

De waarde van 0 t/m F ligt erg voor de hand; evenzo de aansluitende hexadecimale waarden van de funktietoetsen. Het is dus de bedoeling dat na het indrukken van een toets het bijbehorende hexadecimale getal op het display verschijnt.
Eerst een uitstapje naar de monitor. Die bevat een door de gebruiker aanspreekbare subroutine met de schone naam SCANDS. Die zorgt ervoor dat de inhoud van de geheugenplaatsen 00F9, 00FA en 00FB wordt gedisplayed zoals figuur 14 aangeeft.



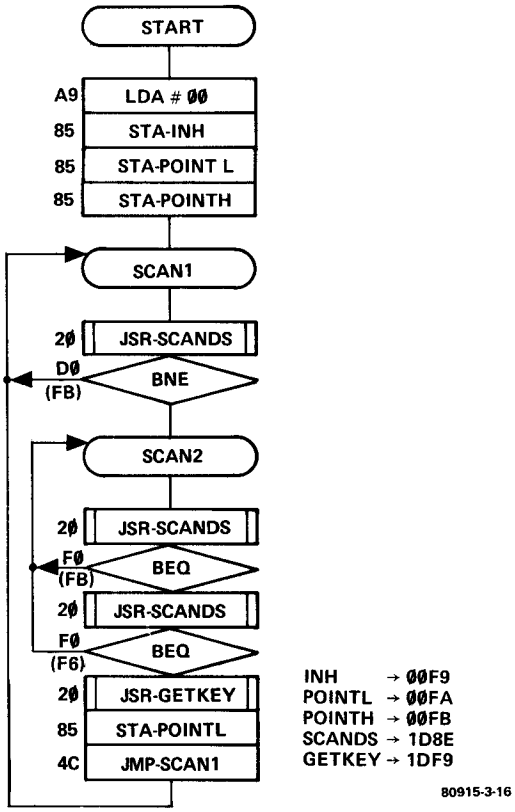
Figuur 14. De displays Di1 . . . Di6 met bijbehorende data-buffers.

De zevensegment-displays moeten de getallen Ø . . . F kunnen weergeven. Dat zijn zestien verschillende mogelijkheden die met vier bits, dat is een halve byte kunnen worden vastgelegd. Wat doet nu SCANDS precies? Eerst worden de linker vier bits uit ØØFB in zevensegmentkode omgezet en aan Di1 toegevoerd, die gedurende een zekere tijd tot oplichterij wordt aangespoord. Dan worden de rechter vier bits van ØØFB na omzetting in zeven segmenten op Di2 gezet en Di2 enige tijd ingeschakeld. Op dezelfde manier komt de inhoud van ØØFA en ØØF9 op het bijbehorende display aan bod.
Hierop sluit een programma AK (Accu Key) aan dat eveneens via een JSR is aan te roepen en dat vaststelt of er een toets is ingedrukt. Nadat alle toetsen een keer zijn ondervraagd kan de accu-inhoud twee toestanden aannemen: A is nul als er geen toets is ingedrukt en ongelijk aan nul als dat wel het geval is. Afhankelijk hiervan kan met een BEQ- of BNE-instruktie al dan niet worden besloten tot een sprong.
Zodra een toets is ingedrukt moet de hexadecimale toetswaarde worden omgezet in de passende zevensegmentkode. Daarvoor is de monitor-subroutine GETKEY geknipt. Ook deze is aanspreekbaar voor de gebruiker. Het grove stroomdiagram van het programma staat in figuur 15. De middelste twee displays moeten de toets in beeld brengen, de linker en rechter twee moeten kontinu ØØ aangeven. Daarom worden aan het begin van het programma de geheugenplaatsen ØØFB en ØØF9 met ØØ geladen. Eerst wordt programmadeel SCAN1 afgehandeld. Dit bevat een SCANDS + AK-subroutine, waarmee de inhoud van geheugenplaatsen ØØF9, ØØFA en ØØFB wordt gedisplayed en waarmee gekeken wordt of een toets is ingedrukt. Via de voorwaardelijke instruktie BNE gaat "men" terug naar SCAN1 zolang een toets niet ingedrukt is geweest. Zodra de accu-inhoud ongelijk nul is *en* de toets losgelaten gaat het programma verder met SCAN2. Daarin wordt de SCANDS + AK-subroutine weer doorlopen. En wéér wordt de lus BEQ-SCAN2 doorlopen zolang A nul is. Vervolgens doen we dit allemaal nog een dunnetjes over.
Op het eerste gezicht lijkt het allemaal zinloos om herhaald vragen te stellen (is er een toets ingedrukt?) als het antwoord na SCAN1 al bekend is. Op het tweede gezicht niet. Gewoon omdat het twee keer doorlopen van SCANDS + AK + BEQ tijd kost. Tijd waarbinnen het dendereffekt van de ingedrukte toets tot het verleden behoort. Zou men deze 'software-debouncing' (geprogrammeerde dender-onderdrukking) niet hebben



Figuur 15. Stroomdiagram van de monitor-routine die zorgt voor toetsdetektie, toetsherkenning en de weergave op het display van de toetswaarde.

ingebouwd dan zou de computer geheid een verkeerde, niet ingedrukte toets hebben herkend.
De staart van het programma bestaat uit de GETKEY-subroutine. De toets krijgt het passende hexadecimale getal toegekend. Eerst wordt nog de accu-inhoud op plaats 00FA opgeborgen, dan wordt gesprongen naar SCAN1, waarna de ingedrukte toets op het display verschijnt. En dan is het voor de junior-computer een kwestie van afwachten tot een volgende toets wordt ingedrukt.
Het gedetailleerde stroomdiagram staat in figuur 16. Als startadres is 0200 genomen. Bij de stap van figuur 15 naar figuur 16 valt eigenlijk niet zo veel meer op te merken. De startadressen van de monitor-subroutines zijn netjes ingevuld. Wat men tevergeefs zoekt zijn de RTS-instrukties na het afwerken van. subroutines. Het monitorprogramma zorgt er namelijk automatisch voor dat na afloop van de subroutine de eerstvolgende af te werken geheugenplaats op het menu komt te staan, dat is dus de geheugenplaats drie adressen hoger dan die waarop de J(monitor-)SR-opcode staat.



Figuur 16. Gedetailleerd stroomdiagram van het programma van figuur 15.

Laten we maar eens gaan toetsen. Eerst de drie offsets berekenen (en noteren!), dan het eigenlijke programma intoetsen:

AD				xxxx	XX	
1	F	D	5	1FD5	D8	
GO				ØØØØ	ØØ	
Ø	B	Ø	8	ØBØ8	FB	→ offset noteren!
1	Ø	Ø	D	1ØØD	FB	→ offset noteren!
1	5	Ø	D	15ØD	F6	→ offset noteren!
RST						
AD						
Ø	2	Ø	Ø	Ø2ØØ	XX	
DA		A	9	Ø2ØØ	A9	LDA #
+		Ø	Ø	Ø2Ø1	ØØ	
+		8	5	Ø2Ø2	85	STAZ
+		F	9	Ø2Ø3	F9	INH

+		8	5	0204	85	STAZ
+		F	A	0205	FA	POINTL
+		8	5	0206	85	STAZ
+		F	B	0207	FB	POINTH
+		2	0	0208	20	JSR–
+		8	E	0209	8E	SCANDS + AK
+		1	D	020A	1D	
+		D	0	020B	D0	BNE
+		F	B	020C	FB	
+		2	0	020D	20	JSR–
+		8	E	020E	8E	SCANDS + AK
+		1	D	020F	1D	
+		F	0	0210	F0	BEQ
+		F	B	0211	FB	
+		2	0	0212	20	JSR–
+		8	E	0213	8E	SCANDS + AK
+		1	D	0214	1D	
+		F	0	0215	F0	BEQ
+		F	6	0216	F6	
+		2	0	0217	20	JSR–
+		F	9	0218	F9	GETKEY
+		1	D	0219	1D	
+		8	5	021A	85	STAZ
+		F	A	021B	FA	POINTL
+		4	C	021C	4C	JMP–
+		0	8	021D	08	SCAN 1
+		0	2	021E	02	
AD				021E	02	
0	2	0	0	0200	A9	programma startadres
GO				0000	00	programma loopt; nul zolang geen toets ingedrukt
GO				0013	00	toetswaarde GO (!!!)
6				0006	00	toetswaarde 6
AD				0010	00	toetswaarde AD
DA				0011	00	toetswaarde DA
B				000B	00	toetswaarde B

Enzovoorts, totdat men er genoeg van heeft:

RST

Na het intoetsen van het startadres is in het voorbeeld GO tweemaal achterelkaar ingedrukt. De eerste keer is nodig om het programma aan het werk te zetten; drukt men vervolgens nog een keer GO in, dan gaat het om de toetswaarde van GO !! Die is 13.

Nogmaals: de software-teller

We weten nu wat subroutines zijn en dat je met de subroutine SCANDS wat kunt laten zien op het display. Bijvoorbeeld het tellen van de software-teller (pagina 78 . . . 80).

Onderbreek figuur 9a na "BEG1" en voeg figuur 9b tussen. Toetsprogramma is van Ø21Ø t/m Ø215 ongewijzigd. Op Ø216 t/m Ø228 komt het programma van figuur 9b; toets de volgende data in: A5, Ø1, 85, FB, A5, ØØ, 85, FA, A9, FF, 85, Ø2, 2Ø, 8E, 1D, C6, Ø2, DØ en F9. Op Ø229 t/m Ø235 komt de rest van figuur 9a (stond vroeger op Ø216 t/m Ø222). De offset van de laatste BNE van figuur 9a (was Ø221; nu Ø234) was F4, is E1.
Wat doet het toegevoegde programma? Eerst gaat COUNTH naar POINTH en COUNTL naar POINTL. Waar normaal een adres staat komt nu een tellerstand (zie figuur 14). Vervolgens wordt DSCNT (ØØØ2) geladen. De inhoud ervan bepaalt hoeveel keer achterelkaar in het aansluitende programmadeel SCANDS wordt doorlopen en dus hoe snel er wordt geteld (Sneller tellen? Verlaag de inhoud van DSCNT; met FF verschijnt elke tellerstand ca. 1 seconde in beeld). Tijdens het tellen blijven de datadisplays op A9 staan (= eerste opcode). De laatste zichtbare tellerstand, 1FFF, wordt, met FF in DSCNT, bereikt na ruim 2 uur (evt. eindstand wijzigen door wijziging van de operand van CMP #, nu op Ø232). Als het programma klaar is toont het display "Ø237 xx". Het adres Ø237 is 2 hoger dan het adres met de BRK. Waarom deze "BRK-remweg"? Zie Aanhangsel 3 op pagina 199 van boek 2.

Implied adresseren: operand bekend

Instrukties met deze vorm van adresseren zijn kort maar krachtig: één byte lang. Er hoeft geen operand te worden opgegeven. In het overzicht op pagina 150 . . . 155 zijn ze met IMP aangeduid (25 stuks). Een paar kennen we al, een aantal wordt nu besproken; BRK, RTI en SEI komen later in dit hoofdstuk uitgebreid aan de orde.
Dit zijn ze:

BRK	DEX	PHA	RTS	TAY
CLC	DEY	PHP	SEC	TSX
CLD	INX	PLA	SED	TXA
CLI	INY	PLP	SEI	TXS
CLV	NOP	RTI	TAX	TYA

SED en CLD

Met de instrukties ADC en SBC was het mogelijk om een bepaalde geheugeninhoud bij de accu-inhoud op te tellen of ervan af te trekken. Bij het optellen van twee getallen kan er sprake zijn van het optreden van een overdracht (carry) als het resultaat van de bewerking groter is dan bijvoorbeeld 11111111=FF. Hoofdstuk 2 vertelt daar meer over. In ieder

geval wordt dan de carryvlag gezet, 1. Nu kan er in de accu niet alleen "normaal" binair worden gerekend maar ook decimaal. Na een SED wordt de D-vlag van het statusregister 1 gemaakt en wordt er decimaal gewerkt. Een CLD maakt de D-vlag nul, waardoor er binair (hexadecimaal) tewerk wordt gegaan.
Het verschil zit 'm in de maximale accu-inhoud die bij overschrijding een carry geeft. Binair moet het resultaat groter zijn dan 11111111=FF voor een carry, decimaal groter dan 10011001=99 (de BCD-kode van 9 is 1001; zie hoofdstuk 2).
Tip. Moet er binair worden gerekend, begin dan altijd met een CLD-instruktie. Het verschil tussen binair programmeren en decimaal doen (door de computer) is namelijk nogal groot. Bij decimaal rekenen begint men met een SED en eindigt met een CLD.
N.B. Na het oproepen van de monitor (indrukken van de RST-toets) is er automatisch sprake van binair rekenen (D=O).

NOP ("doe niets")

Stel je hebt een programma ingetiept van een paar honderd regels (geheugenplaatsen). Het werkt niet, dus maar debuggen. Het blijkt dat bijna aan het begin een instruktie moet worden toegevoegd. O bitter! Dat wordt overtiepen geblazen. Tenzij . . . Tenzij je op een aantal strategische plaatsen in het programma een paar NOP's opneemt, die zijn op te vatten als reserveplaatsen. Nu hoeven slechts weinig geheugenplaatsen te worden gewijzigd. In het gunstigste geval komt de extra instruktie op de plaatsen met een, twee of drie opeenvolgende NOP's. Vallen er bij het debuggen instrukties weg dan kan men de geheugenplaatsen vullen met (de opcode van) een NOP. Dus toch niet helemaal een loze instruktie.

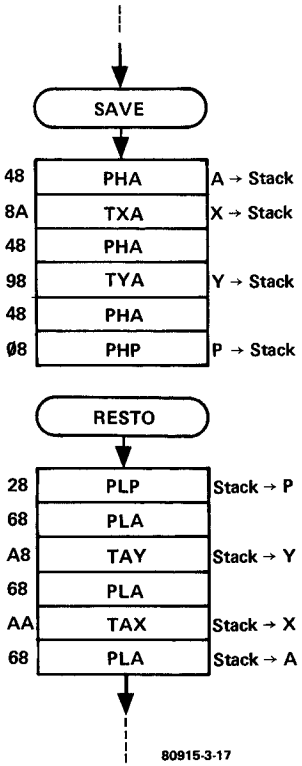
Push-Pull-Transfer

Alle implied instrukties waarvan de mnemonics ("instruktie-steno") beginnen met P of T betreffen intern datatransport van of naar A, X, Y, S of P, of van of naar de stack, op pagina 01. Er zijn heel wat toepassingen van dit data-stuivertje-wisselen.
Bijvoorbeeld: Stel we springen naar een subroutine. Daar hebben we net zoals in het hoofdprogramma het X-register nodig. Het ligt dus voor de hand om alvorens te springen de inhoud van X in veiligheid te stellen, ergens anders op te slaan en na terugkeer van de subroutine weer alles bij het oude te maken:

TXA	breng X-inhoud naar accu
PHA	breng accu-inhoud naar stack
JSR-XXXX	spring naar de subroutine (en doe met X wat je wilt)
RTS	keer terug van subroutine
PLA	breng stack-inhoud terug naar accu
TAX	breng accu-inhoud naar X-register

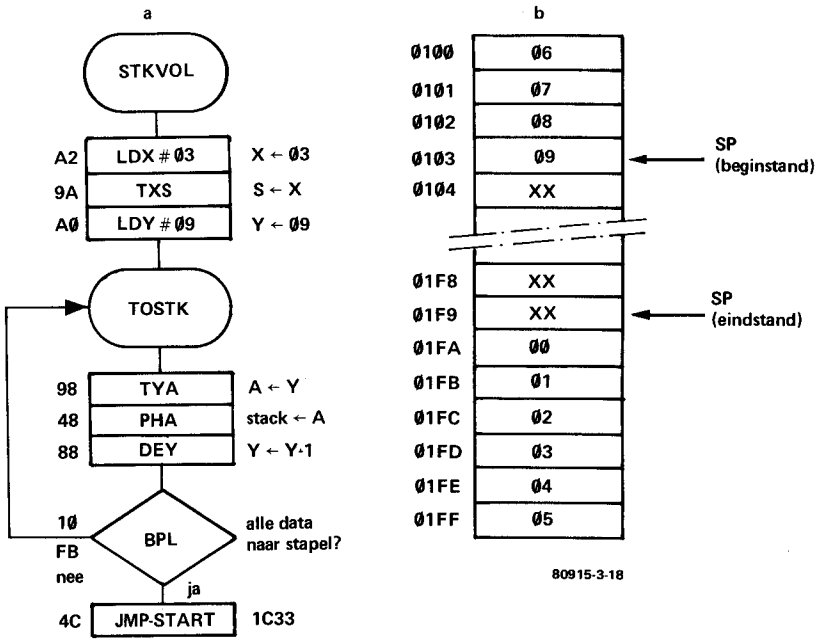
Het had ook anders gekund:

STX-SAVX	breng inhoud X-register naar geheugenplaats SAVX
JSR-XXXX	spring naar subroutine (X-register vrij ter beschikking)
RTS	keer terug van subroutine
LDX-SAVX	breng inhoud geheugenplaats SAVX terug naar X-register



Figuur 17. Het "redden" (bewaren) van de inhoud van interne registers van de 6502-microprocessor op de stapel, en het weer beschikbaar stellen ervan gaat met push-instrukties respektievelijk pull-instrukties.

De instrukties STX en LDX vergen echter meer bytes. De eerst gegeven oplossing is dus beter. De stack is dus een ideale tijdelijke dataopslagplaats. Ging het zojuist om het in de ijskast zetten van de inhoud van het X-register, vaak is het nodig om vòòr de sprong naar een subroutine de inhoud van alle registers A, X, Y en P te bewaren (op de stack) en na afloop van de subroutine weer ter beschikking te stellen. Daartoe is opname van een bewaarprogramma SAVE vòòr de sprong en een herstelprogramma RESTO direkt na de terugkeer uitermate handig. Figuur 17 geeft de twee programma's (routines) weer. Bij SAVE wordt eerst de inhoud van A op de stapel gezet ("push"), vervolgens die van X, Y en P, met de accu als tussenstation. In het RESTO-geval is het net andersom: de stapel wordt stukje voor stukje afgebroken ("pull") en via de accu gaat de bewaarde data naar het register in kwestie. We zien ook dat wat het laatst de stapel opging er weer het eerst afgaat. Dat heeft alles te maken met het mechanisme van de stapelwijzer (stack pointer) dat we bij de subroutines hebben besproken.



Figuur 18. Programma, waarmee achtereenvolgens tien data de stapel op gaan (a). De stapel en de stapelwijzer (SP = Stack Pointer) na afloop van dit programma (b).

De stapelwijzer (ligt vast via S) staat gericht op een geheugenplaats in pagina Ø1, die als eerste aan bod is als er (opnieuw) data naar de stapel komt. Na het aanroepen van de monitor *via het indrukken van RST* staat de stapelwijzer gericht op Ø1FF, de laagste plaats (met het hoogste adres) van de stapel. Je kunt deze wijzer ook richten op een ander adres in pagina Ø1. Dat gaat zo:

LDX #Ø3 stop Ø3 in het X-register
TXS breng X naar het register S

De stapelwijzer staat nu gericht op Ø1Ø3. Nu is Ø1ØØ de hoogste stapelplaats, met het laagste adres. Met Ø1Ø3 meegerekend zijn er dus nog vier "lege" plaatsen. Wat gebeurt er nou als er tien keer achter elkaar data op de stapel wordt gegooid, dus na uitvoering van het programma van figuur 18a? Figuur 18b laat dit zien. Het is niet zo dat na de eerste vier data (waar nog plaats voor is) de volgende zes data de mist in gaan omdat de stapel vol is. Nee, als de stapel vol is, beginnen we opnieuw van beneden af, dus vanaf adres Ø1FF. Het is dus *niet* zo dat, als de stapel vol is, deze doorloopt naar pagina ØØ.

Het opnieuw beginnen vanaf Ø1FF betekent dat de oorspronkelijke data wordt overschreven en dus verloren gaat. Nu wordt er in het algemeen belangrijke programma-data op de stapel opgeborgen. Het kan dus zaak

zijn om er op te letten dat de stapel niet over loopt, zoals met het programma van figuur 18a het geval is.
N.B. In figuur 18a wordt na afloop gesprongen naar het centrale punt START van de monitor (zie hoofdstuk 7 in boek 2).

Accu-adressering

Het gaat om een variant op de ingebouwde adressering. Om vier een-byte-instrukties die de accu-inhoud manipuleren. Om ASL, LSR, ROL en ROR. Schuiven (shift) en draaien (rotate). De toepassing zit in allerlei rekenkundige bewerkingen.

LSR – Logical Shift Right (opcode 4A). Hierdoor schuiven de bits naar rechts:

```
b7  b6  b5  b4  b3  b2  b1  bØ
 p   q   r   s   t   u   v   w   [C=X]
         ↓
        LSR
         ↓
 Ø   p   q   r   s   t   u   v   [C=W]
```

Bovenaan staat de bitpositie (bee-zoveel). De letters P . . . W stellen een bepaald bitpatroon voor van enen en nullen. We zien dat alle bits naar rechts schuiven; het meest linkse bit wordt Ø en bit bØ=W bepaalt de carry-vlag. Is W Ø dan wordt de carry gereset, bij een 1 voor W geset. De N-vlag blijft Ø. Acht keer achter elkaar een LSR levert een accu-inhoud ØØØØØØØØ op. Uitsluitend dan wordt de Z-vlag geset.
ASL–Arithmetic Shift Left (opcode ØA). De bits schuiven nu naar links:

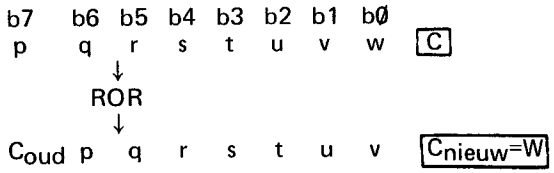
```
      b7  b6  b5  b4  b3  b2  b1  bØ
C=X    p   q   r   s   t   u   v   w
              ↓
             ASL
              ↓
C=P    q   r   s   t   u   v   w   Ø
```

Alle bits schuiven een plaatsje naar links; het meest rechtse bit wordt Ø en bit b7 (was P) bepaalt de carry-vlag: 1 bij P=1 en Ø bij P=Ø. De N-vlag wordt geset zodra bit zeven 1 (Q) is. De Z-vlag wordt geset bij een accu-inhoud ØØØØØØØØ, die bijvoorbeeld optreedt na acht opeenvolgende ASL's.
ROL–ROtate Left (opcode 2A). Eens kijken:

```
b7  b6  b5  b4  b3  b2  b1  bØ
 p   q   r   s   t   u   v   w   [C=X]
         ↓
        ROL
         ↓
 q   r   s   t   u   v   w   Coud      [Cnieuw=P]
```

Ook hier schuiven de bits een plaats naar links op. Het verschil met ASL is dat nu bit bØ gelijk is aan de oude waarde van C. Voor het overige is het vlagvertoon hetzelfde als bij ASL.

ROR—ROtate Right (opcode 6A). Een keer raden:



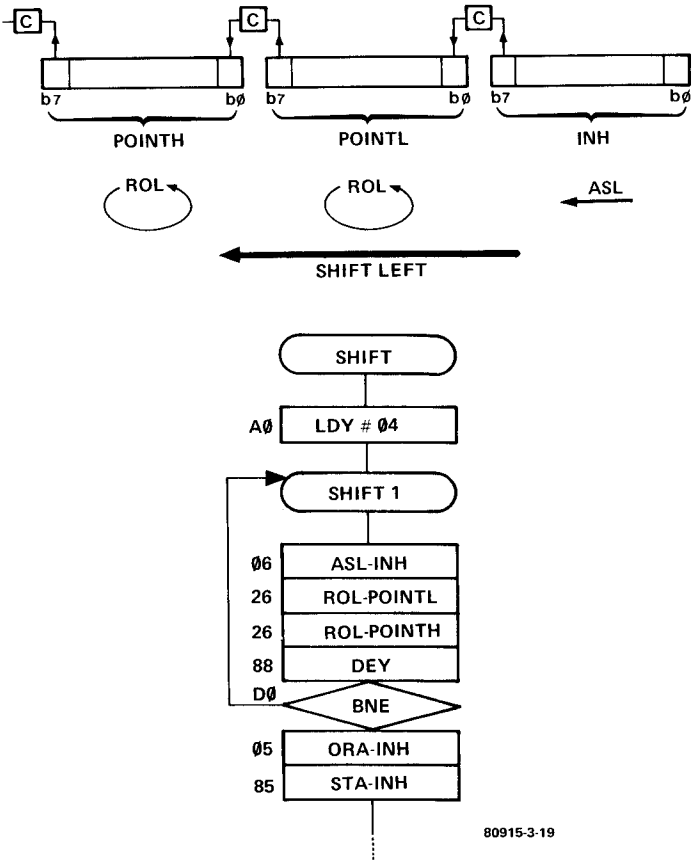
Dat lijkt allemaal erg veel op LSR. Met dit verschil dat het zevende bit nu niet Ø wordt maar de oude waarde krijgt van C. Voor het overige is het allemaal gelijk aan de situatie bij LSR.
Het verschil tussen schuiven en draaien: bij schuiven gaat er een bit uit de accu verloren, bij draaien niet.
N.B. De vier schuif/draai-instrukties van daarnet kunnen ook worden losgelaten op de kombinatie: carry-vlag-inhoud van een geheugenplaats. Het gaat dan om instrukties met absolute adressering of varianten daarvan.

Schuiven en draaien — een voorbeeld

We weten al dat ingetoetste data bij de junior-computer van rechts naar links in het display doorschuift. We weten ook van figuur 14 dat de weer te geven data ligt opgeslagen in een drietal display-buffers: de geheugenplaatsen met adressen ØØFB, ØØFA en ØØF9. Het doorschuiven van weer te geven informatie is in figuur 19 in beeld gebracht. Zodra een nieuwe toets wordt ingedrukt kan naar SHIFT worden gesprongen. Als gevolg daarvan schuiven alle bits van de drie buffers vier plaatsen op naar links. Vier bits zijn precies genoeg voor de verzorging van een display-karakter, dus na SHIFT zijn de linker vijf op het display staande karakters een plaats naar links opgeschoven; het meest linkse karakter gaat verloren en het meest rechtse karakter hoort bij de ingedrukte toets die de oorzaak was van het aanroepen van SHIFT.
Tijdens het vier keer achterelkaar bitje schuiven dient de carry-vlag als tussenstation. Eerst schuift de b7 van INH de carry in als gevolg van een ASLZ (de aan de mnemonics toegevoegde Z duidt op het gebruik van pagina-nul-adressering; opcode Ø6) en wordt bit nul Ø. Vervolgens laat de monitor een ROLZ (opcode 26) los op POINTL; de bØ hiervan krijgt de carry-inhoud toegekend, wordt dus gelijk aan de b7 van INH en de b7 van POINTL schuift nu de carry in. De derde fase betreft ook een ROLZ, toegepast op POINTH. Bit nul van POINTH is uiteindelijk gelijk aan b7 van POINTL en de b7 van POINTH bepaalt nu de carry-vlag. En dat gaat zo vier keer achterelkaar. Uiteindelijk zijn de oorspronkelijke linker vier bits van POINTH verloren gegaan; de rechter vier bits van INH zijn Ø geworden. De laatste twee rechthoeken van figuur 19 betreffen het gelijk maken van deze vier bits aan de toetswaarde van de ingedrukte toets (die in de accu staat).
Nu volgt een programma waarbij gebruik wordt gemaakt van een aantal zojuist besproken instrukties en monitor-routines.

Decimaal optellen

Voor het decimaal optellen van twee getallen van zes cijfers staan ons zakrekenmachientjes ter beschikking van een paar tientjes — of potlood en



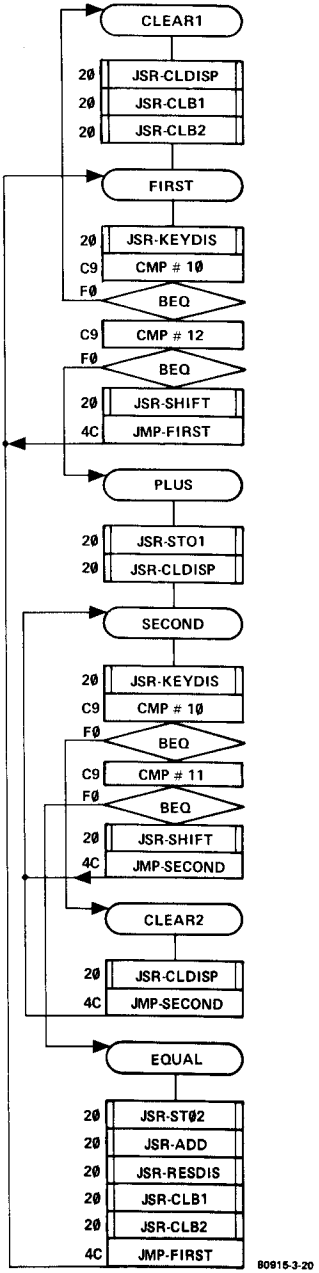
Figuur 19. Programma waarmee ingetoetste data van rechts naar links het display inschuift. Zie ook figuur 21a op pagina 100, behorend bij het decimale optel-programma.

papier voor een paar dubbeltjes. Een speciaal optelprogramma maken lijkt op het te lijf gaan van een mug met een kanon. Is dat ook. Maar door dit te doen leren we het kanon te gebruiken voor meer ingewikkelde, vreedzame toepassingen.
We willen als volgt optellen:

X X X X X X + X X X X X X = X X X X X X
eerste getal tweede getal resultaat

en stellen nog zo wat eisen:

De getallen moeten bij het intoetsen van rechts naar links in het display verschijnen en dus van links (honderdduizendtallen) naar rechts (eenheden) worden ingetoetst. Voor het ingeven van de getallen moet het display schoongemaakt ("clear", een software-"bordenwisser"). De toetsen Ø ... 9 zijn nodig voor de getallen; als = doet DA , met toetswaar-

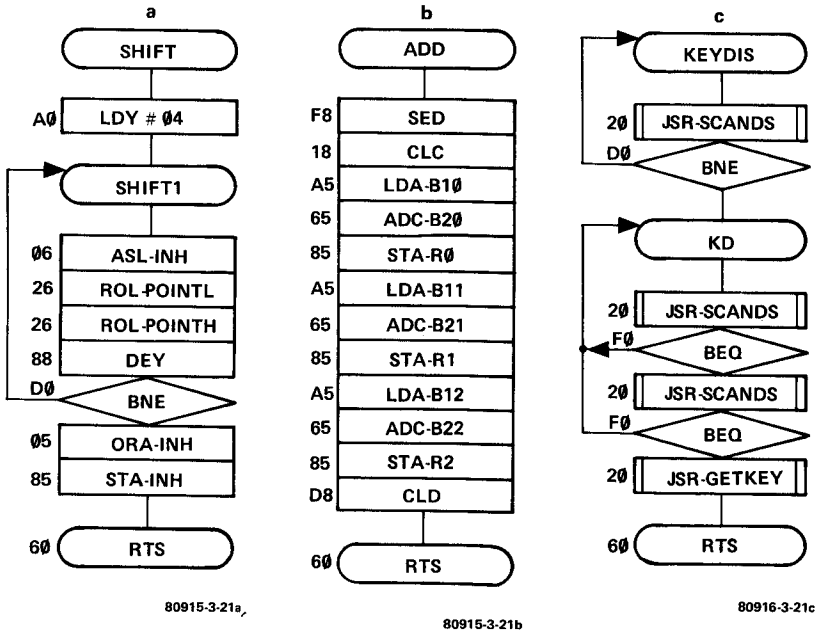


Figuur 20. Hoofdprogramma voor het decimaal optellen van twee getallen van zes cijfers.

de 11 dienst en als schoonmaaktoets doet AD , met toetswaarde 1Ø dienst. Verder is uiteraard + nodig. Is het resultaat van de optelling groter dan 999999, dan moet er geen foutmelding komen; deze situatie wordt apart aangepakt. Een foutief ingetoetst getal moet via een clear worden weggemaakt, waarna men opnieuw kan gaan intoetsen.
In figuur 20 staat het hoofdprogramma voor het decimaal optellen van twee getallen. Nou, daar komt wel wat bij kijken. Maar liefst negen subroutines worden er gebruikt; de deelprogramma's staan in figuur 21.
Nog een paar opmerkingen vooraf. Er zijn 4x3=12 geheugenplaatsen (buffers) nodig om de diverse data in kwijt te raken. Naam en adres van elk volgt hieronder:

POINTH ØØFB	POINTL ØØFA	INH ØØF9	display-buffers
B12 ØØØ2	B11 ØØØ1	B1Ø ØØØØ	buffers voor het eerste getal
B22 ØØØ5	B21 ØØØ4	B2Ø ØØØ3	buffers voor het tweede getal
R2 ØØØ8	R1 ØØØ7	RØ ØØØ6	buffers voor het resultaat van de optelling

waarbij elke buffer twee displays bedient. De volgorde van links naar rechts is gelijk aan de volgorde waarin men een getal opschrijft.
Het programma (figuur 20). Begonnen wordt met CLEAR1, met als gevolg dat display- en getalbuffers met ØØØØØØ worden gevuld (zie ook de figuren 21d, 21e en 21f). Dan volgt FIRST: meteen springt het programma naar de subroutine KEYDIS (figuur 21c), waarin gebruik wordt gemaakt van SCANDS en GETKEY, oude bekenden die zorg dragen voor de sturing van het display, het afvragen van het toetsenbord en de software-kontaktdenderonderdrukking. Is er een toets ingedrukt en losgelaten dan stelt GETKEY vast welke toets dat is. Na de terugkeer van KEYDIS in het hoofdprogramma staat de toetswaarde van de ingedrukte toets in de accu. Via CPM#1Ø wordt er vervolgens getest of de CLEAR-toets soms is ingedrukt (dat is dus AD). De instruktie CPM#12 test of + is ingedrukt. Gaat het niet om deze stuurtoetsen dan blijven cijfertoetsen (Ø ... 9) of = als enige over. In SHIFT (figuur 21a) wordt dan het getal de displaybuffer ingeschoven en via KEYDIS zichtbaar gemaakt.
Is CLEAR = AD ingedrukt, dan springt het programma naar CLEAR1. Is daarentegen + ingedrukt, dan gaat het hoofdprogramma verder op het deel met de naam PLUS, waarbinnen wordt gesprongen naar de subroutine STO1 (figuur 21h). Wat doet STO1? Die verhuist de inhoud van de displaybuffers (= het eerste getal) naar de buffers B12-B11-BØ. In de displaybuffers kan nu het tweede getal worden gezet, nadat ze via de subroutine CLDISP (figuur 21f) zijn klaargezet voor nieuw gebruik. Voor het tweede getal is programmadeel SECOND bedoeld: Wèèr eerst KEYDIS. Na terugkeer daaruit zit de toetswaarde van de ingedrukte toets in de accu. Is het een cijfertoets, dan wordt met SHIFT2 het tweede ingegeven getal, in de display-buffers geschoven. Weer wordt gekeken of het niet CLEAR of + is die was ingedrukt. Was het CLEAR , dan wordt het laatst ingegeven getal uitgewist via CLDISP.
Er is nog een andere mogelijkheid: toets = (DA) is ingedrukt. Dit be-



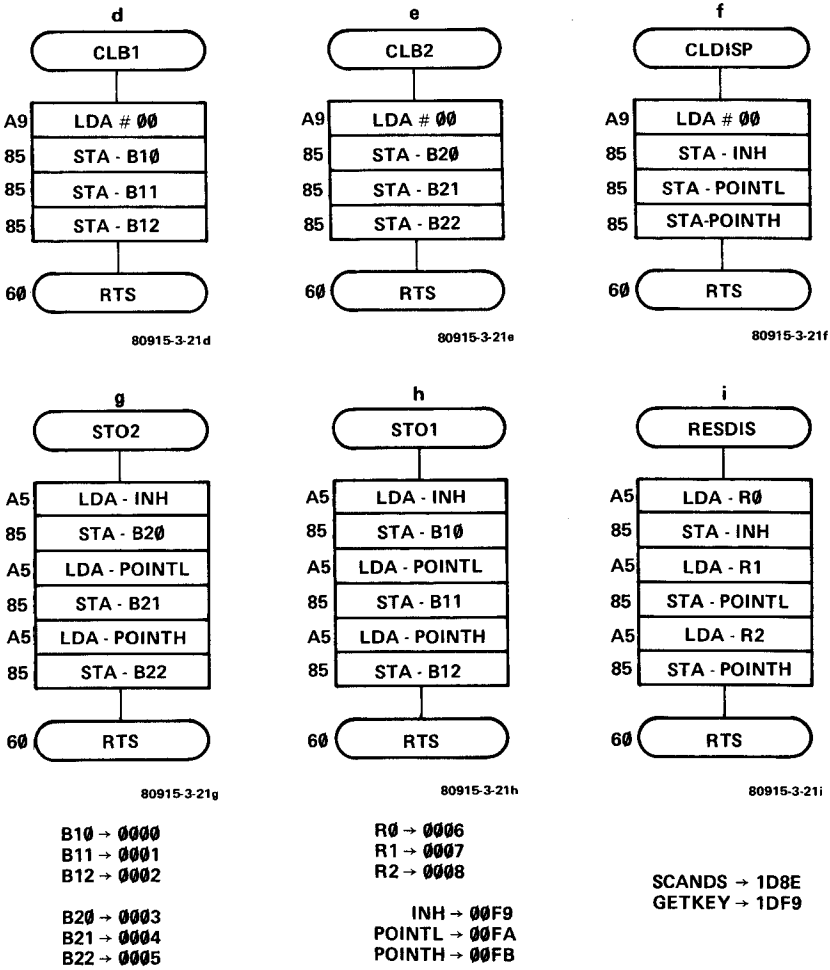
Figuur 21a t/m 21i. Negen subroutines (deelprogramma's), horend bij het programma van figuur 20.

tekent dat men kennelijk klaar is met het ingeven van het tweede getal en dat dus het eigenlijke optellen kan geschieden. In dat geval is programmadeel EQUAL aan bod. Eerst gaat de inhoud van de display-buffers via STO2 (figuur 21g) de buffers voor het tweede getal in. Via ADD (figuur 21b) worden de twee getallen decimaal opgeteld; het resultaat komt in de buffers R2-R1-R0. Rest nog om de inhoud van de resultaatbuffers met RESDIS (figuur 21i) de display-buffers in te sturen en weer te geven via KEYDIS, en om met CLB1 en CLB2 de getalbuffers klaar te zetten voor nieuw gebruik.
Uit dit niet helemaal uitgewerkte programma (het toetsprogramma komt in hoofdstuk 4) blijkt dat het gebruik van subroutines de zaak overzichtelijker maakt. Of in ieder geval minder onoverzichtelijk.

Absolute Indexed, X Addressing

Absolute X-geïndexeerde adressering

Als eerste van de geïndexeerde adresseringsmogelijkheden komt de boven genoemde aan de orde. Inderdaad, een hele mondvol, de terminologie wordt er niet eenvoudiger op. Het programmeren wèl.
Wat was absolute adressering ook weer? Instrukties met absolute adressering bestaan uit drie bytes. Het eerste byte is voor de opcode van de instruktie; de volgende twee bytes (ADL, ADH) geven het adres waar de data is waarop de instruktie wordt losgelaten.
Bij absolute X-geïndexeerde adressering gaat het om een variant van



absolute adressering. Ook hier drie bytes: het eerste byte voor de opcode, het tweede voor de ADL en het derde voor de ADH van een adres. Dit adres is echter in het algemeen niet het adres met de operand-data van de instruktie. Waar dan wèl? De operand-data van de instruktie is te vinden op het adres dat gelijk is aan het opgegeven adres *plus* de inhoud van het X-register.

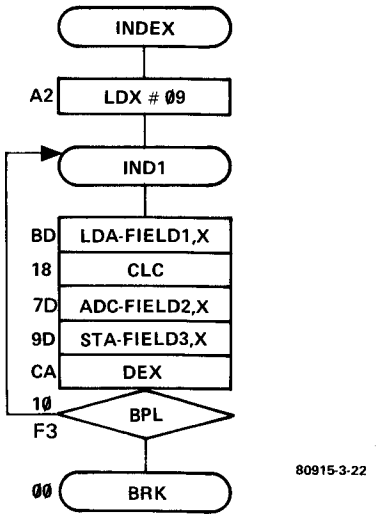
Vergelijk het met de huiselijke betekenis van een adres. De post is niet geadresseerd op "Dorpsstraat 106", maar op "Dorpsstraat 100, drie huizen verder" (even nummering). De drie is te vergelijken met de inhoud van het X-register.

En wat is nu eigenlijk het nut van de X-indexering? Om dat uit de doeken te doen moeten we een uitstapje maken. De velden in. Een veld (field) is een stel opeenvolgende geheugenplaatsen die bedoeld zijn voor te bewerken (operand-)data of voor data als resultaat van een bewerking. De

X-indexering is buitengewoon handig als het om meerdere velden gaat, waarbij een aantal velden operand-data bevat en een aantal velden resultaat-data. Bewerkingen hebben betrekking op overeenkomende posities in elk der operand-velden; op overeenkomende posities in de resultaat-velden komt het resultaat van de bewerkingen. Denk daarbij maar gewoon aan *tabellen.* Een voorbeeld met drie velden FIELD1, 2 en 3:

FIELD1:	Ø12Ø	FIELD2:	ØØ11	FIELD3:	Ø3ØØ
Ø12Ø	Ø1	ØØ11	1Ø	Ø3ØØ	11
Ø121	Ø2	ØØ12	2Ø	Ø3Ø1	22
Ø122	Ø3	ØØ13	3Ø	Ø3Ø2	33
Ø123	Ø4	ØØ14	4Ø	Ø3Ø3	44
Ø124	Ø5	ØØ15	5Ø	Ø3Ø4	55
Ø125	Ø6	ØØ16	6Ø	Ø3Ø5	66
Ø126	Ø7	ØØ17	7Ø	Ø3Ø6	77
Ø127	Ø8	ØØ18	8Ø	Ø3Ø7	88
Ø128	Ø9	ØØ19	9Ø	Ø3Ø8	99
Ø129	ØA	ØØ1A	AØ	Ø3Ø9	AA

De velden 1 en 2 bevatten getallen die regelsgewijs bij elkaar worden opgeteld; het resultaat staat op dezelfde regel in veld 3. De velden worden gekenmerkt door het beginadres. Welke regel op een bepaald moment aan de orde is hangt af van de inhoud van het X-indexregister.



Figuur 22. Programma voor het optellen van een getal 1 uit veld 1 bij een getal 2 op een overeenkomstige plaats van veld 2, met het resultaat op dezelfde positie op veld 3. Er is sprake van getabelleerd optellen.

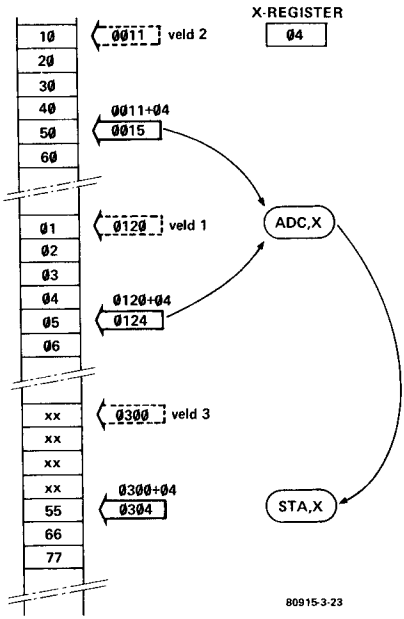
In figuur 22 staat het optelprogramma volgens de tabel, onder gebruikmaking van absolute X-geïndexeerde adressering. Het eigenlijke optellen vindt plaats in het programmadeel IND1. Eerst wordt de accu geladen met

de inhoud van een bepaalde geheugenplaats van FIELD1. Na een CLC (carry C = Ø) wordt de inhoud van de overeenkomstige plaats van FIELD2 daarbij opgeteld (ADC); het resultaat van de optelling gaat de overeenkomstige geheugenplaats van FIELD3 in (STA).
Het programma begint met het laden van X met Ø9. Dan de al beschreven optelling en dan wordt de inhoud van X met één verlaagd. De eerste optelling heeft dus betrekking op adres Ø12Ø+Ø9=Ø129 van FIELD1, ØØ11+Ø9=ØØ1A van FIELD2 en Ø3ØØ+Ø9=Ø3Ø9 van FIELD3. Met andere woorden: Er wordt begonnen op de onderste regel van de tabel en van beneden naar boven gewerkt. Een opmerking over de instruktie-notatie. Neem als voorbeeld LDA-FIELD1,X. Eerst de mnemonics: LDA, gevolgd door het streepje, dat duidt op absolute adressering. Dan de naam van het veld, een komma en een X. In termen van bytes: eerst de opcode, dan ADL en ADH van het eerste adres van het betreffende veld.
Overigens: de optelling volgens de tabel kan ook "ouderwets", bijvoorbeeld met absolute adressering plaatsvinden:

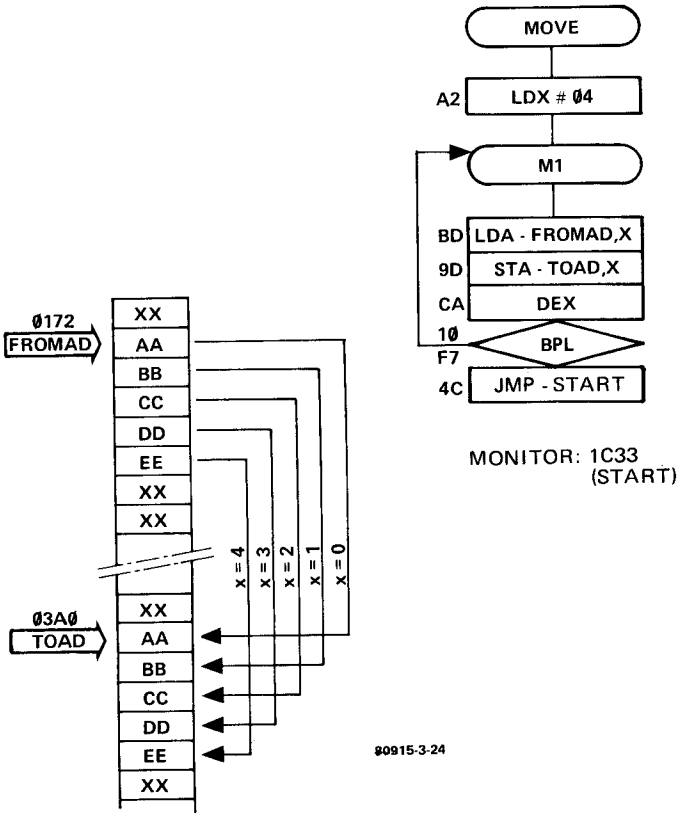
eerste byte:	LDA-opcode
tweede byte:	ADL van te laden data
derde byte:	ADH van te laden data
vierde byte:	CLC (carry C = Ø)
vijfde byte:	ADC-opcode
zesde byte:	ADL van bij accu-inhoud op te tellen data
zevende byte:	ADH van bij accu-inhoud op te tellen data
achtste byte:	STA-opcode
negende byte:	ADL van op te bergen accu-inhoud
tiende byte:	ADH van op te bergen accu-inhoud

En dan hebben we nog maar een keer opgeteld, dus een regel van de tabel uitgevoerd. Bij een korte tabel van het voorbeeld, met velden van tien geheugenplaatsen vergt de eigenlijke opteloperatie al 10x10=100 geheugenplaatsen. Vergelijk dat nou eens met het programma van figuur 22: met kop en staart zijn er 16 geheugenplaatsen nodig. Zie daar het grote voordeel van indexering: minder geheugenplaatsen. Trouwens, ook veel overzichtelijkere programma's.
Figuur 23 verschaft nadere visuele inzichten van het programma van figuur 22. Het gaat om een momentopname bij de stand Ø4 van het X-indexregister.
De instrukties met absolute X-geïndexeerde adressering zijn terug te vinden in het instruktie-overzicht achterin dit boek.
Nu nog een ander voorbeeld waaruit blijkt dat geïndexeerde adressering een uitkomst is. Het programma (MOVE) staat in figuur 24. Het is de bedoeling dat data wordt verhuisd van het ene veld (FROMAD, adres Ø172) naar het andere (TOAD, adres Ø3AØ). Men spreekt van het transport van een *datablok*. We zien dat bij het transport de accu als tussenstation fungeert: LDA-, gevolgd door STA-. Het gaat om vijf data. Het X-register krijgt de waarde Ø4 (LDX#Ø4); na een datatransport (eerst adres Ø172+Ø4=Ø176) wordt de inhoud ervan met 1 verlaagd. Dat gaat zo door tot en met X = Ø; dan is alle data verhuisd, of juister: gekopieerd. Zodra de inhoud van X negatief is is aan de voorwaarde voor springen (BPL) niet meer voldaan en stopt het programma. We toetsen als volgt:

```
AD                    xxxx   xx
 Ø   2   Ø   Ø        Ø2ØØ   xx
DA       A   2        Ø2ØØ   A2    LDX#
 +       Ø   4        Ø2Ø1   Ø4
 +       B   D        Ø2Ø2   BD    LDA-,X
 +       7   2        Ø2Ø3   72    ADL van FROMAD
 +       Ø   1        Ø2Ø4   Ø1    ADH van FROMAD
 +       9   D        Ø2Ø5   9D    STA-,X
 +       A   Ø        Ø2Ø6   AØ    ADL van TOAD
 +       Ø   3        Ø2Ø7   Ø3    ADH van TOAD
 +       C   A        Ø2Ø8   CA    DEX
 +       1   Ø        Ø2Ø9   1Ø    BPL
 +       F   7        Ø2ØA   F7    offset
 +       4   C        Ø2ØB   4C    JMP
 +       3   3        Ø2ØC   33    ADL van START (monitor)
 +       1   C        Ø2ØD   1C    ADH van START (monitor)
AD                    Ø2ØD   1C
 Ø   2   Ø   Ø        Ø2ØØ   A2
GO                    Ø2ØØ   A2    programmastart
```



Figuur 23. Een momentopname van het programma van figuur 22, voor de situatie: X=Ø4.



Figuur 24. Programma voor het verhuizen van een datablok.

Aan het eind van het programma springen we naar de monitor (adres 1C33; zie hoofdstuk 7, boek 2); na de uitvoering van het programma verschijnt het startadres in beeld. Het X-register bestaat uit acht bits. Er kunnen dus datablokken van maximaal 256 bytes worden getransporteerd, zij het niet met het programma van figuur 24. Hiermee zijn namelijk maximaal 128 (beginwaarde X = 7F) bytes te verhuizen. Voor X = 8Ø en hoger is de N-vlag 1 en na één byte-transport leidt de BPL ons meteen terug naar de monitor. Remedie: begin met X = ØØ en vervang de BPL door een BNE (zie ook figuur 27).

Absolute Indexed, Y Addressing

Absolute Y-geïndexeerde adressering

Nou, daar zijn we gauw mee klaar. Vervang in het voorgaande hoofdstuk X door Y. Dat was het. Er zijn dus tegelijkertijd twee indexen beschikbaar.

Zero Page Indexed, X Addressing

X-geïndexeerd adresseren op pagina nul

Adresseren op pagina nul is een variant van absolute adressering en X-geïndexeerd adresseren op pagina nul (zucht) is een variant op absolute X-geïndexeerde adressering. In beide gevallen hoeft alleen het rechter adresbyte ADL van het operand-adres te worden opgegeven; het linker adresbyte ADH ligt vast: ØØ. Met als voordeel dat de instruktie twee bytes lang is in plaats van drie. Ook dat komt bekend voor. Want voor het overige verloopt de indexering identiek aan de procedure bij absolute adressering.

Zero Page Indexed, Y Addressing

Y-geïndexeerd adresseren op pagina nul

Zie boven, met Y in plaats van X. Uit het instruktie-overzicht blijkt dat niet alle X-versies een Y-versie hebben, en omgekeerd.
We hebben nu vier vormen van geïndexeerd adresseren besproken. Er zijn twee registers ten allen tijde beschikbaar voor de adressering. Het ging tot nu toe om vormen van direkte (absolute) adressering. Daar komt nu verandering in.
Bij geïndexeerd adresseren op pagina nul wordt geen rekening gehouden met een evt. carry als gevolg van het indexeren. Dat zou namelijk pagina-overschrijding (naar pagina Ø1) betekenen!

Indirekte adressering

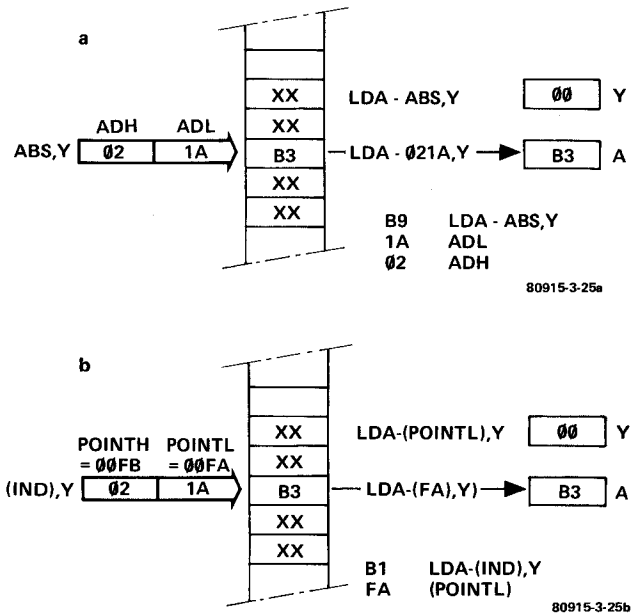
Indirekte adressering is geen eenvoudige zaak om uit te leggen. We hebben er zo lang mogelijk mee gewacht, maar nu is het dan zo ver. Hebben we het echter door, dan wordt het programmeren een stuk eenvoudiger; korter en overzichtelijker.
Een groot voordeel van indirekte adressering is verder dat adressen waarin data wordt opgeborgen of waaruit data wordt gehaald in het begin van de programmeerfase niet meteen vast hoeven te liggen. Het is zelfs zo dat deze adressen kunnen worden berekend door de computer. Met name aan deze adresseringsmetode is het grote succes van de 6502-microprocessor te danken.

Indirect Indexed Addressing (Post-Indexed Indirect Addressing[1])

Indirekte Y-geïndexeerde adressering

Eerst een herhaling. In figuur 25a is aangegeven hoe met behulp van Y-geïndexeerde adressering de accu wordt geladen met de inhoud van de geheugenplaats met adres Ø21A. In het Y-register staat ØØ. Het laad-programmadeel ziet er zo uit:

[1] *Indirekte adressering, gevolgd door indexering (vergelijk pagina 112).*

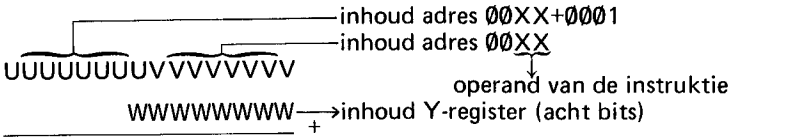


Figuur 25. Het laden van de accu met bepaalde data via absolute Y-geïndexeerde adressering (25a) en via indirekte Y-geïndexeerde adressering (25b).

B9 LDA,Y
1A ADL van laadadres (operand)
Ø2 ADH van laadadres (operand)

Hierdoor komt de inhoud van adres Ø21A (B3) in de accu terecht. Immers, Y=ØØ, dus het operand-adres is Ø21A+ØØ=Ø21A. Allemaal "oude koek" inmiddels. Nou ja . . . Y in plaats van X.
Van essentieel belang bij de laadoperatie van figuur 25a is dat het operand-adres bekend moet zijn. In figuur 25b, met indirekte geïndexeerde adressering, met (een deel van) een adres als operand is de data waarop de instruktie moet opereren *niet* aanwezig op het operand-adres. Waar dan wel? Die data is aanwezig op een geheugenplaats waarvan het adres zich bevindt op de geheugenplaatsen met het operand-adres.
We moeten nu donders goed uitkijken en onderscheid maken tussen het *operand-adres* en het *effektieve adres*; de twee geheugenplaatsen die bij het operand-adres horen bevatten het (effektieve) adres van de geheugenplaats waar de door de instruktie te bewerken data zich bevindt.
Terug naar figuur 25b. De instruktie die het laden van de accu met de inhoud van geheugenplaats Ø21A tot gevolg heeft is een instruktie met indirekte geïndexeerde adressering. De notatie is als volgt:mnemonics-(IND),Y; op de plaats van IND komt informatie over het operand-adres, dat in dit verband ook wel het *indirekte adres* wordt genoemd. In het geval van figuur 25b dus LDA-(FA),Y.
Tussen de haakjes staat FA; dat is het rechter byte van het indirekte adres.

Het linker byte van het indirekte adres is ØØ; de geheugenplaats bevindt zich namelijk op pagina nul. Het adres ØØFA noemen we POINTL. Op dat adres ligt het rechter byte van het effektieve adres. Het linker byte van het effektieve adres ligt één geheugenplaats verder, op adres ØØFB, eveneens op pagina nul; dit adres is POINTH gedoopt.
We weten van figuur 25a dat het effektieve adres Ø21A moet zijn, dus de inhoud van POINTL is 1A en van POINTH Ø2.
We zien dat achter de opcode van de instruktie (in ons geval LDA-) FA staat. Dus slechts één van de vier bytes van de twee adressen waar het effektieve adres ligt opgeslagen hoeft worden opgegeven. Twee van de vier zijn automatisch bekend vanwege de adressering op pagina nul. De derde moet achter de opcode worden gespecificeerd (FA). De vierde is heel simpel één geheugenplaatsje hoger dan de derde; de computer weet dat hij het daar moet zoeken, en zal vinden.
Dat is nog niet alles. Het gaat niet alleen om indirekte maar ook om geïndexeerde adressering. Over indexering op zich hoeven we het niet meer te hebben. Wel over hoe dat hier in zijn werk gaat.
De twee opeenvolgende geheugenplaatsen op pagina nul bevatten het effektieve adres. Hebben we gezegd. Dat is alleen waar als de inhoud van register Y ØØ is. En dat is ie in figuur 25b. Het effektieve adres volgt namelijk uit de optelling van het adres in de twee geheugenplaatsen op pagina nul bij de inhoud van het Y-register.
Je zou het ook zo kunnen zeggen:



waarbij een eventuele carry als gevolg van de optelling van bytes V en W (pagina-overschrijding) wordt opgeteld bij byte U.
Ofschoon voor het vastleggen van het effektieve adres in feite twee adressen nodig zijn (namelijk ØØXX en ØØXX+ØØØ1) spreken we van *het* indirekte adres in plaats van *de* indirekte adressen. Op het indirekte adres ligt een direkt oftewel absoluut adres, dat na rekening (optelling) te hebben gehouden met Y overgaat in het effektieve adres. Instrukties met indirekte Y-geïndexeerde adressering zijn twee bytes lang. Eerst de opcode, dan het rechter byte van een adres op pagina nul.
Let op! Als IND van (IND),Y gelijk is aan FF zal de inhoud van ØØFF (ADL) en ØØØØ (ADH) als uitgangspunt dienen voor de indexering. Dus niet de inhoud van ØØFF en Ø1ØØ!

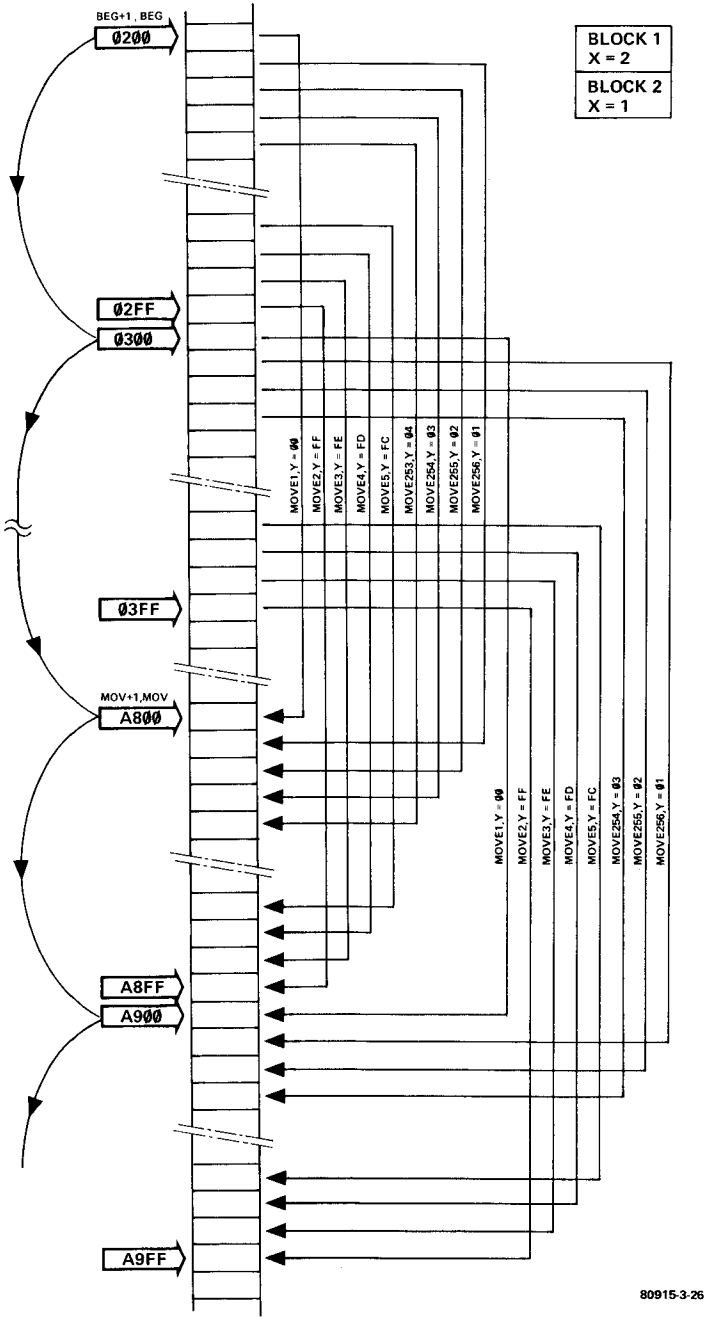
Een toepassing: dataverhuizing in het groot

Onlangs nog, in het hoofdstuk over absolute X-geïndexeerde adressering hebben we het gehad over het transport van een datablok van het ene veld naar het andere. Onder gebruikmaking van indexering was een blokverhuizing van maximaal 256 bytes mogelijk. Heel leuk, maar vaak heerst er een

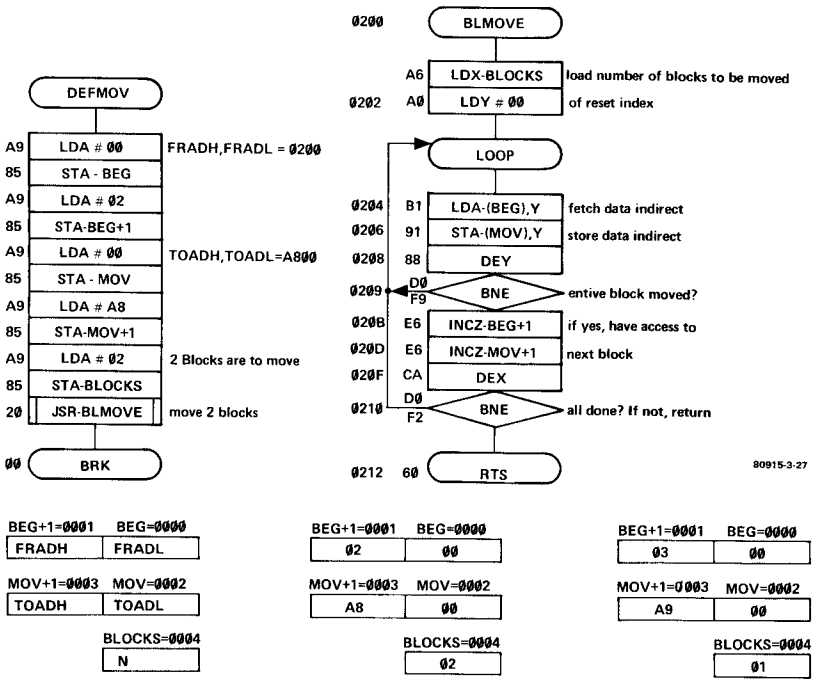
grotere verhuiswoede en is 256 bytes verre van voldoende. Wat te denken van een programma BLMOVE, waarmee maximaal 255 datablokken van elk 256 bytes, dus van 255x256=65.280 bytes mogelijk is? Niet gek. En bovendien een goed voorbeeld van wat er mogelijk is met indirekte Y-geïndexeerde adressering. Want daar hebben we het nog steeds over.
Als ouverture van de eigenlijke programmabespreking eerst figuur 26, waarin het transport van twee datablokken van een kwart k in kaart is gebracht.
Het eerste adres van het eerste datablok is Ø2ØØ, het laatste Ø2FF, 255 geheugenplaatsen verder. Het beginadres komt te staan op twee adressen op pagina nul; op adres BEG ØØ en op adres BEG+1 Ø2. Het eerste datablok (dat overigens op pagina Ø2 staat) moet worden verhuisd naar pagina A8, met beginadres A8ØØ en eindadres A8FF. Het eerste adres van de bestemming van het datablok komt eveneens te staan op twee adressen op pagina nul; op adres MOV ØØ en op adres MOV+1 A8. De diverse adressen op pagina nul met de namen van de op het adres verblijvende bytes zijn:

BEG	op adres ØØØØ. Inhoud heet FRADL en is gelijk aan ØØ
BEG+1	op adres ØØØ1. Inhoud heet FRADH en is gelijk aan Ø2
MOV	op adres ØØØ2. Inhoud heet TOADL en is gelijk aan ØØ
MOV+1	op adres ØØØ3. Inhoud heet TOADH en is gelijk aan A8
BLOCKS	op adres ØØØ4. Inhoud heet N en is gelijk aan Ø2 (= aantal te verhuizen datablokken)

Die namen zitten logisch inelkaar, dachten we: FR slaat op FRom oftewel "van", en TO is in gewoon nederlands "naar"; ADL (rechter byte) en ADH (linker byte) zijn zeer vertrouwd. Hopen we.
Het laden van de genoemde niet-anonieme geheugenplaatsen kan met de hand (toetsenbord) gebeuren. Een speciaal programma ervoor is natuurlijk nog veel leuker. Vandaar DEFMOV, een programma (fig. 27) dat de twee indirekte adressen BEG & BEG+1 en MOV & MOV+1 laadt met de juiste data. Nadat de eigenlijke taken van DEFMOV zijn verricht (vijf keer achterelkaar LDA, gevolgd door STA) volgt een sprong naar de subroutine BLMOVE (figuur 27), waardoor de datablokken van hun plaats komen. Nu meer bijzonderheden.
Het X-register is geprogrammeerd als blokteller; het wordt geladen met de inhoud van BLOCKS. Het Y-register doet dienst als echt index-register; de inhoud ervan geeft aan hoe ver de verhuizing van data binnen een blok is gevorderd. Is de inhoud van Y 21, dan is de 33-ste geheugenplaats van het blok aan bod. Aan het begin van BLMOVE is Y geladen met ØØ. Het transport van data verloopt met de accu als tussenstation; de hierbij betrokken instrukties LDA en STA zijn de versies met indirekte geïndexeerde adressering.
Na een datatransport wordt de inhoud van Y met 1 verlaagd (DEY); Y doorloopt achtereenvolgens de waarden ØØ, FF, FE, . . . , Ø2, Ø1, ØØ, FF, FE, enzovoorts. Via een spronginstruktie BNE test het programma of alle bytes van een blok al zijn verhuisd. Is dat het geval, dan wordt de inhoud van BEG+1 en MOV+1 met een verhoogd; de inhoud van de blokteller X is na DEX eentje lager geworden. Het programmadeel LOOP wordt dus 256 x de inhoud van BLOCKS keer doorlopen. Daarna gaat het met een RTS terug naar DEFMOV en volgt ter plaatse een BRK: het programma is klaar.



Figuur 26. Het verhuizen van twee blokken van elk 256 bytes in beeld gebracht. Het programma staat in figuur 27.



Figuur 27. Programma waarmee maximaal 255 blokken van 256 bytes kunnen worden verhuisd. Er is gebruik gemaakt van indirekte Y-geïndexeerde adressering.

Nu nog wat meer details. Zoals al gezegd bevatten indirekte adressen zoals BEG & BEG+1, en MOV & MOV+1 direkte adressen die met inachtname van de inhoud van het Y-indexregister van belang zijn voor de transport-instrukties binnen het programmadeel LOOP. Het gaat telkens om twee opeenvolgende adressen op pagina nul. Reden genoeg voor de introduktie van een nieuwe kreet: de *address pointer ofwel adreswijzer.* Dat is een patroon van zestien bits dat ontstaat door het aanelkaar plakken van twee bytes, de inhoud van het indirekte adres. De wijzer staat gericht op het direkte ofwel absolute adres dat wordt gevormd door de inhoud van het indirekte adres. Er zijn verschillende (soorten) wijzers in het spel. Kijk maar naar figuur 26. De pointers BEG+1, BEG en MOV+1, MOV (u ziet, de *notatie van een pointer* bestaat uit de twee adresnamen, gescheiden door een komma) wijzen naar het beginadres van een veld. Deze wijzers veranderen sprongsgewijze 256 posities (namelijk nadat een datablok is veranderd). Tijdens het afwerken van een blok blijven deze adreswijzers op hun plaats; ze hebben een min of meer statisch karakter.
Daarnaast zijn er wat we zouden willen noemen dynamische adreswijzers. Deze staan gericht op adressen die afhankelijk zijn van de toestand van het Y-register, en die veranderen na elk transport van een byte. In figuur 26 zijn er dynamische pointers bij het laatste adres van de diverse velden (02FF, 03FF, A8FF en A9FF). Dat levert misschien verwarring op. Er zijn

geen vier dynamische pointers maar twee; eentje voor het vertrekveld en eentje voor het veld van aankomst. Er staan dus op een bepaald moment dynamische pointers op Ø2FF en A8FF en op een ander moment op Ø3FF en A9FF.
De dynamische pointer van het vertrekveld wijst op het adres met de eigenlijke operand van de instruktie LDA; die van het aankomstveld op het adres met de eigenlijke operand van de instruktie STA.

Indexed Indirect Addressing (Pre-Indexed Indirect Addressing[1])

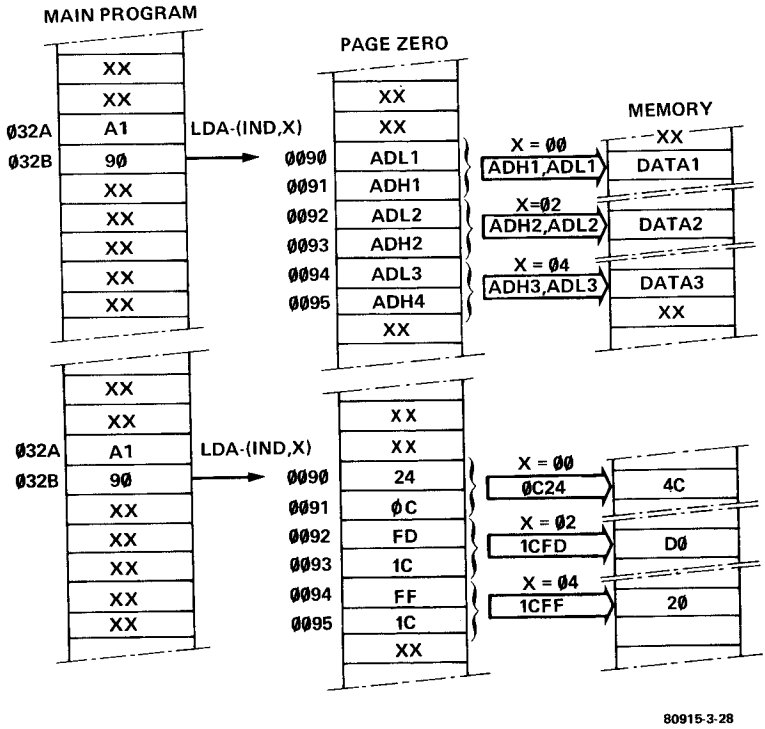
X-geïndexeerde indirekte adressering

Let op het verschil met de vorige adresseringsmetode: "indirekt" en "geïndexeerd" hebben stuivertje gewisseld en "Y-" is vervangen door "X-". Dat stuivertje wisselen betekent indexering bij het vaststellen van het indirekte adres; het indirekte adres bevat het effektieve adres, dat de eigenlijke operand-data van de instruktie bevat.
Het naadje van de kous: ook hier gaat het om instrukties van twee bytes lang. Op de opcode volgt een byte van een adres op pagina nul. Maar nu komt het grote verschil: het eigenlijke indirekte adres ontstaat uit het tweede byte van de instruktie en uit de inhoud van het X-register: bij dat tweede byte wordt het byte opgeteld dat de inhoud van het X-indexregister voorstelt; een eventuele carry wordt genegeerd, dat wil zeggen er wordt letterlijk geen rekening mee gehouden. We hebben dan het rechter byte van de eerste helft van het indirekte adres, waarvan het linker byte ØØ is vanwege de adressering op pagina nul. De tweede helft van het indirekte adres (dat het linker byte bevat van het effektieve adres) volgt automatisch: het is het adres van de geheugenplaats één adres hoger op pagina nul.
De overeenkomsten en verschillen tussen de beide vormen van indirekte en geïndexeerde adressering zijn:

- Bij indirekte Y-geïndexeerde adressering ligt het indirekte adres direkt vast via het tweede byte van de instruktie; het effektieve adres ontstaat indirekt uit de inhoud van het indirekte adres, namelijk met inachtname van de inhoud van het Y-register (indexering).
- Bij X-geïndexeerde indirekte adressering volgt het indirekte adres indirekt uit het tweede byte van de instruktie, namelijk met inachtname van de inhoud van het X-register (indexering); het effektieve adres volgt direkt uit de inhoud van het indirekte adres.

Tja, om nou te zeggen dat dit er allemaal ingaat als gesneden koek . . . Maar misschien zeggen de daden van een programmavoorbeeld meer dan bovenstaande woorden. Het programma van figuur 28 laat zien hoe een laadoperatie met de instruktie LDA in zijn werk gaat onder gebruikmaking van X-geïndexeerde indirekte adressering. In de figuur zien we drie stukjes geheugenkaart die elk twee keer (onder elkaar) voorkomen; de bovenste drie betreffen de algemene situatie en onder in figuur 28 is met konkrete datavoorbeelden gewerkt. De linker geheugenkaarten (MAIN PROGRAM) laten een paar geheugenplaatsen zien waar het programma zit, de middelste een deel van pagina nul (PAGE ZERO) en de rechter (MEMORY) geheugenplaatsen waaruit de te laden data moet komen.

[1] *Indirekte adressering, voorafgegaan door indexering (vergelijk pagina 106).*



Figuur 28. Een voorbeeld van X-geïndexeerde indirekte adressering.

Op adres Ø32A van MAIN PROGRAM staat A1, de opcode van LDA. De notatie is: LDA-(IND,X). (Bij indirekte Y-geïndexeerde adressering zou dat zijn LDA-(IND),Y, met opcode B1). Op het volgende adres staat 9Ø. Daarbij wordt opgeteld de inhoud van X. Is deze ØØ, dan is het effektieve adres te vinden op de geheugenplaatsen ØØ9Ø (ADL1) en ØØ91 (ADH1) van pagina nul. De adreswijzer ADH1, ADL1 staat gericht op de plaats in MEMORY met de naam DATA1. Deze DATA1 komt in de accu terecht.
Stel nu X=Ø1. We komen dan op pagina nul op de geheugenplaatsen ØØ91 en ØØ92 terecht ofwel in ADH1 en ADL2: adresbytes die niet bij dezelfde geheugenplaats horen. Dat kan nooit de bedoeling zijn. Dit kan worden voorkomen door de inhoud van het X-register met Ø2 of in ieder geval met even bedragen te wijzigen.
Bij X=Ø2 ligt het effektieve adres in ØØ92 en ØØ93; de adreswijzer ADH2, ADL2 staat gericht op DATA2. Bij X=Ø4 ligt het effektieve adres in ØØ94 en ØØ95; de adreswijzer ADH3, ADL3 staat gericht op geheugenplaats DATA3 van MEMORY.
Het nut van deze adresseringsmetode schuilt in het feit dat nu pagina nul van het werkgeheugen kan worden gebruikt als zogenaamde "pointer look up table", hetgeen amerikaans is voor een "wijzer-overzichtstabel". Dus op

pagina nul kunnen adreswijzers worden vastgelegd, die op bepaalde geheugenplaatsen staan gericht.
Let op! Bij het indexeren treedt geen pagina-overschrijding op. Is de ADL van het effektieve adres te vinden in ØØFF, dan staat ADH in ØØØØ, niet in Ø1ØØ! Dus: 1. IND = 9Ø en X = 8E; ADL in ØØ1E, ADH in ØØ1F. 2. IND = 9Ø en X = 6F; ADL in ØØFF, ADH in ØØØØ.

Onderbreking van het lopende programma: NMI, IRQ en RESET

Interrupts en (nogmaals) indirekte adressering

In het hoofdstukje "mag ik even storen?" van hoofdstuk 1 van dit boek is al een tipje van de sluier opgelicht. Nu gaat de hele sluier weg.
Wat was ook weer een interrupt? Een door de een of andere oorzaak ontstaan signaal dat als gevolg heeft een onderbreking van het programma waar de computer op dat moment mee bezig was. Er wordt gesprongen naar een subroutine die passende aktie onderneemt. Aktie, passend bij de aard en de oorzaak van de interrupt. Nadat dit is gebeurd gaat het terug naar het onderbroken programma. Dat vervolgens wordt voortgezet.
Een huiselijk voorbeeld: Stel u zit in uw luie stoel naar muziek te luisteren. De telefoon gaat. Het (muziek)programma moet worden onderbroken en de volgende interruptroutine wikkelt zich af: opstaan – geluid zacht of uit – telefoon aannemen – gesprek voeren – hoorn op de haak – "return" (RTI) naar muziekprogramma, gevolgd door een return naar de stoel.
Na een interrupt wordt dus een subroutine aangesproken en zoals we bij de behandeling van subroutines hebben gezien wordt alvorens de sprong uit het hoofdprogramma te maken de inhoud van de programmateller PC (dus het terugkeeradres) tijdelijk opgeslagen op de stack (stapel). Bij een subroutine na een interrupt gaat het precies zo. Alleen gaat nu ook het statusregister P de stapel op en veroorzaakt niet de instruktie JSR de sprong naar een subroutine, maar in de meeste gevallen een bepaald elektrisch signaal op een pen van het microprocessor-IC. Dus in het geval van een interrupt maken niet altijd de bytes de dienst uit (de opcode van JSR), maar de meer grofstoffelijke elektrische spanningen; via hardware in plaats van via software. Er zijn twee mogelijkheden om elektrisch in te grijpen in de programmavoortgang:

1. *Interrupt Request, IRQ* ofwel onderbrekingsaanvrage. Verkeert de elektrische spanning op de bijbehorende pen van de microprocessor in een logische situatie die er op wijst dat het lopende programma graag in de rede zou willen worden gevallen, dan *kan* dit ook daadwerkelijk tot een programma-onderbreking aanleiding geven. Kan, omdat het via software mogelijk is om de "aanvrage" te negeren, naast zich neer te leggen. Dat gaat via het beïnvloeden van de interrupt-vlag I door een van de beide instrukties:
   CLI (opcode 58) waardoor I=Ø. Interrupt Enable; eventuele onderbrekingen toegestaan
   of:
   SEI (opcode 78) waardoor I=1. Interrupt Disable; IRQ-onderbrekings-

mechanisme uitgeschakeld, dus via IRQ-pen zijn geen onderbrekingen mogelijk.

2. *Non Maskable Interrupt*, ofwel interrupt waaraan niet aan voorbij kan worden gegaan. Als de bijbehorende ingang van de μP in de juiste toestand verkeert *moet* het lopende programma worden onderbroken en via een subroutine de interrumpeerder op zijn wenken worden bediend.

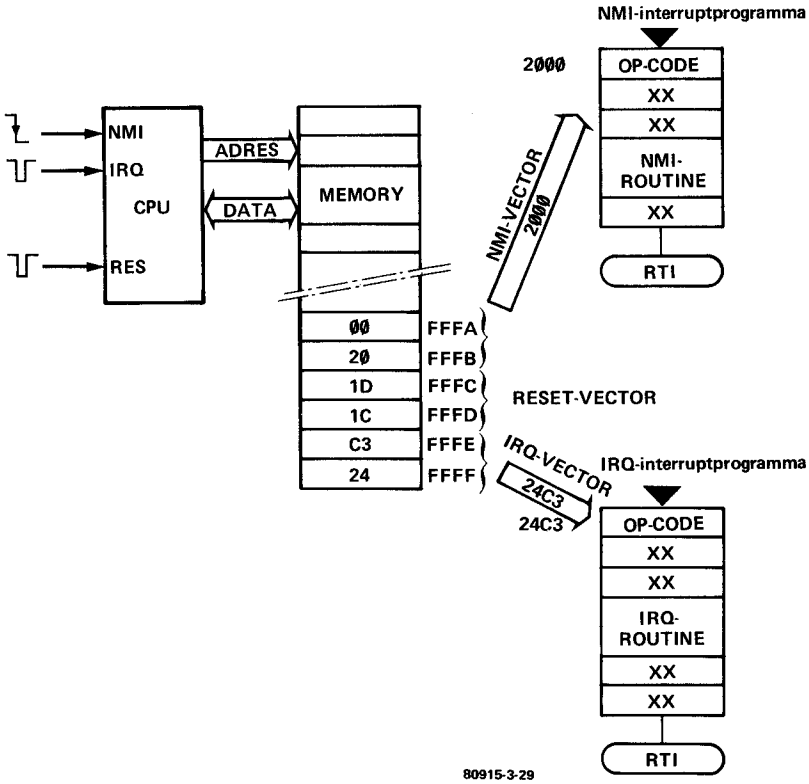
De interrupt-signalen IRQ en NMI zou men evenals het nog te bespreken signaal RES kunnen opvatten als een soort hardware-instrukties of, nog vollediger, hardware-sproninginstrukties, die bij een NMI een onvoorwaardelijk karakter hebben en bij een IRQ een voorwaardelijk karakter. Op het punt van de hardware is er nòg een vermeldenswaardig verschil tussen de beide interrupt-mogelijkheden. Een NMI wordt veroorzaakt tijdens de overgang van logisch 1 naar Ø op de NMI-ingang; dus bij een snelle daling tot 0 volt van de elektrische spanning. Bij een IRQ moet de IRQ-aansluiting logisch Ø zijn, wil er eventueel een onderbreking optreden. Dus een IRQ kan plaatsvinden na de 1/Ø-overgang tot aan de volgende Ø/1-overgang.

NMI-afwikkeling

Zodra een NMI optreedt gaat de microprocessor te rade bij de geheugenplaatsen met adressen FFFA en FFFB; dat wil zeggen nadat de lopende programma-instruktie is afgewerkt. Te rade voor wat? Hij moet weten op welke geheugenplaats de eerste instruktie van de af te handelen interrupt-subroutine op hem ligt te wachten. In figuur 29 is dat adres 2ØØØ. Het rechter byte van dat adres staat op FFFA, het linker byte op FFFB. De *NMI-vektor* is net zo als de al eerder besproken adreswijzers (pointers) een pijl, die staat gericht op het beginadres van de interrupt-subroutine. Net zo als de inhoud van de adreswijzers op plaatsen "onderaan" pagina nul staat vindt men de inhoud van de NMI-vektor op plaatsen onderaan pagina FF.
De interrupt-subroutine wordt instruktie voor instruktie afgewerkt. Subroutines binnen deze subroutine (nesting) zijn ook hier toegestaan. Een normale subroutine werd verlaten met een RTS; bij een interrupt-subroutine gebeurt dat met de instruktie RTI, met opcode 4Ø.

IRQ-afwikkeling

Nadat en zolang de IRQ-aansluiting nul volt is (geworden) is er sprake van interruptie-aanvrage. Wat nu? Ook nu gaat de microprocessor ergens te rade. Kijkt hoe het met de interrupt-vlag I staat. Is deze 1, dan gebeurt er niets: de computer gaat door met waar ie mee bezig was. Is daarentegen I gelijk aan nul, dan wordt de afwerking van de bijbehorende interrupt-subroutine ter hand genomen. Op de geheugenplaatsen FFFE (rechter byte) en FFFF (linker byte) staat het adres waarnaar de *IRQ-vektor* wijst. Dat is dan het startadres van de IRQ-subroutine. In het voorbeeld van figuur 29 is dat 24C3. Ook aan het eind van deze interrupt-subroutine staat de instruktie RTI, die leidt tot het weer opnemen van de draad van het onderbroken programma. Ook binnen de IRQ-subroutine mag subroutine-nesting plaatsvinden.
Zodra een IRQ is aangenomen (de interrupt-vlag I was Ø) wordt I 1 gemaakt. Dit om te voorkomen dat een interrupt-subroutine twee keer achterelkaar wordt uitgevoerd.



Figuur 29. Na een NMI of (soms, want voorwaardelijk) na een IRQ wordt gesprongen naar een interrupt-subroutine, die vervolgens wordt afgewerkt. De NMI-vektor wijst naar het startadres van de NMI-subroutine, de IRQ-vektor naar het startadres van de IRQ-subroutine. Terugkeer uit deze subroutines gaat via de instruktie RTI (*niet* RTS).

Het nul maken van de vlag I kan via de instruktie CLI of RTI. Er moet daarbij op worden gelet dat de IRQ-aansluiting van de microprocessor weer 1 is ("resetten" van de IRQ-lijn) vóór het resetten van de I-vlag; anders wordt de IRQ opnieuw uitgevoerd.
Omdat een IRQ wel te negeren is en een NMI niet is er voor een NMI sprake van een *hogere prioriteit* dan voor een IRQ.

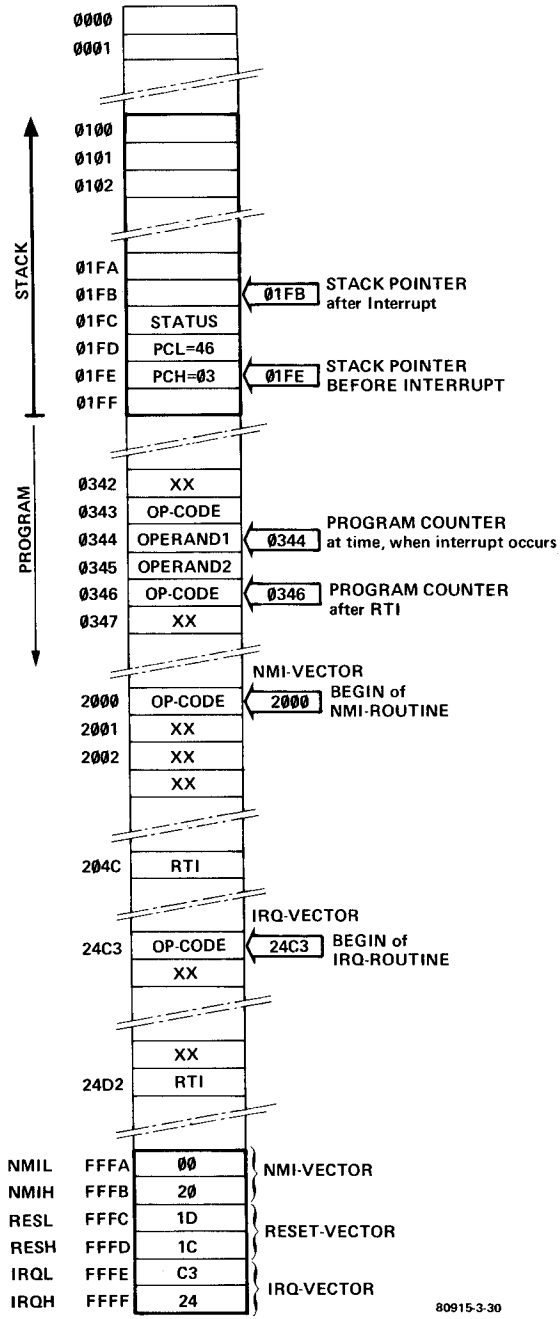
Alles weer bij het oude

Hoe komt het programma na een onderbreking (interrupt) weer aan de voor het onderbroken programma broodnodige informatie, zoals de inhoud van de programmateller PC en die van het statusregister P (het "vlaggenregister")? Het antwoord is heel eenvoudig: net zoals bij de sprong hoofdprogramma-subroutine vice versa. Programmagegevens (inhoud P) gaan evenals het terugkeeradres de stapel op; een RTI zorgt ervoor dat het statusregister P weer is zoals het was, dus dat wijzigingen ten gevolge

van de interrupt-subroutine ongedaan worden gemaakt, en verder dat de eerstvolgende instruktie die aan bod was ook daadwerkelijk aan bod is.

Reset

De computer (microprocessor) kan op nog een derde manier kwa software via hardware worden beinvloed, en dat is via de aansluiting RES van de microprocessor. Als deze pen gedurende een bepaalde minimale tijd logisch nul is springt het programma naar een geheugenplaats die wordt aangewezen door de *reset-vektor*; de adressen van de reset-vektor liggen op de geheugenplaatsen FFFC (rechter byte) en FFFD (linker byte).
Bij de junior-computer wijst de reset-vektor altijd naar het adres 1C1D, dat hoort bij het monitorprogramma (systeem-monitor). Drukt men, bijvoorbeeld na het inschakelen van de junior-computer op de toets RST dan ontstaat via-via een RES voor de microprocessor en gaat het op naar de geheugenplaatsen FFFC en FFFD, en via de daar gevonden reset-vektor naar 1C1D. Zie daar de reden voor resetten na het inschakelen van de junior-computer, zoals we dat in dit hoofdstuk al een paar keer hebben gezien.
De inhoud van de drie vektoren staat op pagina FF. Dat is de hoogst mogelijke pagina. Interessant is de vraag of op de betroffen adressen ook daadwerkelijk geheugenplaatsen zijn "aangesloten". Dat is bij de junior-computer niet het geval. Sla er het hoofdstukje over adresdekodering in hoofdstuk 1 maar op na: slechts de eerste 8 x 4 = 32 pagina's ØØ . . . 1F zijn volledig gedekodeerd.
Gelukkig is het probleem van de niet aangesloten geheugenplaatsen op te lossen. Wat moeten we daarvoor doen? Niets, helemaal niets. Kijk maar: het linker byte van de vektor-geheugenplaatsen is FF. Binair is dat 11111111, die door de adreslijnen (van links naar rechts) A15 . . . A8 worden bepaald. Nu zijn vanwege de zojuist nog genoemde onvolledige adresdekodering de adreslijnen A15, A14 en A13 intern niet met enige hardware doorverbonden en het maakt dan ook geen moer uit of ze Ø of 1 zijn, dus in dit geval is 11111111 gelijk aan XXX11111, bijvoorbeeld ØØØ11111, en dat is 1F. Met andere woorden: de junior-computer legt pagina FF uit als pagina 1F, en deze maakt samen met de pagina's 1C, 1D en 1E deel uit van het 1k-geheugenbereik dat door de EPROM wordt bestreken. En EPROM's zijn ideaal voor het vastleggen van vektoren.
In de vorm van een geheugenkaart (figuur 30) zal nog eens worden nagegaan wat er zo al bij een IRQ en een NMI komt kijken. We zien dat pagina Ø1 is gebruikt als stack; pagina Ø3 bevat het te onderbreken programma. Verder zien we dat de IRQ-vektor wijst naar 24C3, het startadres van de IRQ-subroutine en dat de NMI-vektor wijst op het adres (2ØØØ) waar de NMI-subroutine begint.
Stel het programma is bezig met geheugenplaats Ø343. Er wordt een opcode van een instruktie gehaald. En als een donderslag bij heldere hemel kondigt zich een NMI aan (op zijn amerikaans voluit uitgesproken betekent NMI zo iets als "vijand", "rustverstoorder"). Hoe reageert de computer op deze donderslag? Wel, het is niet de bedoeling om allerlei interne, huishoudelijke akties in detail te beschrijven, maar waar het om gaat is dat het linker byte PCH van de programmateller (Ø3) op de stack komt op die plaats die door de stapelwijzer is aangewezen. Dat is namelijk de eerst-



Figuur 30. Aan de hand van een geheugenkaart is de gang van zaken bij de interrupts (onderbrekingen) in beeld gebracht.

volgende lege plaats op de stapel. Vervolgens wordt de stapelwijzer met 1 verhoogd en gaat het rechter byte PCL de stapel op. Na de daarop volgende verhoging met 1 van de stapelwijzer staat deze gericht op adres Ø1FC. Daar gaat de inhoud van het P-register naar toe. Hierna zoekt de stapelwijzer het weer eentje hogerop en staat dan gericht op Ø1FB.
De volgende fase is de sprong naar de NMI-subroutine; waar die begint ligt vast via de inhoud, de richting van de NMI-vektor; in ons geval 2ØØØ. Ook de I-vlag krijgt de waarde 1, waardoor een eventuele IRQ niet wordt gehonoreerd. Nadat de inhoud van de NMI-vektor in de programmateller is geladen is de interrupt-subroutine NMI aan afwerking toe. En dat gebeurt dan ook. Hoe het zit met de return komt direkt aan de orde.

Stel nu dat het geen NMI was die ten tijde van het afwerken van adres Ø343 ertussen kwam, maar een IRQ. Hoe gaat het dan in zijn werk? Vrijwel identiek. Ook nu gaan achtereenvolgens PCH, PCL en P de stapel op. We zijn er overigens van uitgegaan dat de I-vlag nul was voor het optreden van een IRQ. Via de inhoud van geheugenplaatsen FFFE en FFFF komt "men" aan de weet op welk adres de IRQ-subroutine begint. En ook nu wordt de I-vlag 1 gemaakt, zodat een nieuwe IRQ een roepende in de woestijn is. En een NMI dan? Zo'n onderbreking met "top-priority"? Laat staan meerdere NMI's. Op dit probleem komen we nog terug bij de bespreking van de instruktie met indirekte adressering JMP. Nu volstaat de opmerking dat via *een* NMI-vektor *verscheidene* NMI-subroutines kunnen worden aangeroepen.
Net zo als binnen een subroutine een subroutine kan worden aangeroepen kan binnen een interrupt-subroutine opnieuw een interrupt-routine worden aangesproken (interrupt-nesting).

Nogmaals: alles bij het oude. Oftewel: de terugkeer via een RTI

Aan alles komt een eind, ook aan NMI- of IRQ-subroutines. Als het zover is wijst een instruktie RTI op de terugkeer naar het onderbroken programma. In het voorbeeld van figuur 30 staat een RTI op adres 2Ø4C. Er gebeurt nu van alles. Eerst wordt de stapelwijzer met 1 verlaagd en wijst dus op adres Ø1FC. Daar ligt de inhoud van het P-register en de inhoud van het inmiddels gewijzigde P-register (onder invloed van de interrupt-subroutine) wordt hieraan gelijk gemaakt. Na opnieuw verlagen van de stapelwijzer (Ø1FD) krijgt PCL van de programmateller zijn oude waarde terug; nogmaals met 1 verlagen van de stapelwijzer (Ø1FE) levert de oude waarde van PCH. Daarmee is het terugkeeradres bekend. In het geval van figuur 30 is dat Ø346. We zijn terug in het onderbroken programma. Dat zich vervolgens verder afwikkelt.

Redden wat er te redden valt

Bij de sprong naar een interrupt-subroutine worden het P-register ("vlaggenregister") en de programmateller (terugkeeradres) bewaard, "gered" voor later gebruik. Vaak wil je graag nog meer bewaren. Bijvoorbeeld de inhoud van de interne registers A, X en Y. Dat kan. En wel met "push-"instrukties, die de inhoud van zo'n intern register op de stapel zetten. Ze moeten er dan natuurlijk wèl weer af kunnen worden gehaald. En dat gaat met "pull-"instrukties.

De eigenlijke interrupt-subroutine krijgt aan het begin een paar push-instrukties toegevoegd; aan het eind ervan worden er een paar pull-instrukties aangeplakt. Bijvoorbeeld zò:

```
SAVE   PHA   red accu-inhoud op de stapel
             (waarna de stapelwijzer met 1 wordt verhoogd)
       TXA   breng inhoud X-register naar de accu
       PHA   red accu-inhoud op de stapel
             (waarna de stapelwijzer met 1 wordt verhoogd)
       TYA   breng inhoud Y-register naar de accu
       PHA   red accu-inhoud op de stapel
             (waarna de stapelwijzer met 1 wordt verhoogd)
       . . . eerste instruktie interrupt-subroutine
         .
         .
         .
RESTO  PLA   breng het bovenste van de stapel naar de accu
             (waarna de stapelwijzer met 1 wordt verlaagd)
       TAY   breng accu-inhoud naar Y-register
       PLA   breng het bovenste van de stapel naar de accu
             (waarna de stapelwijzer met 1 wordt verlaagd)
       TAX   breng accu-inhoud naar X-register
       PLA   breng het bovenste van de stapel naar de accu
             (herstel oorspronkelijke accu-inhoud)
             (waarna de stapelwijzer met 1 wordt verlaagd)
END    RTI   keer terug naar het onderbroken programma
```

Er goed op letten dat de IRQ- of NMI-vektor staat gericht op het adres van SAVE, en niet op het eigenlijke startadres van de interrupt-subroutine. Let ook op het feit dat wat als eerste de stapel opging er als laatste vanaf wordt gehaald.

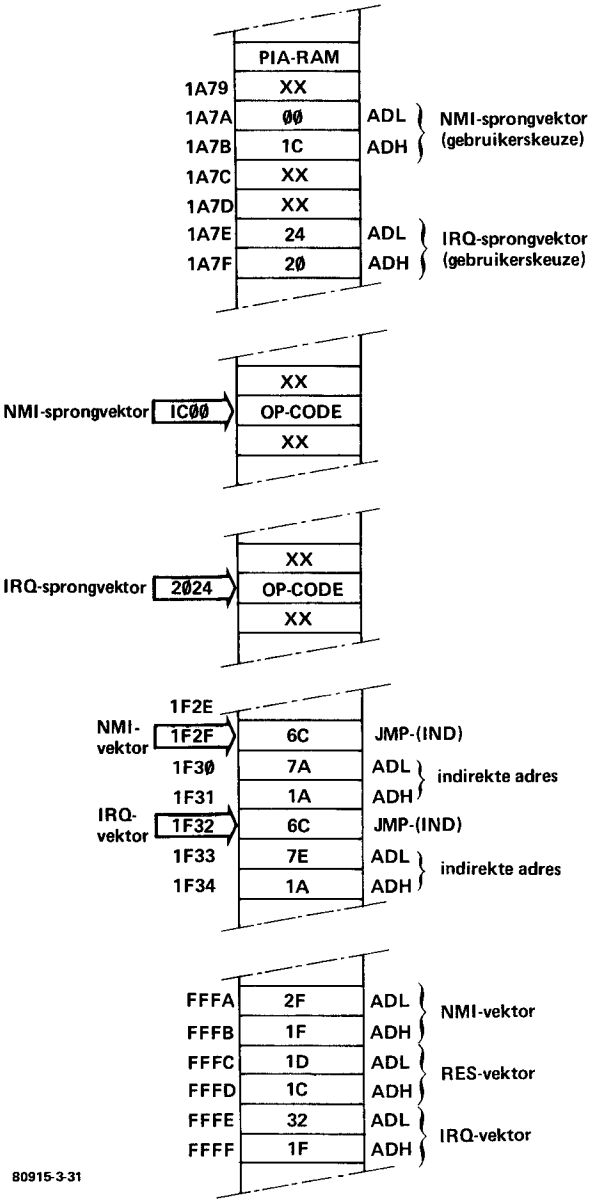
Als laatste, dertiende adresseringsmogelijkheid komt nu de indirekte adressering aan de orde. Het gaat om slechts één instruktie, namelijk JMP-indirekt. Maar wèl erg belangrijk, omdat we nu eindelijk aan de weet komen hoe het komt dat met één NMI verscheidene interrupt-subroutines kunnen worden bereikt.

Indirekte adressering

Na JMP-indirekt alle dertien goed (behandeld)

De instruktie JMP kennen we al. Althans in de versie met absolute (direkte) adressering. Het is dè instruktie om onvoorwaardelijk te springen naar een ander deel van het geheugen, waar dan nieuwe instrukties liggen. Het is een instruktie van drie bytes lang. Het eerste byte is voor de opcode (4C), het tweede voor het rechter byte ADL en het derde voor het linker byte ADH van het adres waarnaar gaat worden gesprongen.

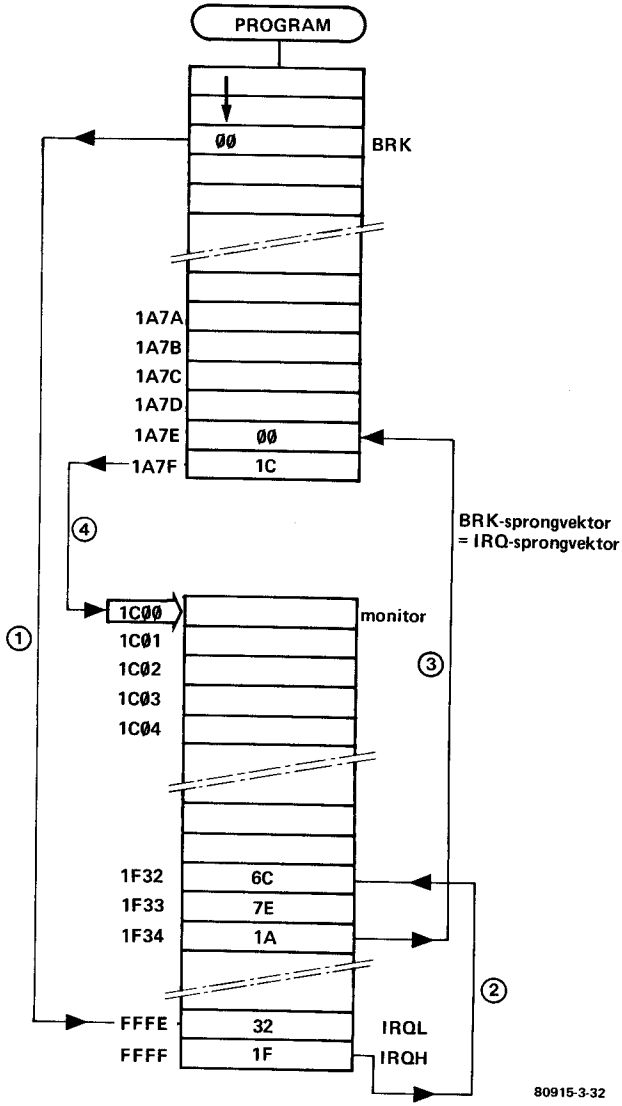
Ook de indirekte versie van JMP vergt drie bytes. Ook hier is het eerste byte gereserveerd voor de opcode (en die is 6C). Het tweede byte van de instruktie bevat de ADL en het derde byte de ADH van een adres; die ADL en ADH bevatten een deel van een *sprongvektor*. Het rechter byte van de sprongvektor staat op de geheugenplaats met adreswijzer ADH, ADL en het linker byte op één geheugenplaats verderop, dus op de plaats met



Figuur 31. Het verwerken ("bedienen" of "servicen") van interrupts via instrukties JMP met indirekte adressering. De inhoud van het indirekte adres wijst op het effektieve (direkte = absolute) sprongadres en is programmeerbaar.

adreswijzer (ADH, ADL) + 1. Daarmee ligt de inhoud van de sprongvektor vast, dus het sprongadres waarop hij staat gericht.
Dus bij een indirekte sprong bekijkt de μP automatisch twee opeenvolgende geheugenplaatsen; de eerste bevat de ADL, de tweede de ADH van de sprongvektor.
De notatie van de instruktie is: JMP- (IND). Dus als we stellen: JMP-(1A7A) (figuur 31), dan staat op adres 1A7A het rechter byte van het sprongadres en op adres 1A7B het linker byte. Er is dus sprake van een indirekte sprong naar de adressen die de sprongvektor vastleggen, en van een *effektieve sprong* naar het sprongadres, waarop de sprongvektor staat gericht.
Wat is het nut van dit indirekte springen bij het afhandelen van een NMI? Laten we dat eens gaan bekijken aan de hand van figuur 31. Stel er is een NMI. De computer kijkt dan op de adressen FFFA en FFFB, op zoek naar de NMI-vektor. Die blijkt 1F2F te zijn. Dat is nu niet het startadres van een subroutine, maar een adres waarop de opcode staat van de instruktie JMP-(IND). De inhoud van IND is 1A7A (= de inhoud van de adressen 1FØØ en 1F31). Het adres waarnaar moet worden gesprongen (waarop de *NMI-sprongvektor* wijst — duidelijk iets anders dan de *NMI-vektor*) ligt op de geheugenplaatsen 1A7A en 1A7B; dat is op pagina 1A.
Pagina 1A maakt deel uit van het RAM-geheugen in de PIA (zie hoofdstuk 1). De NMI-sprongvektor is dus in tegenstelling tot de NMI-vektor (die ligt op pagina FF=pagina 1F in de EPROM) door de gebruiker te specificeren. En dat is hard nodig als via *één* NMI-vektor *verschillende* NMI-subroutines, met verschillende NMI-sprongvektoren moeten kunnen worden bereikt. Wil dat allemaal tot een goed einde komen dan is het van belang dat voordat de NMI optreedt de indirekte adressen met sprongvektoren zijn geladen, dus de adressen 1A7A en 1A7B met een NMI-sprongvektor en de plaatsen 1A7E en 1A7F met een IRQ-sprongvektor.
In het voorbeeld van figuur 31 komt het programma na het optreden van een NMI via de NMI-vektor, een JMP-(IND) en de NMI-sprongvektor terecht op geheugenplaats 1CØØ. Daar ligt de eerste opcode van het monitorprogramma. Na een gehonoreerde IRQ zou het programma via de IRQ-vektor, een JMP-(IND) en de IRQ-sprongvektor terecht komen op geheugenplaats 2Ø24.
De clou van de sprongvektoren in (PIA-)RAM is dat het effektieve sprongadres van een interrupt-subroutine veranderlijk is; hetzij onder invloed van de gebruiker, hetzij onder invloed van (dus tijdens) het programma. Wat dat laatste betreft: een deel van een bepaalde interrupt-subroutine zou daaruit kunnen bestaan dat het indirekte adres wordt geladen met een nieuw effektief (direkt, absoluut) sprongadres dat hoort bij een andere interrupt-subroutine. Vooral het gebruik van BRK's wordt hierdoor buitengewoon interessant.

Let op! Het derde byte XX van JMP-(IND) bepaalt de pagina, waar *beide* adresbytes van de sprongvektor zijn gespecificeerd; het tweede byte bepaalt de positie op die pagina. Is dat namelijk gelijk aan FF, dan staat de ADL van de sprongvektor in XXFF en de ADH in XXØØ!



Figuur 32. De instruktie BRK is de software-versie van de hardware-instruktie IRQ. In deze figuur zijn de vier sprongen aangegeven die optreden zodra ergens in een lopend programma de opcode 00 van BRK wordt herkend. In de standaardsituatie wijst de BRK-sprongvektor (= IRQ-sprongvektor) naar het monitorprogramma (1C00), dat begint met een SAVE-routine die de inhoud van alle interne registers opslaat in de geheugenplaatsen 00EF . . . 00F5.

De uitsmijter: BRK

Programmaonderbrekingen oftewel interrupts waren totnutoe alleen mogelijk via de "hardware-"instrukties NMI en IRQ. Het kan ook via 100% -software. Dus met zo'n ouderwetse instruktie met opcode en zo. Dat is de instruktie BRK, met opcode ØØ, die we al vaak genoeg zijn tegengekomen aan het eind van een programma. Nu eindelijk, aan het eind van dit hoofdstuk 3 zijn we zo ver dat we aan de weet komen hoe de vork precies in de steel zit.

Zodra ergens in het programma een opcode in het geheugen wordt opgehaald en het is ØØ, treedt de IRQ-situatie op. Na een BRK gaat het dus nàar de adressen FFFE en FFFF om aan de weet te komen waarnaar de IRQ-vektor wijst; dus waarheen het programma moet springen.

Wat kun je zo al doen met een BRK? Figuur 32 geeft een voorbeeld. Op een gegeven moment stoot het programma op een BRK. Dat leidt tot het zoeken van de IRQ-vektor op de adressen FFFE en FFFF en tot een sprong naar het adres, aangewezen door de IRQ-vektor. Vervolgens wordt gesprongen naar het indirekte adres, waar de BRK-sprongvektor (= IRQ-sprongvektor) ligt opgeslagen. In het geval van figuur 32 leidt dat tot een sprong naar het monitorprogramma.

Een uiterst belangrijke toepassing van de BRK-instruktie is het "debuggen", foutloos maken van een gebruikersprogramma. We gaan er hier niet in detail op in, wel even globaal.

Stel een programma "doet het niet" in eerste instantie. Deze veronderstelling is realistisch, namelijk gebaseerd op ervaring. De ervaring dat programmeren leren het doorlopen inhoudt van een harde, maar aangename leerschool. Stel nu dat bij kontrole blijkt dat de eerste helft van een programma goed is en dat vanaf een zekere plaats verkeerde instrukties, met van de juiste afwijkende instruktielengte zijn gebruikt, of instrukties vergeten. Je kunt natuurlijk het programma opnieuw intypen vanaf het punt dat het fout gaat. Je kunt echter ook BRK's inlassen die via IRQ's leiden tot een gedeelte van het geheugen (waarvan het startadres moet worden opgegeven op de geheugenplaatsen 1A7E en 1A7F), waar de in te voegen juiste instrukties staan, af te sluiten met een RTI. In wezen is er dus sprake van een soort "wegomlegging".

Vaak zullen er meerdere wegomleggingen, dus meerdere BRK's zijn. Het interruptprogramma van de eerste BRK kan dan voor de RTI worden afgesloten met het laden van 1A7E en 1A7F met de BRK-sprongvektor die hoort bij het interruptprogramma van de tweede BRK, enzovoorts.

Bij het BRK-gebeuren hoort een vlag. Dat is de B-vlag die deel uitmaakt van de vlaggenparade van het P-register. Zodra een BRK optreedt is de B-vlag gezet, 1 gemaakt. Gaat het om een normale hardware-IRQ, dan is de B-vlag Ø. Een hardware-IRQ kan worden genegeerd door het 1 maken van vlag I; de software versie BRK trekt zich niets van I aan. Een BRK is dus onvoorwaardelijk (non-maskable).

N.B. Voor aanvullende informatie over BRK: zie Aanhangsel 3 van boek 2 (pagina 199)

Zo, dat was dan hoofdstuk 3. Het was de laatste bladzijden wel erg veel teorie, en weinig praktijk van programma-voorbeelden. Straks, in hoofdstuk 4 wordt dat heel anders. Want u weet nu al wat interrupts zijn. En heeft alle dertien (adresseringsmogelijkheden) goed (onder de knie).

4

Alle begin is eenvoudig

Programmeren zonder poespas

Het laatste hoofdstuk van dit boek ligt nu voor u. In boek 2 ligt nog veel "lekkers" op u te wachten. Maar voor dat u daar als fijnproever aan toe bent moet u wel eerst de smaak van het programmeren te pakken krijgen. Aan de hand van het basis-menu van hoofdstuk 3. In dit hoofdstuk krijgt u nog wat programmavoorbeelden voorgeschoteld. Want: goed voorbeeld doet goed volgen.

Geprogrammeerde I/O, de monitor-routines, hexadecimaal editen en assembleren – het staat u allemaal nog te wachten, in boek 2. Stuk voor stuk machtig interessante onderwerpen, daar niet van, maar pas nadat u het basis-programmeren, met de kennis van dit en het vorige hoofdstuk onder de knie hebt. Tenslotte beginnen de pianolessen óók niet met de minutenwals van Chopin, maar met "Jan, daar ligt een kip in het water". In ieder geval met iets betrekkelijk eenvoudigs.
Dit hoofdstuk bevat een aantal programma's, elk met globaal en gedetailleerd stroomdiagram en met het toetsenwerk, op basis van de kennis van hoofdstuk 3. En verder praktische tips.

Vingeroefeningen met het decimale optelprogramma. Waarin ook algemene praktische tips en regels

Letterlijk oefenen met de vingers, want er moet achtereenvolgens 594 keer een toets worden ingedrukt en wel zodanig dat het decimale optelprogramma van 196 bytes (geheugenplaatsen) in het werkgeheugen (RAM) komt te staan. Als dit eenmaal is gebeurd, kunnen we iets met dat programma gaan doen. In dit geval het optellen van twee decimale getallen van maximaal zes cijfers.
Het optelprogramma is besproken in hoofdstuk 3. Van belang daarbij zijn de figuren 20 (gedetailleerd stroomdiagram van het hoofdprogramma), 21 (de gedetailleerde stroomdiagrammen van negen subroutines) en 14 (naam en adres van elk der drie display-buffers). Gedetailleerde stroom-

diagrammen zijn de laatste fase in de programma-voorbereiding: elke handeling en beslissing is direkt vertaald in overeenkomstige instrukties. De opcodes en dus de bytes voor de geheugenplaatsen zijn bekend: een kwestie van opzoeken (zie het Aanhangsel). Ook de operand-gegevens zoals data, offset, of adres, die van belang zijn voor de uitvoering ervan zijn al bekend, zij het niet altijd meteen in detail. Dat wil zeggen: de bytes zijn nog niet bekend. Dit zullen we toelichten aan de hand van het toetsprogramma van het programma: decimaal optellen. Dat staat in figuur 1 van dit hoofdstuk. Voordat we dat doen eerst een overzicht.
Het toetsprogramma van figuur 1 bestaat uit de volgende delen:
adressen Ø2ØØ tot en met Ø248 : hoofdprogramma (figuur 20)
adressen Ø249 tot en met Ø258 : subroutine SHIFT (fig. 21a)
adressen Ø259 tot en met Ø26E : subroutine ADD (figuur 21b)
adressen Ø26F tot en met Ø281 : subroutine KEYDIS (fig. 21c)
adressen Ø282 tot en met Ø28A : subroutine CLB1 (fig. 21d)
adressen Ø28B tot en met Ø293 : subroutine CLB2 (fig. 21e)
adressen Ø294 tot en met Ø29C : subroutine CLDISP (fig. 21 f)
adressen Ø29D tot en met Ø2A9 : subroutine STO2 (fig. 21g)
adressen Ø2AA tot en met Ø2B6 : subroutine STO1 (fig. 21h)
adressen Ø2B7 tot en met Ø2C3 : subroutine RESDIS (fig. 21i)

Voorbereiding toetsprogramma

Terug naar het probleem van daarnet van het alles nog niet precies weten. Neem nou adres Ø2Ø6. Daar komt de opcode van JSR te staan. Er moet worden gesprongen naar een subroutine. We weten welke, maar bij het opstellen van het toetsprogramma kennen we het startadres van de subroutine nog niet, omdat we nog niet precies weten hoe we uitkomen met het aantal nodige geheugenplaatsen (= het aantal nodige adressen). Dit brengt ons op twee algemene en zeer praktische regels:

1. *Het verdient sterke aanbeveling om de overgang van gedetailleerd stroomdiagram naar het fysieke toetsprogramma te laten verlopen via een "papieren" toetsprogramma!*
2. *Voor operand-adressen en offset-bytes die nog niet bekend zijn moeten voldoende geheugenplaatsen worden gereserveerd; in tweede instantie, in een later stadium worden de ontbrekende bytes ingevuld.*

Regel twee is belangrijk. Want:

3. *Programma's moeten een aaneengesloten aantal geheugenplaatsen innemen. In het geval van een (hoofd)programma met subroutine(s) geldt e.e.a. ook voor de subroutine(s).*

Deze regel heeft te maken met het voor de microprocessor huishoudelijke feit dat de programmateller (PC) na uitvoering van een instruktie automatisch het adres van de volgende geheugenplaats bevat. En als op de bewuste geheugenplaats geen instruktie wacht op uitvoering is de junior-computer tot werkeloosheid gedoemd. En dat kan natuurlijk nooit de bedoeling zijn.
Lege geheugenplaatsen ontstaan bijvoorbeeld indien tijdens het intoetsen van een programma in de data-mode (d.w.z. van de toetsen AD en DA is DA het laatste ingedrukt) per abuis twee maal achter elkaar de toets + wordt ingedrukt.
Lege plaatsen kan men overigens achteraf vullen met het byte EA, dat is de

opcode van de loze instruktie NOP. Dezelfde instruktie kunnen we op een aantal plaatsen invoegen (al in de papieren fase) om het vervelende effekt van vergeten instrukties en/of operanden, kortom het effekt van vergeten bytes gedeeltelijk teniet te doen. Want niets is vervelender dan het (bijna) overnieuw intoetsen van het programma.
We hebben het al even over fouten gehad. Die kunnen optreden bij het programmeren, dus tijdens de fasen die leiden tot het papieren toetsprogramma, of tijdens het intoetsen zelf. Het is belangrijk dat men na het intoetsen van het programma de mogelijkheid heeft om nog eens na te gaan welk byte er op een bepaald adres staat, of dat men misschien zelfs wel het hele programma nog eens de revu wil laten passeren. Verder is het belangrijk dat men korrekties kan aanbrengen.
Dat kan.
Maar eerst even iets anders. *Zodra men de junior-computer uitschakelt gaan alle programma's, die sinds de laatste keer dat de junior-computer is ingeschakeld, zijn ingetoetst, verloren!* Het is dus een misverstand om te denken dat men – tenzij het apparaat blijft ingeschakeld – weken later nog wel dat ene foutje uit het ingetoetste programma kan halen.

Zijn alle geheugenplaatsen goed bezet?

Nu het bekijken van geheugenplaatsen en het eventuele korrigeren van de inhoud daarvan. Daarbij spelen de toetsen AD , DA en + een grote rol. Wil men een bepaalde geheugenplaats bekijken, dan toetst men als volgt:

AD X X X X

waarbij XXXX het adres is van de geheugenplaats waarvan men de inhoud wil zien. Dit adres staat op de linker vier (adres-)displays; het byte waarmee de geheugenplaats is gevuld staat in hexadecimale vorm op de rechter twee (data-)displays.
Wil men de volgende geheugenplaats bekijken, dus die met een adres één hoger, dan drukt men toets + in. Het adresdisplay toont dan XXXX+ØØØ1, het datadisplay de op dit adres op dat moment gevestigde geheugeninhoud.
Voor het wijzigen van de inhoud van een geheugenplaats zorgt men er eerst voor dat de bewuste geheugenplaats op het display staat. Vervolgens drukt men DA in, en twee keer toetsen levert het nieuwe byte op, dat dan ook op het display verschijnt. Moeten er opeenvolgende geheugenplaatsen worden overschreven, dan kan men dankbaar gebruik maken van de toets + , net zoals dat bij het intoetsen van een programma het geval is. Al met al zijn er voldoende mogelijkheden om na te gaan of het papieren toetsprogramma korrekt is ingebracht, en om eventuele korrekties aan te brengen.
Nog even die kwestie van het nog niet bekend zijn van adres-bytes en offsets, aan het begin van de papieren toetsprogrammafase. Men weet nog niet naar welk adres voorwaardelijk of onvoorwaardelijk moet worden gesprongen, maar wel is bekend welk programmadeel na de sprong aan de orde is. Die programmadelen worden namelijk voorafgegaan door een naam. Of, in computertaal, een *label.* In het toetsprogramma van figuur 1 zijn ze in de rechter helft (de *comment*-helft, waarin wat gegevens over wat je op een bepaald moment eigenlijk aan het doen bent) aangegeven door een rechthoek, met daarin de naam van het beestje. Ook de subroutines

	toetsen			adres	data	
RST				xxxx	xx	
AD				xxxx	xx	
Ø	2	Ø	Ø	Ø2ØØ	xx	
DA		2	Ø	Ø2ØØ	2Ø	JSR- CLEAR1
+		9	4	Ø2Ø1	94	ADL van CLDISP
+		Ø	2	Ø2Ø2	Ø2	ADH van CLDISP
+		2	Ø	Ø2Ø3	2Ø	JSR-
+		8	2	Ø2Ø4	82	ADL van CLB1
+		Ø	2	Ø2Ø5	Ø2	ADH van CLB1
+		2	Ø	Ø2Ø6	2Ø	JSR-
+		8	B	Ø2Ø7	8B	ADL van CLB2
+		Ø	2	Ø2Ø8	Ø2	ADH van CLB2
+		2	Ø	Ø2Ø9	2Ø	JSR- FIRST
+		6	F	Ø2ØA	6F	ADL van KEYDIS
+		Ø	2	Ø2ØB	Ø2	ADH van KEYDIS
+		C	9	Ø2ØC	C9	CMP #
+		1	Ø	Ø2ØD	1Ø	met 1Ø
+		F	Ø	Ø2ØE	FØ	BEQ
+		F	Ø	Ø2ØF	FØ	FØ plaatsen terug is CLEAR1
+		C	9	Ø21Ø	C9	CMP #
+		1	2	Ø211	12	met 12
+		F	Ø	Ø212	FØ	BEQ
+		Ø	6	Ø213	Ø6	(offset) Ø6 plaatsen verder is PLUS
+		2	Ø	Ø214	2Ø	JSR-
+		4	9	Ø215	49	ADL van SHIFT
+		Ø	2	Ø216	Ø2	ADH van SHIFT
+		4	C	Ø217	4C	JMP- (onvoorwaardelijke sprong)
+		Ø	9	Ø218	Ø9	ADL van FIRST
+		Ø	2	Ø219	Ø2	ADH van FIRST
+		2	Ø	Ø21A	2Ø	JSR- PLUS
+		A	A	Ø21B	AA	ADL van STO1
+		Ø	2	Ø21C	Ø2	ADH van STO1
+		2	Ø	Ø21D	2Ø	JSR-
+		9	4	Ø21E	94	ADL van CLDISP
+		Ø	2	Ø21F	Ø2	ADH van CLDISP
+		2	Ø	Ø22Ø	2Ø	JSR- SECOND
+		6	F	Ø221	6F	ADL van KEYDIS
+		Ø	2	Ø222	Ø2	ADH van KEYDIS
+		C	9	Ø223	C9	CMP #
+		1	Ø	Ø224	1Ø	met 1Ø
+		F	Ø	Ø225	FØ	BEQ
+		Ø	A	Ø226	ØA	ØA plaatsen verder (offset) is CLEAR2
+		C	9	Ø227	C9	CMP #
+		1	1	Ø228	11	met 11
+		F	Ø	Ø229	FØ	BEQ
+		Ø	C	Ø22A	ØC	(offset); ØA plaatsen verder is EQUAL
+		2	Ø	Ø22B	2Ø	JSR-
+		4	9	Ø22C	49	ADL van SHIFT
+		Ø	2	Ø22D	Ø2	ADH van SHIFT
+		4	C	Ø22E	4C	JMP-
+		2	Ø	Ø22F	2Ø	ADL van SECOND
+		Ø	2	Ø23Ø	Ø2	ADH van SECOND

+	2	Ø	Ø231	2Ø	JSR- CLEAR2
+	9	4	Ø232	94	ADL van CLDISP
+	Ø	2	Ø233	Ø2	ADH van CLDISP
+	4	C	Ø234	4C	JMP-
+	2	Ø	Ø235	2Ø	ADL van SECOND
+	Ø	2	Ø236	Ø2	ADH van SECOND
+	2	Ø	Ø237	2Ø	JSR- EQUAL
+	9	D	Ø238	9D	ADL van STO2
+	Ø	2	Ø239	Ø2	ADH van STQ2
+	2	Ø	Ø23A	2Ø	JSR-
+	5	9	Ø23B	59	ADL van ADD
+	Ø	2	Ø23C	Ø2	ADH van ADD
+	2	Ø	Ø23D	2Ø	JSR-
+	B	7	Ø23E	B7	ADL van RESDIS
+	Ø	2	Ø23F	Ø2	ADH van RESDIS
+	2	Ø	Ø24Ø	2Ø	JSR-
+	8	2	Ø241	82	ADL van CLB1
+	Ø	2	Ø242	Ø2	ADH van CLB1
+	2	Ø	Ø243	2Ø	JSR-
+	8	B	Ø244	8B	ADL van CLB2
+	Ø	2	Ø245	Ø2	ADH van CLB2
+	4	C	Ø246	4C	JMP-
+	Ø	9	Ø247	Ø9	ADL van FIRST
+	Ø	2	Ø248	Ø2	ADH van FIRST
N.B. Laatste regel van hoofdprogramma					
+	A	Ø	Ø249	AØ	LDY #; subroutine SHIFT
+	Ø	4	Ø24A	Ø4	Ø4 in Y-indexregister
+	Ø	6	Ø24B	Ø6	ASLZ SHIFT1
+	F	9	Ø24C	F9	INH op adres ØØF9
+	2	6	Ø24D	26	ROLZ
+	F	A	Ø24E	FA	POINTL op adres ØØFA
+	2	6	Ø24F	26	ROLZ
+	F	B	Ø25Ø	FB	POINTH op adres ØØFB
+	8	8	Ø251	88	DEY verlaag inhoud Y met 1
+	D	Ø	Ø252	DØ	BNE
+	F	7	Ø253	F7	F7 plaatsen terug is SHIFT1
+	Ø	5	Ø254	Ø5	ORAZ
+	F	9	Ø255	F9	bit voor bit ORren met INH
+	8	5	Ø256	85	STAZ
+	F	9	Ø257	F9	inhoud van accu naar INH (adr ØØF9)
+	6	Ø	Ø258	6Ø	RTS terugkeer naar hoofdprogramma
+	F	8	Ø259	F8	SED decimaal rekenen subroutine ADD
+	1	8	Ø25A	18	CLC
+	A	5	Ø25B	A5	LDAZ
+	Ø	Ø	Ø25C	ØØ	ADL van B1Ø (adres ØØØØ); BIØ in accu
+	6	5	Ø25D	65	ADCZ
+	Ø	3	Ø25E	Ø3	ADL van B2Ø (adres ØØØ3); accu = B1Ø + B2Ø
+	8	5	Ø25F	85	STAZ
+	Ø	6	Ø26Ø	Ø6	ADL van RØ; accu → RØ
+	A	5	Ø261	A5	LDAZ
+	Ø	1	Ø262	Ø1	ADL van B11 (adres ØØØ1); B11 in accu
+	6	5	Ø263	65	ADCZ

+	Ø	4	Ø264	Ø4	ADL van B21 (adres ØØØ4); accu = B11 + B21
+	8	5	Ø265	85	STAZ
+	Ø	7	Ø266	Ø7	ADL van R1 (adres ØØØ7); accu → R1
+	A	5	Ø267	A5	LDAZ
+	Ø	2	Ø268	Ø2	ADL van B12 (adres ØØØ2); B12 in accu
+	6	5	Ø269	65	ADCZ
+	Ø	5	Ø26A	Ø5	ADL van B22 (adres ØØØ5); accu = B12 + B22
+	8	5	Ø26B	85	STAZ
+	Ø	8	Ø26C	Ø8	ADL van R2 (adres ØØØ8); accu → R2
+	D	8	Ø26D	D8	CLD herstel situatie van binair rekenen
+	6	Ø	Ø26E	6Ø	RTS terugkeer naar hoofdprogramma
+	2	Ø	Ø26F	2Ø	JSR- subroutine KEYDIS
+	8	E	Ø270	8E	ADL van SCANDS } in monitor
+	1	D	Ø271	1D	ADH van SCANDS } (1D8E)
+	D	Ø	Ø272	DØ	BNE
+	F	B	Ø273	FB	(offset); FB plaatsen terug is KEYDIS
+	2	Ø	Ø274	2Ø	JSR- KD
+	8	E	Ø275	8E	ADL van SCANDS } in monitor
+	1	D	Ø276	1D	ADH van SCANDS } (1D8E)
+	F	Ø	Ø277	FØ	BEQ
+	F	B	Ø278	FB	(offset); FB plaatsen terug is KD
+	2	Ø	Ø279	2Ø	JSR-
+	8	E	Ø27A	8E	ADL van SCANDS } in monitor
+	1	D	Ø27B	1D	ADH van SCANDS } (1D8E)
+	F	Ø	Ø27C	FØ	BEQ
+	F	6	Ø27D	F6	(offset); F6 plaatsen terug is KD
+	2	Ø	Ø27E	2Ø	JSR-
+	F	9	Ø27F	F9	ADL van GETKEY } in monitor
+	1	D	Ø28Ø	1D	ADH van GETKEY } (1DF9)
+	6	Ø	Ø281	6Ø	RTS terugkeer naar hoofdprogramma
+	A	9	Ø282	A9	LDA #; subroutine CLB1
+	Ø	Ø	Ø283	ØØ	ØØ → accu
+	8	5	Ø284	85	STAZ
+	Ø	Ø	Ø285	ØØ	accu → B1Ø (= ØØ)
+	8	5	Ø286	85	STAZ
+	Ø	1	Ø287	Ø1	ØØ → B11
+	8	5	Ø288	85	STAZ
+	Ø	2	Ø289	Ø2	ØØ → B12
+	6	Ø	Ø28A	6Ø	RTS terugkeer naar hoofdprogramma
+	A	9	Ø28B	A9	LDA #; subroutine CLB2
+	Ø	Ø	Ø28C	ØØ	ØØ → accu
+	8	5	Ø28D	85	STAZ
+	Ø	3	Ø28E	Ø3	ØØ → B2Ø
+	8	5	Ø28F	85	STAZ
+	Ø	4	Ø29Ø	Ø4	ØØ → B21
+	8	5	Ø291	85	STAZ
+	Ø	5	Ø292	Ø5	ØØ → B22
+	6	Ø	Ø293	6Ø	RTS terugkeer naar hoofdprogramma
+	A	9	Ø294	A9	LDA #; subroutine CLDISP
+	Ø	Ø	Ø295	ØØ	ØØ → accu

+	8	5	Ø296	85	STAZ
+	F	9	Ø297	F9	ØØ → INH (adres ØØF9)
+	8	5	Ø298	85	STAZ
+	F	A	Ø299	FA	ØØ → POINTL (adres ØØFA)
+	8	5	Ø29A	85	STAZ
+	F	B	Ø29B	FB	ØØ → POINTH (adres ØØFB)
+	6	Ø	Ø29C	6Ø	RTS terugkeer naar hoofdprogramma
+	A	5	Ø29D	A5	LDAZ; subroutine STO2
+	F	9	Ø29E	F9	INH (adres ØØF9) → accu
+	8	5	Ø29F	85	STAZ
+	Ø	3	Ø2AØ	Ø3	accu (= INH) → B2Ø (adres ØØØ3)
+	A	5	Ø2A1	A5	LDAZ
+	F	A	Ø2A2	FA	POINTL (adres ØØFA) → accu
+	8	5	Ø2A3	85	STAZ
+	Ø	4	Ø2A4	Ø4	accu (= POINTL) → B21 (adres ØØØ4)
+	A	5	Ø2A5	A5	LDAZ
+	F	B	Ø2A6	FB	POINTH (adres ØØFB) → accu
+	8	5	Ø2A7	85	STAZ
+	Ø	5	Ø2A8	Ø5	accu (= POINTH) → B22 (adres ØØØ5)
+	6	Ø	Ø2A9	6Ø	RTS terugkeer naar hoofdprogramma
+	A	5	Ø2AA	A5	LDAZ; subroutine STO1
+	F	9	Ø2AB	F9	INH (adres ØØF9) → accu
+	8	5	Ø2AC	85	STAZ
+	Ø	Ø	Ø2AD	ØØ	accu (= INH) → B1Ø (adres ØØØØ)
+	A	5	Ø2AE	A5	LDAZ
+	F	A	Ø2AF	FA	POINTL (adres ØØFA) → accu
+	8	5	Ø2BØ	85	STAZ
+	Ø	1	Ø2B1	Ø1	accu (= POINTL) → B11 (adres ØØØ1)
+	A	5	Ø2B2	A5	LDAZ
+	F	B	Ø2B3	FB	POINTH (adres ØØFB) → accu
+	8	5	Ø2B4	85	STAZ
+	Ø	2	Ø2B5	Ø2	accu (= POINTH) → B12 (adres ØØØ2)
+	6	Ø	Ø2B6	6Ø	RTS terugkeer naar hoofdprogramma
+	A	5	Ø2B7	A5	LDAZ; subroutine RESDIS
+	Ø	6	Ø2B8	Ø6	RØ (adres ØØØ6) → accu
+	8	5	Ø2B9	85	STAZ
+	F	9	Ø2BA	F9	accu (= RØ) → INH (adres ØØF9)
+	A	5	Ø2BB	A5	LDAZ
+	Ø	7	Ø2BC	Ø7	R1 (adres ØØØ7) → accu
+	8	5	Ø2BD	85	STAZ
+	F	A	Ø2BE	FA	accu (= R1) → POINTL (adres ØØFA)
+	A	5	Ø2BF	A5	LDAZ
+	Ø	8	Ø2CØ	Ø8	R2 (adres ØØØ8) → accu
+	8	5	Ø2C1	85	STAZ
+	F	B	Ø2C2	FB	accu (= R2) → POINTH (adres ØØFB)
+	6	Ø	Ø2C3	6Ø	RTS terugkeer naar hoofdprogramma

Figuur 1. Het toetsprogramma van het decimale optelprogramma uit hoofdstuk 3 bestaat uit 196 bytes, beslaat dus 196 geheugenplaatsen. Het gedetailleerde stroomdiagram van het hoofdprogramma staat in figuur 20 van hoofdstuk 3, die van de negen subroutines in figuur 21a . . . 21i.

hebben elk een label. Bij het opzetten van het toetsprogramma kan men voorlopig in plaats van adresbytes het label aangeven van het programmadeel waarnaar wordt gesprongen. Als later het adres van het label bekend is vult men de adresbytes of het offset-byte alsnog in.
Ja, en wanneer is nou eigenlijk alles bekend? Offset-bytes kan men met de junior-computer berekenen. Daarover straks meer. En wanneer zijn alle adresbytes bekend? Pas als het aantal geheugenplaatsen nodig voor het programma van waar uit wordt gesprongen bekend is. Dus als de bytesbegroting rond is.
Het is niet alleen belangrijk dat een programmadeel een aaneengesloten reeks geheugenplaatsen omvat, maar ook dat programmadelen elkaar niet overlappen. Gebruikt men per abuis bepaalde adressen voor verschillende doeleinden (dus voor verschillende programma's), dan blijft het laatst ingetoetste behouden. Als je niet uitkijkt kan een subroutine op adressen staan die nodig zijn voor het hoofdprogramma.
Het is overigens geen wet van meden en perzen dat hoofdprogramma en subroutines op elkaar aansluiten kwa adressen, zoals het geval is in figuur 1. Tussen hoofdprogramma en eerstvolgende subroutine mogen zich op de geheugenplattegrond (memory map) best lege, ongebruikte geheugenplaatsen bevinden, evenals tussen subroutines. Het is natuurlijk wel een kwestie van ekonomisch gebruik van geheugenplaatsen om zo min mogelijk open plaatsen te krijgen. Vooral bij grote programma's kan dat van belang zijn. Immers: als het papier schaars is laat je ook niet het onderste kwart van een vel leeg om op een nieuw vel verder te gaan. Bij kleinere programma's kan men natuurlijk kwistiger omgaan met geheugenplaatsen en kan men ruimer begroten: Als een hoofdprogramma bijvoorbeeld 50 geheugenplaatsen beslaat is er niets op tegen om een subroutine 100 plaatsen vanaf het startadres van de hoofdroutine te plannen. Dan kan men vrijwel meteen adresbytes invullen en hoeft men dat niet meer pas achteraf te doen.
Hoeveel geheugenplaatsen staan ons eigenlijk ter beschikking?

4. De 1 k-RAM die de gebruiker van de junior-computer in de standaarduitvoering ten dienste staat voor eigen programma's beslaat de adressen ØØØØ . . . Ø3FF. Dat zijn vier pagina's, elk van 256 bytes:

pagina ØØ : ØØØØ . . . ØØFF
pagina Ø1 : Ø1ØØ . . . Ø1FF
pagina Ø2 : Ø2ØØ . . . Ø2FF
pagina Ø3 : Ø3ØØ . . . Ø3FF

waarbij overigens voor pagina nul enige beperkingen gelden. Een aantal geheugenplaatsen van pagina nul is gereserveerd voor de data die het resultaat is van akties van het monitorprogramma. Dat zijn de 31 geheugenplaatsen met de adressen ØØE1 . . . ØØFF. Een paar van die geheugenplaatsen kennen we al, zoals ØØF9 (label INH), ØØFA (POINTL) en ØØFB (POINTH). En verder de plaatsen ØØEF . . . ØØF5, die zijn gereserveerd voor de inhoud van alle interne registers van de 6502 (SAVE-routine). Met de andere monitor-geheugenplaatsen op pagina nul kunt u kennismaken in *Junior-computer 2.*
Andere geheugenplaatsen op pagina nul dient men zèlf te reserveren, indien men gebruik maakt van adressering op pagina nul. Pagina Ø1 kan als stapel (stack) worden gebruikt.

Er zijn ook niet te beschrijven geheugenplaatsen:

5. De 1 k-EPROM van de junior-computer bevat het monitorprogramma. Dit beslaat de adressen 1CØØ . . . 1FFF, dus de pagina's 1C, 1D, 1E en 1F. Deze geheugenplaatsen zijn slechts in zoverre toegankelijk voor de gebruiker van de junior-computer dat men gebruik kan maken van bepaalde monitor-routines.

Dat laatste is ook in figuur 1 het geval. Kijk maar naar de subroutines KEYDIS (Ø26F . . . Ø281), waarin gebruik wordt gemaakt van de monitor-routines SCANDS en GETKEY.
Maar we zijn er nog niet:

6. De 1/8 k-RAM in de PIA staat de gebruiker van de junior-computer ten dienste voor eigen programma's. Deze RAM omvat de adressen 1AØØ . . . 1A7F; dat is de eerste helft van pagina 1A.

Net als bij pagina nul van de standaard-RAM geldt ook hier een beperking: de vier geheugenplaatsen 1A7A, 1A7B, 1A7E en 1A7F zijn gereserveerd voor de NMI-sprongvektor en voor de IRQ-sprongvektor. Deze geheugenplaatsen zijn alleen dan te gebruiken voor een gebruikersprogramma indien geen gebruik wordt gemaakt van NMI's en IRQ's (zie het hoofdstukje "opstarten", dat binnenkort aan de orde is).
Een gedeelte van de geheugenplaatsen van de tweede helft van pagina 1F (1A8Ø . . . 1AFB) is nodig voor de werking van de PIA. Meer hierover in *Junior-computer 2.* Ook deze geheugenplaatsen dient men (voorlopig) bij het programmeren te mijden.

De offset: het "spring-byte"

Voor het berekenen van de offset (adresverplaatsing) van een voorwaardelijke spronginstruktie kunnen we gebruik maken van de monitor-routine BRANCH met startadres 1FD5. Dat hebben we in hoofdstuk 3 al gezien. Nadat het programma is gestart via GO toetsen we eerst het rechter byte ADL in van het adres van de opcode van de voorwaardelijke spronginstruktie, vervolgens de ADL van het sprongadres. Laten we eens alle offsets van figuur 1 gaan berekenen:

RST	AD			XXXX	XX	
1	F	D	5	1FD5	D8	
GO				ØØØØ	ØØ	
Ø	E	Ø	Ø	ØEØØ	FØ	FØ op adres Ø2ØF
1	2	1	A	121A	Ø6	Ø6 op adres Ø213
2	5	3	1	2531	ØA	ØA op adres Ø226
2	9	3	7	2937	ØC	ØC op adres Ø22A
5	2	4	B	524B	F7	F7 op adres Ø253
7	2	6	F	726F	FB	FB op adres Ø273
7	7	7	4	7774	FB	FB op adres Ø278
7	C	7	4	7C74	F6	F6 op adres Ø27D
RST						

N.B. Tussen twee offsetberekeningen kan men door het indrukken van een willekeurige kommandotoets het display ØØØØØØ maken.

Het bepalen van de offset hoeft overigens niet met het monitorprogramma te gebeuren; het kan ook anders. Het komt erop neer dat men het aantal stappen telt dat nodig is om het sprongadres te bereiken; elke verhoging of verlaging van het adres levert een bijdrage van 1 aan het aantal stappen vooruit of terug. *Uitgangspunt van het stappen tellen is niet het adres met de opcode van de spronginstruktie, maar het adres, twee plaatsen verder,* dus het adres van de geheugenplaats die volgt op de geheugenplaats met het offset-byte. Stappen terug zijn negatief. Via de twee-komplement-notatie van het getal komt men dan tot het hexadecimale offset-byte. In figuur 5 van hoofdstuk 2 staat een handig overzicht van de hexadecimale notatie van de negatieve getallen −1 t/m −40. Zie ook Aanhangsel 4 (pagina 158).

Opstarten

Ook het opstarten van de junior-computer is al her en der in hoofdstuk drie aan de orde gekomen. Toch stellen we het hier nog even aan de orde. Het opstarten gaat zo:

RST	AD	XXXX XX
1 A	7 A	1A7A XX
DA	Ø Ø	1A7A ØØ
+	1 C	1A7B 1C
+		1A7C XX
+		1A7D XX
+	Ø Ø	1A7E ØØ
+	1 C	1A7F 1C

Wat we in feite hebben gedaan is het laden van de geheugenplaats met adres 1A7E met ØØ en van plaats 1A7F met 1C. Deze geheugenplaatsen (op pagina 1F, in de PIA-RAM) bevatten het effektieve adres (1CØØ, in het monitor-programma) waarnaar wordt gesprongen zodra een BRK optreedt, en dat is de software-versie van de hardware-instruktie IRQ. Het laatste deel van hoofdstuk 3, en met name figuur 31 is aan deze kwestie gewijd. De geheugenplaatsen 1A7A en 1A7B bevatten ook het effektieve adres 1CØØ. Hiernaar wordt gesprongen na het optreden van een NMI. In de standaard-versie van de junior-computer is dat het geval als de toets ST wordt ingedrukt.

Het indrukken van toets RST tijdens het opstarten heeft een sprong naar de geheugenplaatsen 1FFC+1FFD tot gevolg. Daar ligt de resetvektor 1C1D, die wijst naar dat deel van het monitorprogramma dat zich bezig houdt met toetsdetektie, toetsherkenning en het in beeld brengen van ingetoetste adressen en data.

De gegeven toetsen-startceremonie is soms te vereenvoudigen. Ziet men af van het gebruik van BRK-instrukties, dan hoeven de plaatsen 1A7E en 1A7F niet te worden geladen. Ziet men af van het gebruik van de toets ST en van stapvoorstap-programmeren, dan hoeven de plaatsen 1A7A en 1A7B niet te worden gevuld. Ziet men van beide af, dan kan worden volstaan met een RST . In alle drie gevallen wijst de sprongvektor naar het monitorprogramma. Na RST bereikt men het monitorprogramma (START: 1C33) via een opstartroutine, via een NMI of IRQ (=BRK) wordt 1C33 bereikt nadat een programmadeel (met startadres 1CØØ) is doorlopen

dat zich bezighoudt met het bewaren van de inhoud van alle registers van de 6502. Uiteraard valt er direkt na het inschakelen van de junior-computer niet zoveel te bewaren; vandaar de direkte sprong naar 1C33 na een RST. Na de warming up en nadat het programma korrekt is ingetoetst kunnen we het gaan uitvoeren:

```
AD  Ø 2 Ø Ø     Ø2ØØ2Ø (ingeven startadres)
GO              ØØØØØØ (programmastart)
2   4 5 6       ØØ2456
+               ØØØØØØ
4   1 3 2       ØØ4132
DA  (= [=] )    ØØ6588
AD  (= [CLEAR] ) ØØØØØØ
1   9 8 5 3 1   198531
+               ØØØØØØ
8   3 2 7 Ø 2   832702
DA              Ø31233
AD              ØØØØØØ
```

In het laatste geval overschrijdt het resultaat van de optelling 999999 en is dus de maat vol ("overflow"). Alleen de rechter zes cijfers van het getal verschijnen in beeld.

De junior-computer als dobbelsteen

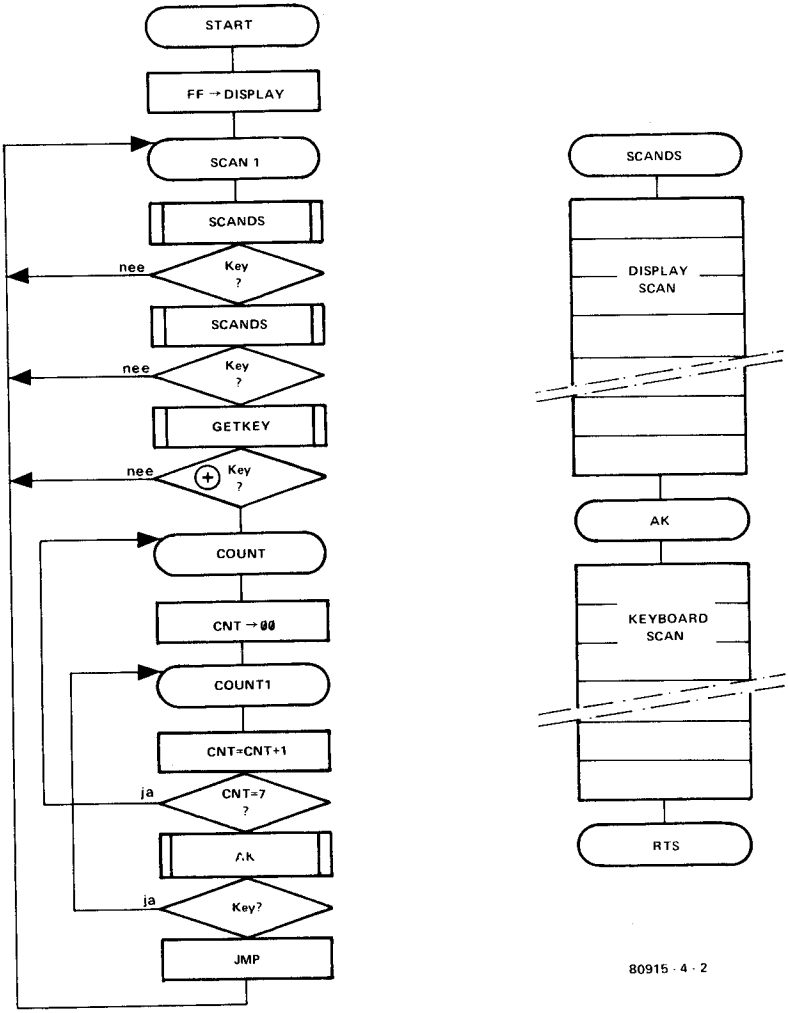
De teerling werpen via software

Ook voor dit programmavoorbeeld ligt de nadruk op de edukatieve waarde. Dobbelstenen zijn goedkoop en de kosten van een elektronische dobbelsteen zijn nu ook niet bepaald hoog te noemen. Dat neemt niet weg dat je met de junior-computer kunt dobbelen – zonder dat de stenen van tafel rollen.

Het werpen van de dobbelsteen is nagebootst door het indrukken van de plustoets. Zodra deze toets is losgelaten is het rollen beëindigd. Er ligt dan een vlak met een bepaald aantal ogen boven. Zoals bekend kan dat aantal variëren van 1 tot en met 6. Het aantal boven liggende ogen is nagebootst met een software-teller, die periodiek telt van 1 tot 7 oftewel van 1 tot en met 6. De teller start zodra de plustoets is ingedrukt en stopt als deze is losgelaten. De tellerstand die was bereikt bij het loslaten van de toets (Ø1, Ø2, Ø3, Ø4, Ø5, of Ø6) verschijnt op de middelste twee displays (die normaal het rechter adresbyte ADL aangeven). De linker en rechter twee displays laten "FF" zien.

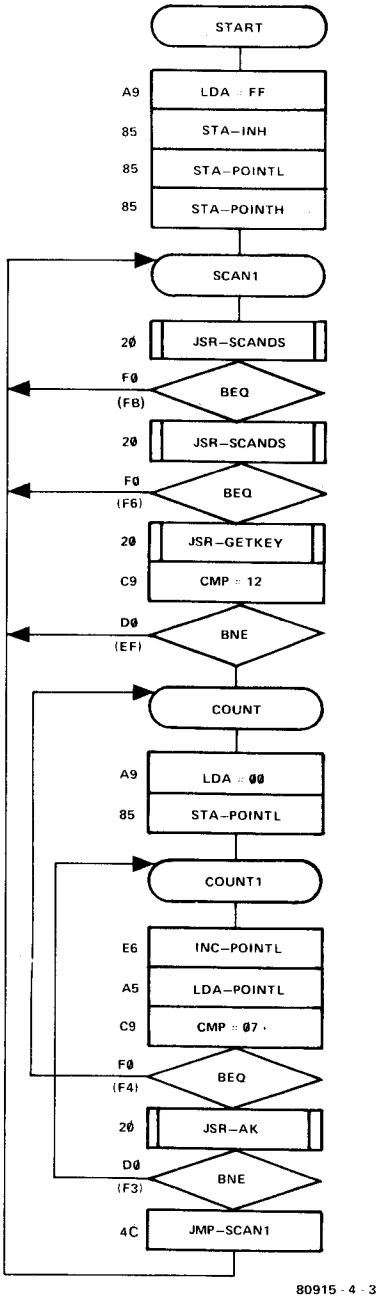
Het globale stroomdiagram van de software-dobbelsteen staat in figuur 2. Er is dankbaar gebruik gemaakt van de monitor-routines SCANDS, AK en GETKEY, die we in hoofdstuk 3 ook al een paar keer zijn tegengekomen.

Wat gebeurt er allemaal in figuur 2? Begonnen wordt met het vullen van het display met FFFFFF. Vervolgens is het programmadeel, gelabeld SCAN1 aan de orde. Dat begint met de monitor-routine SCANDS; deze zorgt ervoor dat de inhoud van de drie display-buffers (zie figuur 14 van hoofdstuk 3) op de displays komt te staan, en verder kijkt de routine of er een toets is ingedrukt. Met de laatste taak is de direkt op SCANDS aan-



Figuur 2. Het globale stroomdiagram van de software-dobbelsteen. Er wordt de hulp ingeroepen van drie verschillende monitor-routines.

sluitende routine AK belast. Zolang geen toets is ingedrukt keert het programma terug naar SCAN1 (wachtlus). Is er wel een toets ingedrukt dan wordt de hele SCANDS-procedure nog eens herhaald. In hoofdstuk 3 staat waarom: zo krijg je een geprogrammeerde toetsdenderonderdrukking. Vervolgens stelt GETKEY vast welke toets is ingedrukt. Zolang dat + niet is gaan we via een wachtlus "terug naar AF".
Was het wèl + , dan begint het eigenlijke periodieke tellen (COUNT). Uit het getailleerde stroomdiagram van figuur 3 blijkt dat begonnen wordt met het via de accu laden van de display-buffer POINTL (voor de middelste



Figuur 3. Het gedetailleerde stroomdiagram van de software-dobbelsteen.

twee displays) met ØØ. Vervolgens (COUNT1) wordt de inhoud van POINTL (=CNT) met 1 opgehoogd en in de accu gezet. Daarna kijken we of de tellerstand CNT gelijk is aan Ø7. De eerste keer is dat natuurlijk niet het geval; na zeven keer doorlopen van COUNT1 wel. Dreigt de tellerstand Ø7 te worden dan gaat het terug naar COUNT: de tellerstand wordt weer ØØ gemaakt.
Bij een tellerstand ongelijk aan Ø7 gaan we door naar de routine AK. Deze routine test of + nog steeds is ingedrukt. Zoja, dan wordt er doorgeteld (sprong naar COUNT1); zonee, dan stopt de teller; via een JMP gaat het terug naar SCAN1 en tijdens SCANDS verschijnt de tellerstand in beeld. En dan herhaalt zich de hele procedure.
In figuur 3 kan men zien hoe het bovenstaande testen op een tellerstand Ø7 in zijn werk gaat. Deze stand (CNT) gaat de accu in en wordt dan via een CMP-instruktie vergeleken met Ø7.
Ook in figuur 3 stellen we vast dat het programmadeel COUNT1 uit een handvol instrukties bestaat. Elke instruktie vergt een paar mikrosekonden (zie het instruktie-overzicht van Aanhangsel 2). Dat betekent dat de teller voldoende snel telt: voldoende snel ten opzichte van de tijd dat de plus-toets is ingedrukt. En dat laatste is belangrijk om de tellerstand niet door manipulaties (valsspelen) te laten beïnvloeden.
Het toetsprogramma staat in figuur 4. Ook hier is als startadres Ø2ØØ genomen. Ook hier is het toetsengebeuren rijkelijk voorzien van kommentaar. De programmastart gaat als volgt:

AD Ø 2 Ø Ø GO

Waarna men zo veel keren als men wenst de plustoets mag indrukken om hoge ogen te gooien.

	toetsen			adres	data	
RST		AD		xxxx	xx	
Ø	2	Ø	Ø	Ø2ØØ	xx	
DA		A	9	Ø2ØØ	A9	A9 LDA IMM [START]
+		F	F	Ø2Ø1	FF	FF → accu
+		8	5	Ø2Ø2	85	STA Z
+		F	9	Ø2Ø3	F9	FF → INH (ØØF9)
+		8	5	Ø2Ø4	85	STA Z
+		F	A	Ø2Ø5	FA	FF → POINTL (ØØFA)
+		8	5	Ø2Ø6	85	STA Z
+		F	B	Ø2Ø7	FB	FF → POINTH (ØØFB)
+		2	Ø	Ø2Ø8	2Ø	JSR- [SCAN1]
+		8	E	Ø2Ø9	8E	ADL } van SCANDS (monitor)
+		1	D	Ø2ØA	1D	ADH } (adres 1D8E)
+		F	Ø	Ø2ØB	FØ	BEQ
+		F	B	Ø2ØC	FB	(offset); FB plaatsen terug is SCAN1
+		2	Ø	Ø2ØD	2Ø	JSR-
+		8	E	Ø2ØE	8E	ADL } van SCANDS (monitor)
+		1	D	Ø2ØF	1D	ADH } (adres 1D8E)
+		F	Ø	Ø21Ø	FØ	BEQ

+		F	6	Ø211	F6	F6 plaatsen terug is SCAN1
+		2	Ø	Ø212	2Ø	JSR-
+		F	9	Ø213	F9	ADL } van GETKEY (monitor)
+		1	D	Ø214	1D	ADH } (adres 1DF9)
+		C	9	Ø215	C9	CMP IMM
+		1	2	Ø216	12	met 12
+		D	Ø	Ø217	DØ	BNE
+		E	F	Ø218	EF	EF plaatsen terug is SCAN1
+		A	9	Ø219	A9	LDA IMM COUNT
+		Ø	Ø	Ø21A	ØØ	ØØ → accu
+		8	5	Ø21B	85	STA Z
+		F	A	Ø21C	FA	ØØ → POINTL (ØØFA)
+		E	6	Ø21D	E6	INC Z COUNT1
+		F	A	Ø21E	FA	POINTL + 1 → POINTL
+		A	5	Ø21F	A5	LDA Z
+		F	A	Ø22Ø	FA	POINTL → accu
+		C	9	Ø221	C9	CMP IMM
+		Ø	7	Ø222	Ø7	met Ø7
+		F	Ø	Ø223	FØ	BEQ
+		F	4	Ø224	F4	F4 plaatsen terug is COUNT
+		2	Ø	Ø225	2Ø	JSR-
+		B	1	Ø226	B1	ADL } van AK (monitor)
+		1	D	Ø227	1D	ADH } (adres 1DB1)
+		D	Ø	Ø228	DØ	BNE
+		F	3	Ø229	F3	F3 plaatsen terug is COUNT1
+		4	C	Ø22A	4C	JMP-
+		Ø	8	Ø22B	Ø8	ADL } van SCAN1
+		Ø	2	Ø22C	Ø2	ADH } van SCAN1
AD						
Ø	2	Ø	Ø	Ø2ØØ	A9	startadres
GO						programmastart
+				FFØ4	FF	de teerling is geworpen!
+				FFØ1	FF	
+				FFØ6	FF	
+				FFØ2	FF	
enzovoorts						

Figuur 4. Het toetsprogramma van de software-dobbelsteen.

Instruktielengte meten via software

Instrukties zijn één, twee of drie bytes lang. Eén byte is er nodig voor de opcode en er zijn geen, één of twee bytes nodig voor het vastleggen van de operand.

De opcode bestaat uit één byte oftewel twee hexadecimale tekens die elk

Rechter hexadecimaal teken

Linker hexadecimaal teken	Ø	1	2	3	4	5	6	7
Ø	BRK (1)	ORA (IND,X) (2)				ORA Z (2)	ASL Z (2)	
1	BPL (2)	ORA (IND),Y (2)				ORA Z,X (2)	ASL Z,X (2)	
2	JSR (3)	AND (IND,X) (2)			BIT Z (2)	AND Z (2)	ROL Z (2)	
3	BMI (2)	AND (IND),Y (2)				AND Z,X (2)	ROL Z,X (2)	
4	RTI (1)	EOR (IND,X) (2)				EOR Z (2)	LSR Z (2)	
5	BVC (2)	EOR (IND),Y (2)				EOR Z,X (2)	LSR Z,X (2)	
6	RTS (1)	ADC (IND,X) (2)				ADC Z (2)	ROR Z (2)	
7	BVS (2)	ADC (IND),Y (2)				ADC Z,X (2)	ROR Z,X (2)	
8		STA (IND,X) (2)			STY Z (2)	STA Z (2)	STX Z (2)	
9	BCC (2)	STA (IND),Y (2)			STY Z,X (2)	STA Z,X (2)	STX Z,Y (2)	
A	LDY IMM (2)	LDA (IND,X) (2)	LDX IMM (2)		LDY Z (2)	LDA Z (2)	LDX Z (2)	
B	BCS (2)	LDA (IND),Y (2)			LDY Z,X (2)	LDA Z,X (2)	LDX Z,Y (2)	
C	CPY IMM (2)	CMP (IND,X) (2)			CPY Z (2)	CMP Z (2)	DEC Z (2)	
D	BNE (2)	CMP (IND),Y (2)				CMP Z,X (2)	DEC Z,X (2)	
E	CPX IMM (2)	SBC (IND,X) (2)			CPX Z (2)	SBC Z (2)	INC Z (2)	
F	BEQ (2)	SBC (IND),Y (2)				SBC Z,X (2)	INC Z,X (2)	

Figuur 5. Deze opcodetabel is de kompakte versie van de opcodetabel volgens Aanhangsel 1, achter in dit boek. De kolom-informatie, dat is het rechter nibble van de opcode, speelt een belangrijke rol in het programma dat voor elke opcode de bijbehorende instruktielengte bepaalt.

een half byte (nibble) voor hun rekening nemen. In Aanhangsel 1, achterin dit boek staat een overzicht van alle 256 mogelijke kombinaties van twee hexadecimale tekens met — indien van toepassing — de bijbehorende mnemonics plus adresseringsmetode (dat laatste uitsluitend voor instrukties met meerdere adresseringsmogelijkheden). Aanhangsel 1 is nog eens kompakt samengevat in de tabelvorm van figuur 5. Voor elke rij van figuur 5 is het linker nibble gelijk; voor elke kolom is het rechter nibble gelijk. De 256 elementen van de tabel bestaan uit 29 opcodes van 1-byte-instrukties, 74 opcodes van 2-byte-instrukties, 48 opcodes van 3-byte-instrukties en 105 lege plaatsen.
Welnu, we willen een programma maken dat van een ingetoetst byte bepaalt of het:
— de opcode is van een 1-byte-instruktie of
— de opcode van 2-byte-instruktie of
— de opcode van een 3-byte-instruktie en (soms) of het
— geen opcode voorstelt
Een dergelijk programma is niet zomaar een edukatieve "spielerei": het te behandelen programma treft men als subroutine aan in vrijwel elke assembler en editor (meer hierover in boek 2).
Van het programma van de "instruktiemeter" bespreken we nu eerst de subroutine LENACC, met het gedetailleerde stroomdiagram van figuur 6. Dit is de "body" van het programma: de beginsituatie is dat het ingetoetste byte in de accu staat; aan het eind is de instruktielengte opgeslagen in geheugenplaats BYTES. Kop en staart van het programma (ingetoetst byte naar accu en inhoud BYTES op het display zetten) volgen later.
Het X-register bevat tijdens het programma informatie over de instruktielengte; X = ØØ betekent geen opcode ("nul-byte-instruktie"), X = Ø1 een 1-byte-instruktie, X=Ø2 een 2-byte-instruktie en X=Ø3 een 3-byte-instruktie. Aan het eind van LENACC, in programmadeel LENEND gaat de inhoud van X naar geheugenplaats BYTES.
Het Y-register vervult in dit programma eveneens een allesbehalve passieve funktie. Indien namelijk een ingetoetst byte niet tot de uitzonderingen

Rechter hexadecimaal teken

Linker hexadecimaal teken	8	9	A	B	C	D	E	F	
Ø	PHP (1)	ORA IMM (2)	ASL A (1)			ORA ABS (3)	ASL ABS (3)		Ø
1	CLC (1)	ORA ABS,Y (3)				ORA ABS,X (3)	ASL ABS,X (3)		1
2	PLP (1)	AND IMM (2)	ROL A (1)		BIT ABS (3)	AND ABS (3)	ROL ABS (3)		2
3	SEC (1)	AND ABS,Y (3)				AND ABS,X (3)	ROL ABS,X (3)		3
4	PHA (1)	EOR IMM (2)	LSR A (1)		JMP ABS (3)	EOR ABS (3)	LSR ABS (3)		4
5	CLI (1)	EOR ABS,Y (3)				EOR ABS,X (3)	LSR ABS,X (3)		5
6	PLA (1)	ADC IMM (2)	ROR A (1)		JMP IND (3)	ADC ABS (3)	ROR ABS (3)		6
7	SEI (1)	ADC ABS,Y (3)				ADC ABS,X (3)	ROR ABS,X (3)		7
8	DEY (1)		TXA (1)		STY ABS (3)	STA ABS (3)	STX ABS (3)		8
9	TYA (1)	STA ABS,Y (3)	TXS (1)			STA ABS,X (3)			9
A	TAY (1)	LDA IMM (2)	TAX (1)		LDY ABS (3)	LDA ABS (3)	LDX ABS (3)		A
B	CLV (1)	LDA ABS,Y (3)	TSX (1)		LDY ABS,X(3)	LDA ABS,X (3)	LDX ABS,Y (3)		B
C	INY (1)	CMP IMM (2)	DEX (1)		CPY ABS (3)	CMP ABS (3)	DEC ABS (3)		C
D	CLD (1)	CMP ABS,Y (3)				CMP ABS,X (3)	DEC ABS,X (3)		D
E	INX (1)	SBC IMM (2)	NOP (1)		CPX ABS (3)	SBC ABS (3)	INC ABS (3)		E
F	SED (1)	SBC ABS,Y (3)				SBC ABS,X (3)	INC ABS,X (3)		F

blijkt te horen (hierover direkt meer) wordt Y geladen met de waarde van het rechter nibble van het ingetoetste byte, dus met de waarde die hoort bij een bepaalde kolom van de opcodetabel van figuur 5. Register Y fungeert vervolgens als index voor het laden van het X-register via absolute Y-geïndexeerde adressering. Het X-register had iets met de instruktielengte te maken: X wordt geladen met de inhoud van de "look up table" (opzoektabel) met label LENTBL. Om precies te zijn: met de inhoud van de Y-de plaats vanaf het adres van LENTBL. Voor elke kolom (Ø . . . F) bevat de opzoektabel de instruktielengte van de bij de kolom horende instrukties. Bevat dus ØØ, Ø1, Ø2 of Ø3.

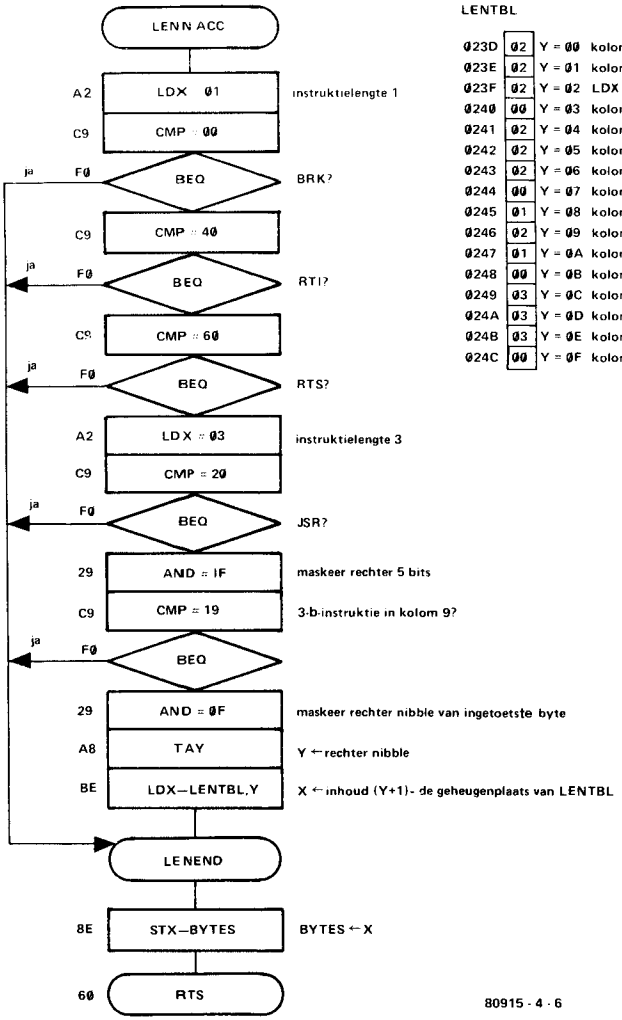
We horen in gedachten de lezer al protesteren: Schitterend, het klopt allemaal voor de 9 kolommen 1, 3, 5, 6, 7, 8, B, D en F, die elk uit 16 "0-", 1-, 2- of 3-byte-instrukties bestaan (achter de mnemonics staat in figuur 5 de instruktielengte tussen haakjes), maar de overige 7 kolommen bevatten instrukties van verschillende lengten! En in die gevallen gaat de LENTBL-vlieger niet helemaal op. Er is echter wat aan te doen.

Onregelmatige kolommen: de "moeilijke" gevallen

Om te beginnen verwaarlozen we de lege plaatsen van alle kolommen die niet uitsluitend uit lege plaatsen bestaan. Doen we dat, dan hebben we in feite aan de oorspronkelijke negen regelmatige kolommen toegevoegd de kolommen 2 (LDX IMM; kolom wordt verklaard tot een kolom met uitsluitend 2-byte-instrukties), 4 (los van de open plaatsen uitsluitend 2-byte-instrukties), A (los van de open plaatsen uitsluitend 1-byte-instrukties), C (los van de open plaatsen uitsluitend 3-byte-instrukties) en kolom E, die op de lege plaats van 9E na uit 3-byte-instrukties bestaat. De lege plaatsen kunnen overigens wel *allemaal* apart worden bekeken; zie verderop in dit verhaal.

Blijven over de kolommen Ø en 9. Kolom Ø bevat voornamelijk 2-byte-instrukties, met 1-byte-uitzonderingen BRK, RTI en RTS en de 3-byte-uitzondering JSR. Kolom 9 bevat zowel 2- als 3-byte-instrukties.

Ondanks deze uitzonderingen kunnen we nog steeds gebruik maken van het al beschreven opzoektabel-systeem. Mits we twee dingen doen: in de opzoektabel bytelengte-informatie zetten die betrekking heeft op de in de kolom *voornamelijk* aanwezige instruktielengten (dus voor Y=ØØ wordt



Figuur 6. De subroutine LENACC zorgt ervoor dat, nadat twee numerieke toetsen Ø . . . F zijn ingedrukt en in de accu geplaatst, de bij de accu-inhoud horende opcode de passende instruktielengte (Ø1, Ø2 of Ø3) krijgt toegevoegd. Indien het rechter nibble van de accu-inhoud, dus de tweede ingedrukte toets gelijk is aan 3,7 B of F (dat komt overeen met een van de lege kolommen 3, 7, B en F van figuur 5), wordt aan het ingetoetste byte de instruktielengte ØØ toegekend.

X=Ø2, evenals voor Y=Ø9) *en* ervoor zorgen dat de minderheidsuitzonderingen van de kolommen Ø en 9 apart worden bekeken. Dat laatste komt erop neer dat de 1-byte-instruktie BRK, RTI en RTS van kolom Ø en de 3-byte-instrukties van kolom Ø (JSR) en 9 apart moeten worden bekeken.

Vooraf. En wat het eerste moet gebeuren zullen we ook maar het eerste bespreken.

Uitzonderingen bevestigen de regel

De subroutine LENACC begint met het laden van X met Ø1. Waarom Ø1? Omdat we eerst de drie 1-byte-instrukties van kolom Ø gaan "uitfilteren". Dat gebeurt met drie achtereenvolgende stukjes CMP . . . BEQ, waarin het ingetoetste byte wordt vergeleken met de opcode van achtereenvolgens BRK, RTI en RTS. Is het byte opcode van één van deze drie instrukties, dan gaan we door naar LENEND:X=Ø1 komt in BYTES te staan.
Nu maken we X gelijk aan Ø3 en filteren we de 3-byte-instruktie JSR van kolom Ø uit met de instruktie CMP# 2Ø plus een aansluitende BEQ. Mocht het soms om de opcode van JSR gaan dan gaat er regelrecht worden gesprongen naar LENEND en komt er Ø3 in BYTES te staan.
Vervolgens worden met een slimme test de bokken (3-byte-instrukties) van de schapen (2-byte-instrukties) van kolom 9 gescheiden. Daartoe wordt de accu, dus het ingetoetste byte geAND met 1F (het zogenaamde *maskeren)*. Dat is een hele mondvol. Daarom nu eerst het hoe, dan het waarom. Even de AND-funktie voor de geest halen:

$$1 \cap X = X \text{ en } Ø \cap X = Ø.$$

Dat gebeurt bit voor bit met de accu-inhoud. Het resultaat komt vervolgens in de accu:
accu voor ANDen:XXXXXXXX
ANDen met 1F :ØØØ11111
accu na ANDen :ØØØXXXXX,
waarbij X van alles kan zijn (don't care), als het maar Ø of 1 is. We zien dat na afloop van het maskeren de rechter vijf bits van het byte ongewijzigd zijn. Het rechter nibble is verderop in het programma nodig voor het bepalen van de kolom waartoe het ingetoetste byte hoort (zie de al besproken opzoektabel-geschiedenis).
Nu het waarom. Het valt op dat in kolom 9 de 2- en 3-byte-instrukties elkaar afwisselen. Het linker nibble van een instruktie van kolom 9 is even voor een 2-byte-instruktie en oneven voor een 3-byte-instruktie. Voor 3-byte-instrukties van kolom 9 zijn de rechter vier bits gelijk aan (hexadecimaal) 9; vanwege het oneven linker nibble is het vierde bit van rechts gelijk aan 1; de linker drie bits zijn door het maskeren Ø geworden. Met andere woorden: door vergelijking (CMP IMM) met 19 stellen we vast of het een 3-byte-instruktie is of niet. Is dat het geval (testen met een aansluitende BEQ) dan komt de inhoud van X (Ø3) in BYTES te staan (sprong naar LENEND).
Waarmee alle uitzonderingsgevallen zijn uitgefilterd. Rest nog het ANDen (maskeren) met ØF, waardoor het rechter nibble van het ingetoetste byte wordt geïsoleerd, het opbergen van de accu-inhoud in het Y-register (TAY) en het laden van X met de betreffende "regel" (afhankelijk van Y) van de opzoektabel, en ook voor de regelmatige gevallen is LENEND in zicht.

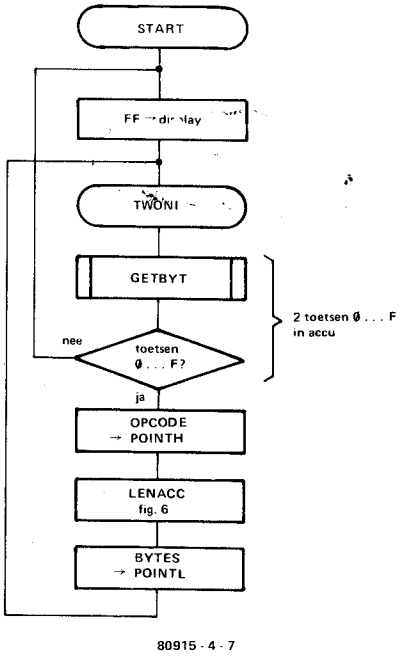
Alle lege plaatsen identificeren

Met de subroutine LENACC van figuur 6 zijn alleen de nul-byte-instrukties van de kolommen 3, 7, B en F als zodanig (X=ØØ) te identificeren; de overige 41 zijn onder tafel geveegd. Hoewel het niet zal worden gedaan in

de uitwerking van figuur 6 zullen we nu aangeven hoe alle 105 open plaatsen van de opcodetabel van figuur 5 kunnen worden opgespoord. Dat gaat zo:
De subroutine LENACC begint nu met het laden van X met ØØ. Met 26 instrukties CMP IMM . . . BEQ wordt gekeken of het ingetoetste byte hoort bij een lege plaats van kolom Ø, 4, 9, A, C of E. Zoja, dan voert de betreffende BEQ naar LENEND. Zonee, dan wordt X=Ø2 gemaakt en via CMP# A2 plus BEQ wordt gekeken of het soms de opcode is van LDX IMM, die in zijn dooie eentje kolom 2 siert. In de opcodetabel LENTBL wijzigt de waarde voor Y=Ø2 van Ø2 in ØØ. En dan volgt verder het al besproken, ongewijzigde programma van figuur 6.

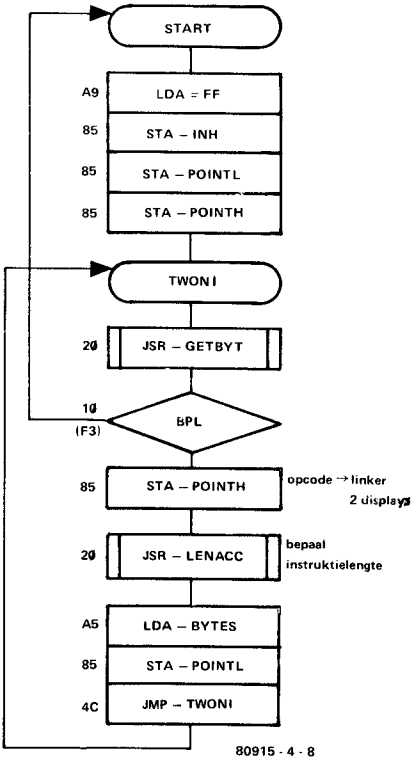
Kop en staart

Figuur 7 geeft het globale, figuur 8 het gedetailleerde stroomdiagram van het komplete programma van de "instruktiemeter". Wat gebeurt er allemaal? We drukken eerst twee numerieke toetsen in (Ø . . . F). Het overeenkomende byte komt onzichtbaar in de accu en zichtbaar op de linker twee displays terecht. De middelste twee displays geven de bij het byte (de opcode) horende instruktielengte weer met Ø1, Ø2 of Ø3. Of ØØ, in het geval dat het rechter nibble van het byte gelijk is aan 3, 7, B of F. De rechter twee displays tonen doorlopend "FF".
Het globale stroomdiagram (figuur 7). Na START worden de inmiddels welbekende displaybuffers geladen met FF. Vervolgens is programmadeel



Figuur 7. Het globale stroomdiagram van het instruktielengte-programma.

TWONI (de mnemonics van de Engelse versie van "twee nibbles") aan de beurt. De monitorroutine GETBYT zorgt ervoor dat de toetswaarde van twee achter elkaar ingedrukte toetsen in de accu terecht komt. Het linker nibble van de accu-inhoud is de toetswaarde van de eerste toets, het rechter nibble de toetswaarde van de tweede toets.
Nu hebben we geen boodschap aan kommandotoetsen. Slechts in de numerieke toetsen Ø . . . F zijn we geïnteresseerd. De routine GETBYT wordt dan ook gevolgd door een test waarin gekeken wordt of de twee ingedrukte toetsen numeriek zijn. Zolang dat niet het geval is is er sprake van een wachtlus: kommandotoetsen worden genegeerd en pas nadat er twee numerieke toetsen zijn ingedrukt gaan we verder. Verder waarmee? Met het kopiëren van de accu-inhoud in de displaybuffer POINTH. Waardoor de linker twee displays de ingetoetste opcode weergeven. Daarna is de al besproken subroutine LENACC aan de orde. Die zorgt ervoor dat geheugenplaats BYTES de instruktielengte van het byte bevat. Rest nog de verhuizing via de accu van de inhoud van BYTES naar POINTL, waardoor, na de sprong naar TWONI, tijdens GETBYT de instruktielengte op de middelste twee displays komt te staan. Want binnen GETBYT wordt ook een keer naar SCANDS gesprongen. En zoals u weet is die goed voor het te kijk zetten van de inhoud van elk der drie displaybuffers.
In het gedetailleerde stroomdiagram van figuur 8 ziet men figuur 7 uitgewerkt in instrukties. Eén ding moet nog worden toegelicht: de instruktie BPL na GETBYT. In de routine GETBYT wordt met een instruktie CMP# 1Ø onderscheid gemaakt tussen kommandotoetsen (toetswaarde $\geqslant$1Ø) en numerieke toetsen, met toetswaarden ØØ . . . ØF. Voor toetswaarden kleiner dan 10 is het gevolg van de vergelijking dat de N-vlag wordt geset, dus N=1. En de instruktie BPL betekent voorwaardelijk springen, afhankelijk van de toestand van de N-vlag. Er wordt namelijk gesprongen bij N=Ø, dus als de toetswaarde 1Ø of hoger is, dus als er sprake is van een kommandotoets. Zo zit dat.
Rest nog de stap van figuur 8 naar het toetsenprogramma van figuur 9, dat uit 77 bytes bestaat, dus 77 geheugenplaatsen in beslag neemt. Het startadres is Ø2ØØ. De hoofdroutine (figuur 8) loopt van adres Ø2ØØ tot en met Ø218, de subroutine LENACC (figuur 6) van Ø219 tot en met Ø23C en de 16 daarop volgende geheugenplaatsen Ø23D tot en met Ø24C zijn gereserveerd voor de opzoektabel LENTBL.
Tja, eigenlijk valt er over dit toetsenprogramma niet zo bijster veel meer te zeggen, vanwege de comments, die de schrijver hier een hoop gras voor de voeten wegneemt. Of het moet zijn dat de geheugenplaats BYTES (adres ØØEØ) zich op pagina nul bevindt. Dat komt omdat de monitor ook een instruktielengte-routine bevat (OPLEN; startadres 1E5C) die BYTES (adres ØØF6, een van de RAM-opbergplaatsen ØØE1 . . . ØØFF op pagina nul) als bestemming heeft voor de instruktielengte.
Nu een opgave voor u: Er is al vermeld dat lege plaatsen van de opcodetabel van figuur 5 niet allemaal worden herkend als nul-byte-instruktie. Er is ook aangegeven hoe dat euvel kan worden verholpen. Het is een goede test voor uw programmeerkennis om deze aanwijzingen konkreet te vertalen in instrukties en in een aangepast toetsprogramma. De juniorcomputer vertelt u na het intoetsen van het aangepaste toetsprogramma of u het goed heeft gedaan.



Figuur 8. Het gedetailleerde stroomdiagram van het instruktielengte-programma.

	toetsen			adres	data	comments
RTS	AD			xxxx	xx	
Ø	2	Ø	Ø	ØØØØ	xx	startadres van instruktiemeter
DA		A	9	ØØØØ	A9	LDA IMM [START]
+		F	F	ØØØ1	FF	FF → accu
+		8	5	ØØØ2	85	STAZ
+		F	9	ØØØ3	F9	accu → INH (adres ØØF9)
+		8	5	ØØØ4	85	STAZ
+		F	A	ØØØ5	FA	accu → POINTL (adres ØØFA)
+		8	5	ØØØ6	85	STA Z
+		F	B	ØØØ7	FB	accu → POINTH (adres ØØFB)
+		2	Ø	ØØØ8	ØØ	JSR- [TWONI]
+		6	F	ØØØ9	6F	ADL } van GETBYT (monitor;
+		1	D	ØØØA	1D	ADH } adres 1D6F)
+		1	Ø	ØØØB	1Ø	BPL; 2 toetsen Ø . . . F ingetoetst?

+	F	3	Ø2ØC	F3	zo nee, dan F3 plaatsen terug (= START)
+	8	5	Ø2ØD	85	STA Z (byte in accu)
+	F	B	Ø2ØE	FB	accu (= op code) →POINTH (adres ØØFB)
+	2	Ø	Ø2ØF	2Ø	JSR-
+	1	9	Ø21Ø	19	ADL } van LENACC
+	Ø	2	Ø211	Ø2	ADH } van LENACC
+	A	5	Ø212	A5	LDA Z
+	E	Ø	Ø213	EØ	BYTES (adres ØØEØ) → accu
+	8	5	Ø214	85	STA Z
+	F	A	Ø215	FA	accu →POINTL (adres ØØFA)
+	4	C	Ø216	4C	JMP ABS
+	Ø	8	Ø217	Ø8	ADL } van sprongadres
+	Ø	2	Ø218	Ø2	ADH } (TWONI)
+	A	2	Ø219	A2	LDX IMM; subroutine LENACC
+	Ø	1	Ø21A	Ø1	Ø1 →X; instruktielengte 1
+	C	9	Ø21B	C9	CMP IMM
+	Ø	Ø	Ø21C	ØØ	met ØØ
+	F	Ø	Ø21D	FØ	BEQ BRK?
+	1	A	Ø21E	1A	1A plaatsen verder is LENEND
+	C	9	Ø21F	C9	CMP IMM
+	4	Ø	Ø22Ø	4Ø	met 4Ø
+	F	Ø	Ø221	FØ	BEQ RTI?
+	1	6	Ø222	16	16 plaatsen verder is LENEND
+	C	9	Ø223	C9	CMP IMM
+	6	Ø	Ø224	6Ø	met 6Ø
+	F	Ø	Ø225	FØ	BEQ RTS?
+	1	2	Ø226	12	12 plaatsen verder is LENEND
+	A	2	Ø227	A2	LDX IMM
+	Ø	3	Ø228	Ø3	Ø3 →X; instruktielengte 3
+	C	9	Ø229	C9	CMP IMM
+	2	Ø	Ø22A	2Ø	met 2Ø
+	F	Ø	Ø22B	FØ	BEQ JSR?
+	Ø	C	Ø22C	ØC	ØC plaatsen verder is LENEND
+	2	9	Ø22D	29	AND IMM
+	1	F	Ø22E	1F	met 1F, maskeer rechter 5 bits
+	C	9	Ø22F	C9	CMP IMM
+	1	9	Ø23Ø	19	met 19
+	F	Ø	Ø231	FØ	BEQ; 3-6-instruktie in kolom 9?
+	Ø	6	Ø232	Ø6	Ø6 plaatsen verder is LENEND
+	2	9	Ø233	29	AND IMM

```
+          Ø   F   Ø234  ØF   met ØF; maskeer rechter nibble
+          A   8   Ø235  A8   TAY; rechter nibble → Y
+          B   E   Ø236  BE   LDX ABS,Y; (Y + 1) de plaats
                              van LENTBL
+          3   D   Ø237  3D   ADL } van LENTBL
+          Ø   2   Ø238  Ø2   ADH } (look up table)
+          8   E   Ø239  8E   STX ABS          LENEND
+          E   Ø   Ø23A  EØ   ADL BYTES
+          Ø   Ø   Ø23B  ØØ   ADH BYTES
+          6   Ø   Ø23C  6Ø   RTS terugkeer naar hoofdprogramma
+          Ø   2   Ø23D  Ø2   kolom Ø; Y = ØØ  LENTBL
+          Ø   2   Ø23E  Ø2   kolom 1; Y = Ø1
+          Ø   2   Ø23F  Ø2   kolom 2; Y = Ø2
+          Ø   Ø   Ø24Ø  ØØ   kolom 3; Y = Ø3
+          Ø   2   Ø241  Ø2   kolom 4; Y = Ø4
+          Ø   2   Ø242  Ø2   kolom 5; Y = Ø5
+          Ø   2   Ø243  Ø2   kolom 6; Y = Ø6
+          Ø   Ø   Ø244  ØØ   kolom 7; Y = Ø7
+          Ø   1   Ø245  Ø1   kolom 8; Y = Ø8
+          Ø   2   Ø246  Ø2   kolom 9; Y = Ø9
+          Ø   1   Ø247  Ø1   kolom A; Y = ØA
+          Ø   Ø   Ø248  ØØ   kolom B; Y = ØB
+          Ø   3   Ø249  Ø3   kolom C; Y = ØC
+          Ø   3   Ø24A  Ø3   kolom D; Y = ØD
+          Ø   3   Ø24B  Ø3   kolom E; Y = ØE
+          Ø   Ø   Ø24C  ØØ   kolom F; Y = ØF
AD
Ø    2    Ø   Ø              startadres
GO                           start programma
          A   9   A9Ø2  FF
          Ø   3   Ø3ØØ  FF
          D   2   D2Ø2  FF
          9   E   9EØ3  FF
          D   5   D5Ø2  FF
          enzovoorts
```

Figuur 9. Het toetsprogramma, gebaseerd op de figuren 6 en 8.

Waarmee hoofdstuk 4, en daarmee dit boek uit is. We zien elkaar hopelijk terug in Junior-computer 2. *Waarin u kunt kennismaken met een hoop zaken die het leven van de junior-computer-amateur nog vreugdevoller maken dan het nu al is. Of ongetwijfeld zal worden.*

1 Opcodes in hexadecimale volgorde

Tabel van opcodes in de hexadecimale volgorde ØØ-FF. Ook de niet gebruikte hexadecimale cijferkombinaties zijn opgenomen.

ØØ	BRK	2Ø	JSR	4Ø	RTI	6Ø	RTS
Ø1	ORA (IND,X)	21	AND (IND,X)	41	EOR (IND,X)	61	ADC (IND,X)
Ø2	–	22	–	42	–	62	–
Ø3	–	23	–	43	–	63	–
Ø4	–	24	BIT Z	44	–	64	–
Ø5	ORA Z	25	AND Z	45	EOR Z	65	ADC Z
Ø6	ASL Z	26	ROL Z	46	LSR Z	66	ROR Z
Ø7	–	27	–	47	–	67	–
Ø8	PHP	28	PLP	48	PHA	68	PLA
Ø9	ORA IMM	29	AND IMM	49	EOR IMM	69	ADC IMM
ØA	ASL A	2A	ROL A	4A	LSR A	6A	ROR A
ØB	–	2B	–	4B	–	6B	–
ØC	–	2C	BIT ABS	4C	JMP ABS	6C	JMP IND
ØD	ORA ABS	2D	AND ABS	4D	EOR ABS	6D	ADC ABS
ØE	ASL ABS	2E	ROL ABS	4E	LSR ABS	6E	ROR ABS
ØF	–	2F	–	4F	–	6F	–
1Ø	BPL	3Ø	BMI	5Ø	BVC	7Ø	BVS
11	ORA (IND),Y	31	AND (IND),Y	51	EOR (IND),Y	71	ADC (IND),Y
12	–	32	–	52	–	72	–
13	–	33	–	53	–	73	–
14	–	34	–	54	–	74	–
15	ORA Z,X	35	AND Z,X	55	EOR Z,X	75	ADC Z,X
16	ASL Z,X	36	ROL Z,X	56	LSR Z,X	76	ROR Z,X
17	–	37	–	57	–	77	–
18	CLC	38	SEC	58	CLI	78	SEI
19	ORA ABS,Y	39	AND ABS,Y	59	EOR ABS, Y	79	ADC ABS,Y
1A	–	3A	–	5A	–	7A	–
1B	–	3B	–	5B	–	7B	–
1C	–	3C	–	5C	–	7C	–
1D	ORA ABS,X	3D	AND ABS,X	5D	EOR ABS,X	7D	ADC ABS,X
1E	ASL ABS,X	3E	ROL ABS,X	5E	LSR ABS,X	7E	ROR ABS,X
1F	–	3F	–	5F	–	7F	–

8Ø	–	AØ	LDY IMM	CØ	CPY IMM	EØ	CPX IMM
81	STA (IND,X)	A1	LDA (IND,X)	C1	CMP (IND,X)	E1	SBC (IND,X)
82	–	A2	LDX IMM	C2	–	E2	–
83	–	A3	–	C3	–	E3	–
84	STY Z	A4	LDY Z	C4	CPY Z	E4	CPX Z
85	STA Z	A5	LDA Z	C5	CMP Z	E5	SBC Z
86	STX Z	A6	LDX Z	C6	DEC Z	E6	INC Z
87	–	A7	–	C7	–	E7	–
88	DEY	A8	TAY	C8	INY	E8	INX
89	–	A9	LDA IMM	C9	CMP IMM	E9	SBC IMM
8A	TXA	AA	TAX	CA	DEX	EA	NOP
8B	–	AB	–	CB	–	EB	–
8C	STY ABS	AC	LDY ABS	CC	CPY ABS	EC	CPX ABS
8D	STA ABS	AD	LDA ABS	CD	CMP ABS	ED	SBC ABS
8E	STX ABS	AE	LDX ABS	CE	DEC ABS	EE	INC ABS
8F	–	AF	–	CF	–	EF	–
9Ø	BCC	BØ	BCS	DØ	BNE	FØ	BEQ
91	STA (IND),Y	B1	LDA (IND),Y	D1	CMP (IND),Y	F1	SBC (IND),Y
92	–	B2	–	D2	–	F2	–
93	–	B3	–	D3	–	F3	–
94	STY Z,X	B4	LDY Z,X	D4	–	F4	–
95	STA Z,X	B5	LDA Z,X	D5	CMP Z,X	F5	SBC Z,X
96	STX Z,Y	B6	LDX Z,Y	D6	DEC Z,X	F6	INC Z,X
97	–	B7	–	D7	–	F7	–
98	TYA	B8	CLV	D8	CLD	F8	SED
99	STA ABS,Y	B9	LDA ABS,Y	D9	CMP ABS,Y	F9	SBC ABS,Y
9A	TXS	BA	TSX	DA	–	FA	–
9B	–	BB	–	DB	–	FB	–
9C	–	BC	LDY ABS,X	DC	–	FC	–
9D	STA ABS,X	BD	LDA ABS,X	DD	CMP ABS,X	FD	SBC ABS,X
9E	–	BE	LDX ABS,Y	DE	DEC ABS,X	FE	INC ABS,X
9F	–	BF	–	DF	–	FF	–

Toelichting. Direkt na de opcode volgt de mnemonics van de instruktie. Deze bestaat uit drie hoofdletters. Indien er meerdere adresseringsmogelijkheden zijn voor een bepaalde instruktie volgt nog een aanduiding van de relevante adresseringsmetode. De betekenis is als volgt:

IMM	Onmiddellijke adressering (immediate addressing)
ABS	Absolute adressering (absolute addressing)
Z	Adressering op pagina nul (zero page addressing)
A	Accu-adressering (Accumulator addressing)
(IND,X)	X-geïndexeerde indirekte adressering (indexed indirect addressing)
(IND),Y	Indirekte Y-geïndexeerde adressering (indirect indexed addressing)
Z,X	X-geïndexeerd adresseren op pagina nul (zero page indexed,X addressing)
Z,Y	Y-geïndexeerd adresseren op pagina nul (zero page indexed,Y addressing)
ABS,X	Absolute X-geïndexeerde adressering (absolute indexed,X addressing)
ABS,Y	Absolute Y-geïndexeerde adressering (absolute indexed,Y addressing)
IND	Indirekte adressering (indirect addressing)

2 Instruktie-overzicht

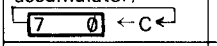
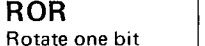
De 56 instrukties van de 6502-microprocessor in alfabetische volgorde. Voor een groot aantal instrukties zijn meerdere adresseringsmetoden mogelijk. Dit levert in totaal 151 verschillende opcodes op.

mnemonics + omschrijving	adresserings-metode(n)	hex opcode	aantal klokpulsen (N)	aantal bytes	beïnvloede vlag(gen)
ADC Add memory to accumulator with carry A + M + C → A (1)	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	69 6D 65 61 71 75 7D 79	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	NV - - - - ZC
AND "AND" memory with accumulator A ∧ M → A (1)	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	29 2D 25 21 31 35 3D 39	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - Z -
ASL Shift left one bit (accu or memory) C ← 7 … Ø ← Ø	ABS Z A Z,X ABS,X	ØE Ø6 ØA 16 1E	6 5 2 6 7	3 2 1 2 3	N - - - - - ZC
BCC Branch on carry clear (2) Branch on C = Ø	REL	9Ø	2	2	- - - - - - - -
BCS Branch on carry set (2) Branch on C = 1	REL	BØ	2	2	- - - - - - - -
BEQ Branch on result zero (2) Branch on Z = 1	REL	FØ	2	2	- - - - - - - -
BIT Test bits in memory: A ∧ M M_7 → N; M_6 → V	ABS Z	2C 24	4 3	3 2	M_7M_6 - - - - Z -
BMI Branch on result minus (2) Branch on N = 1	REL	3Ø	2	2	- - - - - - - -

mnemonics + omschrijving	adresserings-metode(n)	hex opcode	aantal klokpulsen (N)	aantal bytes	beïnvloede vlag(gen)
BNE Branch on result not zero (2) Branch on Z = Ø	REL	DØ	2	2	- - - - - - - -
BPL Branch on result plus (2) Branch on N = Ø	REL	1Ø	2	2	- - - - - - - -
BRK Force break forced interrupt	IMP	ØØ	7	1	- - - 1 - 1 - - B I
BVC Branch on overflow clear (2) Branch on V = Ø	REL	5Ø	2	2	- - - - - - - -
BVS Branch on overflow set (2) Branch on V = 1	REL	7Ø	2	2	- - - - - - - -
CLC Clear carry flag Ø → C	IMP	18	2	1	- - - - - - - Ø C
CLD Clear decimal mode; Ø → D	IMP	D8	2	1	- - - - Ø - - - D
CLI Clear interrupt flag; Ø → I	IMP	58	2	1	- - - - - Ø - - I
CLV Clear overflow flag; Ø → V	IMP	B8	2	1	Ø - - - - - - - V
CMP Compare memory and accumulator A-M	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	C9 CD C5 C1 D1 D5 DD D9	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - ZC
CPX Compare memory and index X X-M	IMM ABS Z	EØ EC E4	2 4 3	2 3 2	N - - - - - ZC
CPY Compare memory and index Y M-Y	IMM ABS Z	CØ CC C4	2 4 3	2 3 2	N - - - - - ZC

mnemonics + omschrijving	adresserings-metode(n)	hex opcode	aantal klokpulsen (N)	aantal bytes	beïnvloede vlag(gen)
DEC Decrement memory by one M-1 → M	ABS Z Z,X ABS,X	CE C6 D6 DE	6 5 6 7	3 2 2 3	N - - - - - Z -
DEX Decrement index X by one X-1 → X	IMP	CA	2	1	N - - - - - Z -
DEY Decrement index Y by one Y-1 → Y	IMP	88	2	1	N - - - - - Z -
EOR "Exclusive or" memory with accumulator A ∀ M → A (1)	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	49 4D 45 41 51 55 5D 59	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - Z -
INC Increment memory by one M + 1 → M	ABS Z Z,X ABS,X	EE E6 F6 FE	6 5 6 7	3 2 2 3	N - - - - - Z -
INX Increment index X by one X + 1 → X	IMP	E8	2	1	N - - - - - Z -
INY Increment index Y by one Y + 1 → Y	IMP	C8	2	1	N - - - - - Z -
JMP Jump to new location (PC + 1) → PCL (PC + 2) → PCH	ABS IND	4C 6C	3 5	3 3	- - - - - - - -
JSR Jump to new location saving return address PC + 2 ↓ (PC + 1) → PCL (PC + 2) → PCH	ABS	20	6	3	- - - - - - - -

mnemonics + omschrijving	adresserings-metode(n)	hex opcode	aantal klokpulsen (N)	aantal bytes	beïnvloede vlag(gen)
LDA Load accumulator with memory M → A (1)	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	A9 AD A5 A1 B1 B5 BD B9	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - Z -
LDX Load index X with memory M → X (1)	IMM ABS Z Z,Y ABS,Y	A2 AE A6 B6 BE	2 4 3 4 4	2 3 2 2 3	N - - - - - Z -
LDY Load index Y with memory M → Y (1)	IMM ABS Z Z,X ABS,X	A0 AC A4 B4 BC	2 4 3 4 4	2 3 2 2 3	N - - - - - Z -
LSR Shift right one bit (memory or accumulator) 0 → [7 0] → C	ABS Z A Z,X ABS,X	4E 46 4A 56 5E	6 5 2 6 7	3 2 1 2 3	0 - - - - - ZC N
NOP No operation	IMP	EA	2	1	- - - - - - - -
ORA "OR" memory with accumulator AVM → A	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	09 0D 05 01 11 15 1D 19	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - Z -
PHA Push accumulator on stack A↓	IMP	48	3	1	- - - - - - - -
PHP Push processor status on stack; P↓	IMP	08	3	1	- - - - - - - -
PLA Pull accumulator from stack A↑	IMP	68	4	1	N - - - - - Z -

mnemonics + omschrijving	adresserings-metode(n)	hex opcode	aantal klokpulsen (N)	aantal bytes	beïnvloede vlag(gen)
PLP Pull processor status from stack; P↑	IMP	28	4	1	(hersteld)
ROL Rotate one bit left (memory or accumulator)  	ABS Z Z,X ABS,X A	2E 26 36 3E 2A	6 5 6 7 2	3 2 2 3 1	N - - - - - ZC
ROR Rotate one bit right (memory or accumulator)  	ABS Z A Z,X ABS,X	6E 66 6A 76 7E	6 5 2 6 7	3 2 1 2 3	N - - - - - ZC
RTI Return from interrupt PC↑ ; P↑	IMP	4Ø	6	1	(hersteld)
RTS Return from subroutine PC↑; PC + 1 → PC	IMP	6Ø	6	1	- - - - - - - -
SBC Subtract memory from accumulator with borrow (3) A-M-$\overline{C}$ → A	IMM ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	E9 ED E5 E1 F1 F5 FD F9	2 4 3 6 5 4 4 4	2 3 2 2 2 2 3 3	N - - - - - ZC
SEC Set carry flag 1 → C	IMP	38	2	1	- - - - - - - 1
SED 1 → D Set decimal mode	IMP	F8	2	1	- - - - 1 - - - D
SEI Set interrupt disable; 1 → I	IMP	78	2	1	- - - - - 1 - - I
STA Store accumulator in memory A → M	ABS Z (IND,X) (IND),Y Z,X ABS,X ABS,Y	8D 85 81 91 95 9D 99	4 3 6 6 4 5 5	3 2 2 2 2 3 3	- - - - - - - -

mnemonics + omschrijving	adresserings-metode(n)	hex opcode	aantal klokpulsen (N)	aantal bytes	beïnvloede vlag(gen)
STX Store index X in memory; X → M	ABS Z Z,Y	8E 86 96	4 3 4	3 2 2	- - - - - - - -
STY Store index Y in memory; Y → M	ABS Z Z,X	8C 84 94	4 3 4	3 2 2	- - - - - - - -
TAX Transfer accumulator to index X A → X	IMP	AA	2	1	N - - - - - Z -
TAY Transfer accumulator to index Y A → Y	IMP	A8	2	1	N - - - - - Z -
TSX Transfer stack pointer to index X S → X	IMP	BA	2	1	N - - - - - Z -
TXA Transfer index X to accumulator X → A	IMP	8A	2	1	N - - - - - Z -
TXS Transfer index X to stack pointer X → S	IMP	9A	2	1	N - - - - - Z -
TYA Transfer index Y to accumulator Y → A	IMP	98	2	1	N - - - - - Z -

Opmerkingen

(1) Aan N 1 toevoegen indien paginagrens wordt overschreden

(2) Aan N 1 toevoegen indien de sprong plaatsvindt naar dezelfde pagina;
aan N 2 toevoegen indien de sprong plaatsvindt naar een andere pagina

(3) Borrow = non-carry ($\overline{C}$)

IMM	Onmiddellijke (immediate) adressering
ABS	Absolute (direkte) adressering
Z	Adressering op pagina nul (zero page)
A	Accu-adressering
IMP	Ingebouwde (implied) adressering
(IND,X)	X-geïndexeerde indirekte adressering
(IND),Y	Indirekte Y-geïndexeerde adressering
Z,X	X-geïndexeerde adressering op pagina nul
Z,Y	Y-geïndexeerde adressering op pagina nul
ABS,X	Absolute X-geïndexeerde adressering
ABS,Y	Absolute Y-geïndexeerde adressering
REL	Relatieve adressering
IND	Indirekte adressering

3 Hex dump van de EPROM

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
1C00: 85 F3 68 85 F1 68 85 EF 85 FA 68 85 F0 85 FB 84
1C10: F4 86 F5 BA 86 F2 A2 01 86 FF 4C 33 1C A9 1E 8D
1C20: 83 1A A9 04 85 F1 A9 03 85 FF 85 F6 A2 FF 9A 86
1C30: F2 D8 78 20 88 1D D0 FB 20 88 1D F0 FB 20 88 1D
1C40: F0 F6 20 F9 1D C9 13 D0 13 A6 F2 9A A5 FB 48 A5
1C50: FA 48 A5 F1 48 A6 F5 A4 F4 A5 F3 40 C9 10 D0 06
1C60: A9 03 85 FF D0 14 C9 11 D0 06 A9 00 85 FF F0 0A
1C70: C9 12 D0 09 E6 FA D0 02 E6 FB 4C 33 1C C9 14 D0
1C80: 0B A5 EF 85 FA A5 F0 85 FB 4C 7A 1C C9 15 10 EA
1C90: 85 E1 A4 FF D0 0D B1 FA 0A 0A 0A 0A 05 E1 91 FA
1CA0: 4C 7A 1C A2 04 06 FA 26 FB CA D0 F9 A5 FA 05 E1
1CB0: 85 FA 4C 7A 1C 20 D3 1E A4 E3 A6 E2 E8 D0 01 C8
1CC0: 86 E8 84 E9 A9 77 A0 00 91 E6 20 4D 1D C9 14 D0
1CD0: 2A 20 6F 1D 10 F7 85 FB 20 6F 1D 10 F0 85 FA 20
1CE0: D3 1E A0 00 B1 E6 C5 FB D0 07 C8 B1 E6 C5 FA F0
1CF0: D9 20 5C 1E 20 F8 1E 30 E9 10 3E C9 10 D0 0A 20
1D00: 20 1E 10 C9 20 47 1E F0 C1 C9 13 D0 14 20 20 1E
1D10: 10 BB 20 5C 1E 20 F8 1E A5 FD 85 F6 20 47 1E F0
1D20: A9 C9 12 D0 07 20 F8 1E 30 A0 10 0D C9 11 D0 09
1D30: 20 83 1E 20 EA 1E 4C CA 1C A9 EE 85 FB 85 FA 85
1D40: F9 A9 03 85 F6 20 8E 1D D0 FB 4C CA 1C A2 02 A0
1D50: 00 B1 E6 95 F9 C8 CA 10 F8 20 5C 1E 20 8E 1D D0
1D60: FB 20 8E 1D F0 FB 20 8E 1D F0 F6 20 F9 1D 60 20
1D70: 5C 1D C9 10 10 11 0A 0A 0A 0A 85 FE 20 5C 1D C9
1D80: 10 10 04 05 FE A2 FF 60 A0 00 B1 FA 85 F9 A9 7F
1D90: 8D 81 1A A2 08 A4 F6 A5 FB 20 CC 1D 88 F0 0D A5
1DA0: FA 20 CC 1D 88 F0 05 A5 F9 20 CC 1D A9 00 8D 81
1DB0: 1A A0 03 A2 00 A9 FF 8E 82 1A E8 E8 2D 80 1A 88
1DC0: D0 F5 A0 06 8C 82 1A 09 80 49 FF 60 48 84 FC 4A
1DD0: 4A 4A 4A 20 DF 1D 68 29 0F 20 DF 1D A4 FC 60 A8
1DE0: B9 0F 1F 8D 80 1A 8E 82 1A A0 7F 88 10 FD 8C 80
1DF0: 1A A0 06 8C 82 1A E8 E8 60 A2 21 A0 01 20 B5 1D
1E00: D0 07 E0 27 D0 F5 A9 15 60 A0 FF 0A B0 03 C8 10
1E10: FA 8A 29 0F 4A AA 98 10 03 18 69 07 CA D0 FA 60
1E20: 20 6F 1D 10 21 85 FB 20 60 1E 84 F7 84 FD C6 F7
1E30: F0 12 20 6F 1D 10 0F 85 FA C6 F7 F0 07 20 6F 1D
1E40: 10 04 85 F9 A2 FF 60 20 A6 1E 20 DC 1E A2 02 A0
1E50: 00 B5 F9 91 E6 CA C8 C4 F6 D0 F6 60 A0 00 B1 E6
```

```
      Ø  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
1E6Ø: AØ Ø1 C9 ØØ FØ 1A C9 4Ø FØ 16 C9 6Ø FØ 12 AØ Ø3
1E7Ø: C9 2Ø FØ ØC 29 1F C9 19 FØ Ø6 29 ØF AA BC 1F 1F
1E8Ø: 84 F6 6Ø A5 E6 85 EA A5 E7 85 EB A4 F6 B1 EA AØ
1E9Ø: ØØ 91 EA E6 EA DØ Ø2 E6 EB A5 EA C5 E8 DØ EC A5
1EAØ: EB C5 E9 DØ E6 6Ø A5 E8 85 EA A5 E9 85 EB AØ ØØ
1EBØ: B1 EA A4 F6 91 EA A5 EA C5 E6 DØ Ø6 A5 EB C5 E7
1ECØ: FØ 1Ø 38 A5 EA E9 Ø1 85 EA A5 EB E9 ØØ 85 EB 4C
1EDØ: AE 1E 6Ø A5 E2 85 E6 A5 E3 85 E7 6Ø 18 A5 E8 65
1EEØ: F6 85 E8 A5 E9 69 ØØ 85 E9 6Ø 38 A5 E8 E5 F6 85
1EFØ: E8 A5 E9 E9 ØØ 85 E9 6Ø 18 A5 E6 65 F6 85 E6 A5
1FØØ: E7 69 ØØ 85 E7 38 A5 E6 E5 E8 A5 E7 E5 E9 6Ø 4Ø
1F1Ø: 79 24 3Ø 19 12 Ø2 78 ØØ 1Ø Ø8 Ø3 46 21 Ø6 ØE Ø2
1F2Ø: Ø2 Ø2 Ø1 Ø2 Ø2 Ø2 Ø1 Ø1 Ø2 Ø1 Ø1 Ø3 Ø3 Ø3 Ø3 6C
1F3Ø: 7A 1A 6C 7E 1A B1 E6 AØ FF C4 EE FØ ØD D1 EC DØ
1F4Ø: ØA 88 B1 EC AA 88 B1 EC AØ Ø1 6Ø 88 88 88 DØ E9
1F5Ø: 6Ø 38 A5 E4 E9 FF 85 EC A5 E5 E9 ØØ 85 ED A9 FF
1F6Ø: 85 EE 2Ø D3 1E 2Ø 5C 1E AØ ØØ B1 E6 C9 FF DØ 1D
1F7Ø: C8 B1 E6 A4 EE 91 EC 88 A5 E7 91 EC 88 A5 E6 91
1F8Ø: EC 88 84 EE 2Ø 83 1E 2Ø EA 1E 4C 65 1F 2Ø F8 1E
1F9Ø: 3Ø D3 2Ø D3 1E 2Ø 5C 1E AØ ØØ B1 E6 C9 4C FØ 16
1FAØ: C9 2Ø FØ 12 29 1F C9 1Ø FØ 1A 2Ø F8 1E 3Ø E6 A9
1FBØ: Ø3 85 F6 4C 33 1C C8 2Ø 35 1F FØ EE 91 E6 8A C8
1FCØ: 91 E6 DØ E6 C8 2Ø 35 1F FØ EØ 38 E5 E6 38 E9 Ø2
1FDØ: 91 E6 4C AA 1F D8 A9 ØØ 85 FB 85 FA 85 F9 2Ø 6F
1FEØ: 1D 1Ø F2 85 FB 2Ø 6F 1D 1Ø EB 85 FA 18 A5 FA E5
1FFØ: FB 85 F9 C6 F9 4C DE 1F FF FF 2F 1F 1D 1C 32 1F
```

U ziet hier in één oogopslag de inhoud van de 1024 geheugenplaatsen van de EPROM IC2 in hexadecimale vorm. Om precies te zijn: in de vorm van 64 rijen van elk 16 bits. Helemaal links treft u voor elke rij het adres aan van de geheugenplaats waarin het byte van de eerste kolom (kolom Ø) moet komen. Boven elke kolom ziet u het kolomnummer (Ø . . . F) staan.

4 Van hexadecimaal naar decimaal en omgekeerd

In de tabellen staan alle 256 getallen, die met twee hexadecimale cijfers kunnen worden weergegeven. Boven elke kolom staat het rechter cijfer en links van elke rij het linker cijfer van het hexadecimaal weergegeven getal. De bovenste helft is voor positieve getallen, de onderste helft voor negatieve getallen (twee-komplementnotatie).
Een paar voorbeelden:

59_{10} = $3B
-92_{10} = $A4
$27 = 39_{10}
$F8 = -8_{10}

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

5 Aansluitingen van de konnektors

De aansluitingen van de uitbreidingskonnektor

Om praktische redenen is de konnektor 90 graden gedraaid. De a-aansluitingen bevinden zich het dichtst bij de print. Een zij-aanzicht van konnektor plus print levert dus een van rechts naar links oplopende nummering op.
De blokjes boven 32c en onder 1c geven de oriëntatienokken aan, die nodig zijn voor het op de juiste manier aansluiten van de "male"-konnektor.

32a	massa	32a	o	o	32c	32c massa
31a	RAM-R/W	31a	o	o	31c	31c niet gebruikt
30a	Φ1	30a	o	o	30c	30c EX
29a	K1	29a	o	o	29c	29c R/$\overline{W}$
28a	niet gebruikt	28a	o	o	28c	28c K2
27a	Φ2	27a	o	o	27c	27c niet gebruikt
26a	A1	26a	o	o	26c	26c AØ
25a	A3	25a	o	o	25c	25c A2
24a	A5	24a	o	o	24c	24c A4
23a	A7	23a	o	o	23c	23c A6
22a	A9	22a	o	o	22c	22c A8
21a	A11	21a	o	o	21c	21c AØ
20a	A13	20a	o	o	20c	20c A12
19a	A15	19a	o	o	19c	19c A14
18a	−5 V	18a	o	o	18c	18c K3
17a	K4	17a	o	o	17c	17c +12 V
16a	naar 16c	16a	o	o	16c	16c met 16a doorverbonden,
15a	K5	15a	o	o	15c	15c K6
14a	K7	14a	o	o	14c	14c S0
13a	niet gebruikt	13a	o	o	13c	13c niet gebruikt
12a	IRQ	12a	o	o	12c	12c NMI
11a	niet gebruikt	11a	o	o	11c	11c niet gebruikt
10a	D7	10a	o	o	10c	10c D6
9a	D5	9a	o	o	9c	9c D4
8a	D3	8a	o	o	8c	8c D2
7a	D1	7a	o	o	7c	7c DØ
6a	niet gebruikt	6a	o	o	6c	6c niet gebruikt
5a	RES	5a	o	o	5c	5c RDY
4a	massa	4a	o	o	4c	4c massa
3a	niet gebruikt	3a	o	o	3c	3c niet gebruikt
2a	niet gebruikt	2a	o	o	2c	2c niet gebruikt
1a	+5 V	1a	o	o	1c	1c +5 V

De aansluitingen van de poortkonnektor