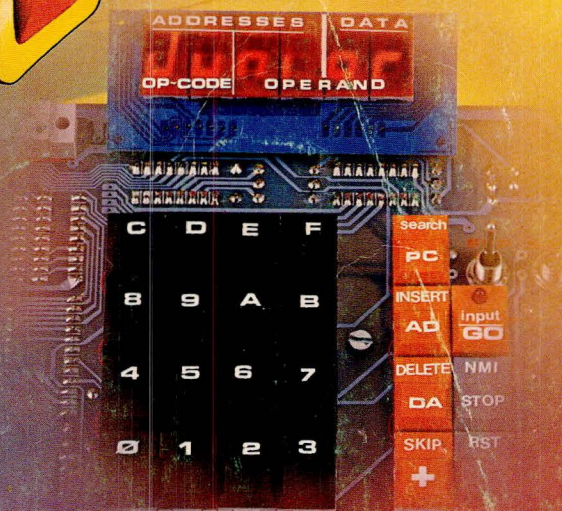


# een volwassen computer voor beginners

deel 1

# JUNIOR COMPUTER



uitgeversmij.  
elektuur b.v.



*F. N. van't Hof*

# Junior-computer 1

een volwassen computer  
voor beginners

A.Nachtmann  
G.H.Nachbar

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

© UITGEVERSMATSCHAPPIJ ELEKTUUR B.V. 1980  
Overname van de inhoud van dit boek of een deel daarvan  
zonder schriftelijke toestemming van de uitgeefster is ver-  
boden.

**AUTEURSRECHT**

De auteursrechtelijke bescherming van de inhoud strekt zich  
mede uit tot de illustraties met inbegrip van de printed  
circuits en tot de ontwerpen daarvoor.

In verband met Artikel 30 Rijksoktrooiwet mogen de op-  
genomen schakelingen slechts voor partikuliere of weten-  
schappelijke doeleinden worden vervaardigd en niet in of  
voor een bedrijf.

Het toepassen van schakelingen geschiedt buiten verant-  
woordelijkheid van de uitgeefster.

**NADRUKRECHT:**

Voor Duitsland: Elektor Verlag GmbH, 5133 Gangel 1.

Voor Groot Brittannië: Elektor Publishers Ltd. Canterbury.

Voor Frankrijk: Elektor sarl LeDoulieu, 59940 Estaires.

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 micro-computer 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 te zijnertijd, 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 inziens niet: de opzet van de junior-computer is compact; 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 complete, 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*

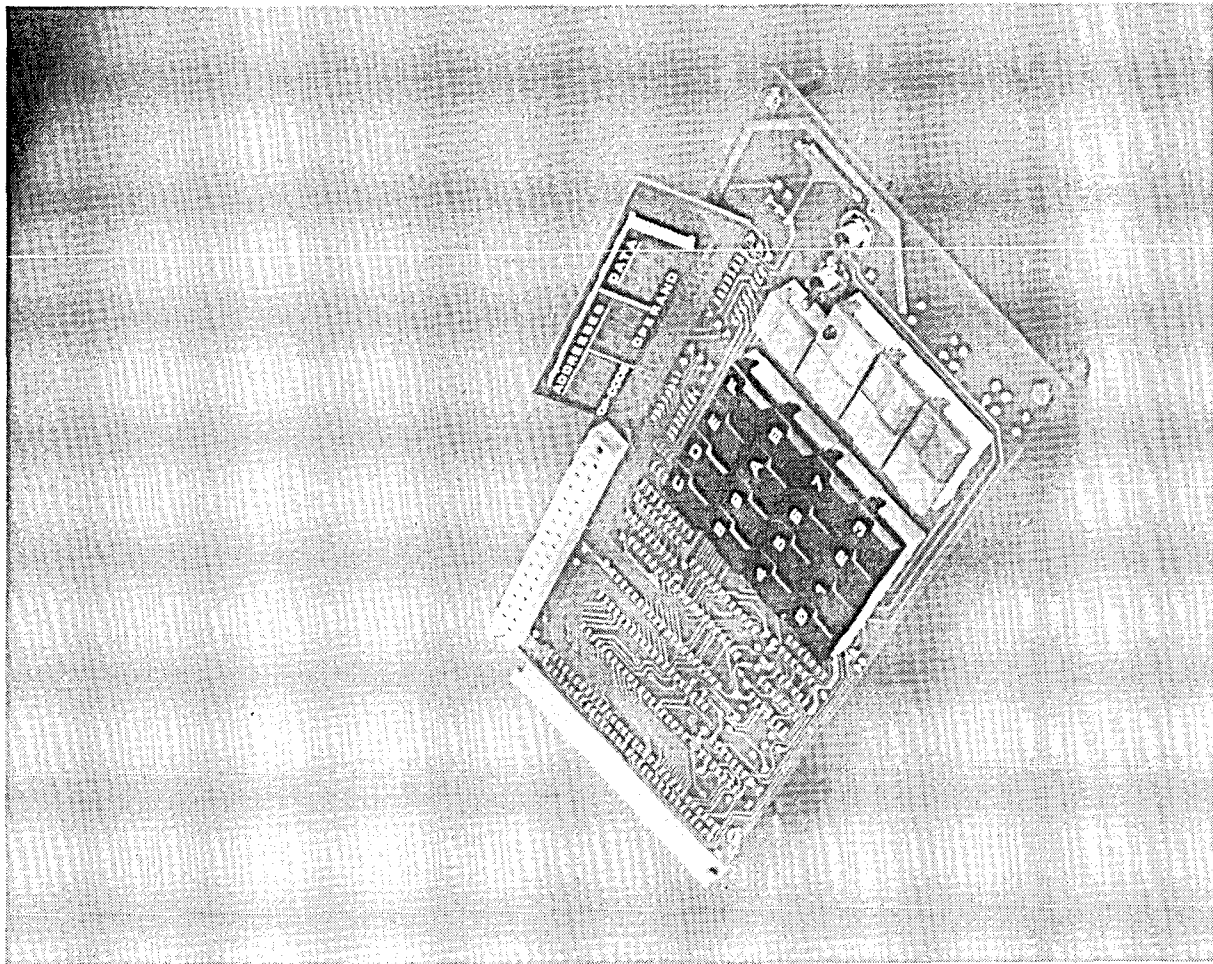
*Junior-computer 2* is het vervolg van dit boek. Het boek is gewijd aan:

- 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
- Grotere voorbeeld-programma's
- De listing van het monitorprogramma

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

# Inhoud

<b>Hoofdstuk 1</b> .....	<b>7</b>
<i>Eerste kennismaking – Waarom en hoe?</i>	
<b>Hoofdstuk 2</b> .....	<b>37</b>
<i>Digitale vingeroefeningen – digi-taal en digi-rekenen</i>	
<b>Hoofdstuk 3</b> .....	<b>57</b>
<i>Programmeren – omgangsvormen</i>	
<b>Hoofdstuk 4</b> .....	<b>125</b>
<i>Alle begin is eenvoudig – programmeren zonder poespas</i>	
<b>Aanhangsel 1</b> .....	<b>149</b>
<i>Opcodes in hexadecimale volgorde</i>	
<b>Aanhangsel 2</b> .....	<b>150</b>
<i>Instructie-overzicht</i>	
<b>Aanhangsel 3</b> .....	<b>156</b>
<i>Hex dump van het monitorprogramma</i>	



# 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 "vles 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)?

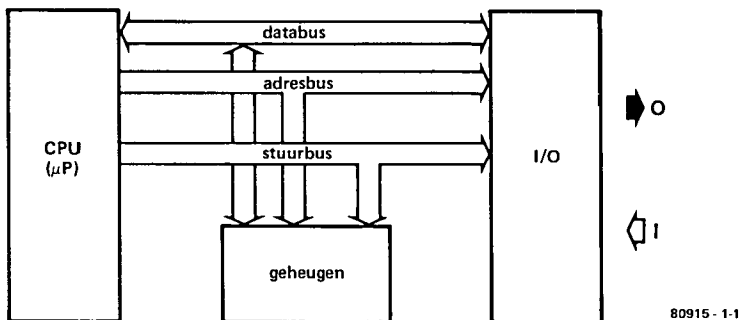
## Het zwart maken van de doos

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. Simpel 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 betreffelijke 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 instructies 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.

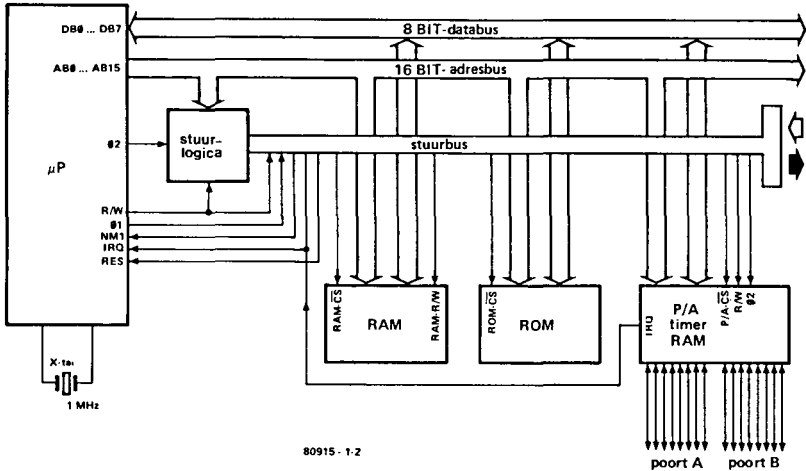
In een computer is data de grondstof voor de bewerkingen, maar ook de in kode vastgelegde omschrijving van de bewerkingen (programma). Die grondstof is voorbereid, namelijk in elektrisch-digitale vorm gegoten. Er is dataverkeer tussen de drie blokken, zoals het ophalen door de CPU van een volgende opdracht, 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 selectie 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 ( $\mu P$ ), 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 instructies. 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 concrete computer: de junior-computer.

heet bidirektioneel. Echter nooit in twee richtingen tegelijk, dus eenrichtingsverkeer in de ene of in de andere, tegengestelde richting. Van invloed op de richting is het lees/schrijfsignaal  $R/\bar{W}$  = Read/ $\bar{W}$ rite 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 hoofdsorten; 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<sup>1</sup>. 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 ook 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 contacten 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 poortselectie 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 $\mu P$ : 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 complete systeem. De klokgenerator levert twee signalen

---

<sup>1</sup>*Naast de ROM, die tijdens het fabricageproces 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 "fusible 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 instructie 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  $\mu$ P. Voor het adres van de volgende uit te voeren programma-instructie is de stand van een bepaalde interne teller van belang, die de uitvoering van de elkaar opvolgende instructies regelt. Adressen van data als grondstof, halffabrikaat of eindproduct 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 . . .*

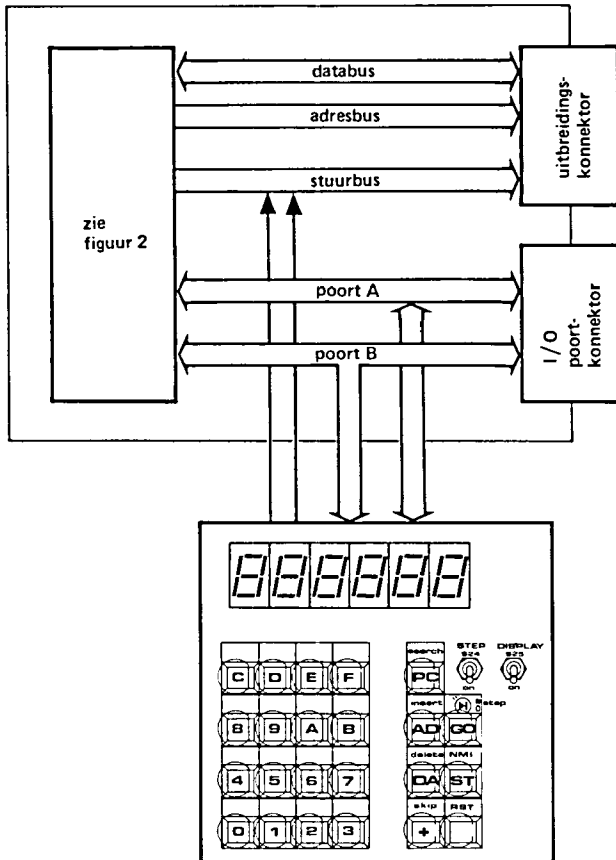
In het blok PIA van figuur 2 zit ook nog een programmeerbare timer. Deze komt in een apart hoofdstuk aan de orde, hoofdstuk 5. Het blok stuurlogica bevat een zogenaamde adresdekoder. Hiermee worden de diverse chip-selekt-signalen gemaakt, die een bepaald geheugendeel of de PIA klaar zetten voor gebruik, zodat dat men ermee kan lezen en schrijven.

Tot zover de bespreking in blokken van het kwa functie 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 compleet. 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: contacten 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 conversatie verhogen, of het aantal conversatiemogelijkheden. 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 complete 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 complete junior-computer elektronisch in elkaar zit; het prinsipeschema 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 instructieset) en om een hoop andere, nog te openbaren software-mogelijkheden.

### *Alles op z'n tijd*

Voor de  $\mu P$  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 microseconde een stel elektrische prikkels wordt geleverd: een  $\emptyset 1$  gevolgd door een  $\emptyset 2$ , die van belang zijn voor het prepareren van resp. de adresbus en de databus. Dat zijn er nagenoeg een miljoen per seconde.

### *Bussen vol enen en nullen*

Vanuit de processor IC1 vertrekt de adresbus. Die bestaat uit de lijnen A0...A15. Voor de databus, de lijnen D0...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...A0 resp. D7...D0 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 bitpatronen. Dat het "alfabet" slechts uit twee "letters" (0 en 1) bestaat komt omdat met binaire (digitale) logika wordt gewerkt.

## Geheugenorganisatie

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) programmeergeheugen, 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 bevroren (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, A0 . . . 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.

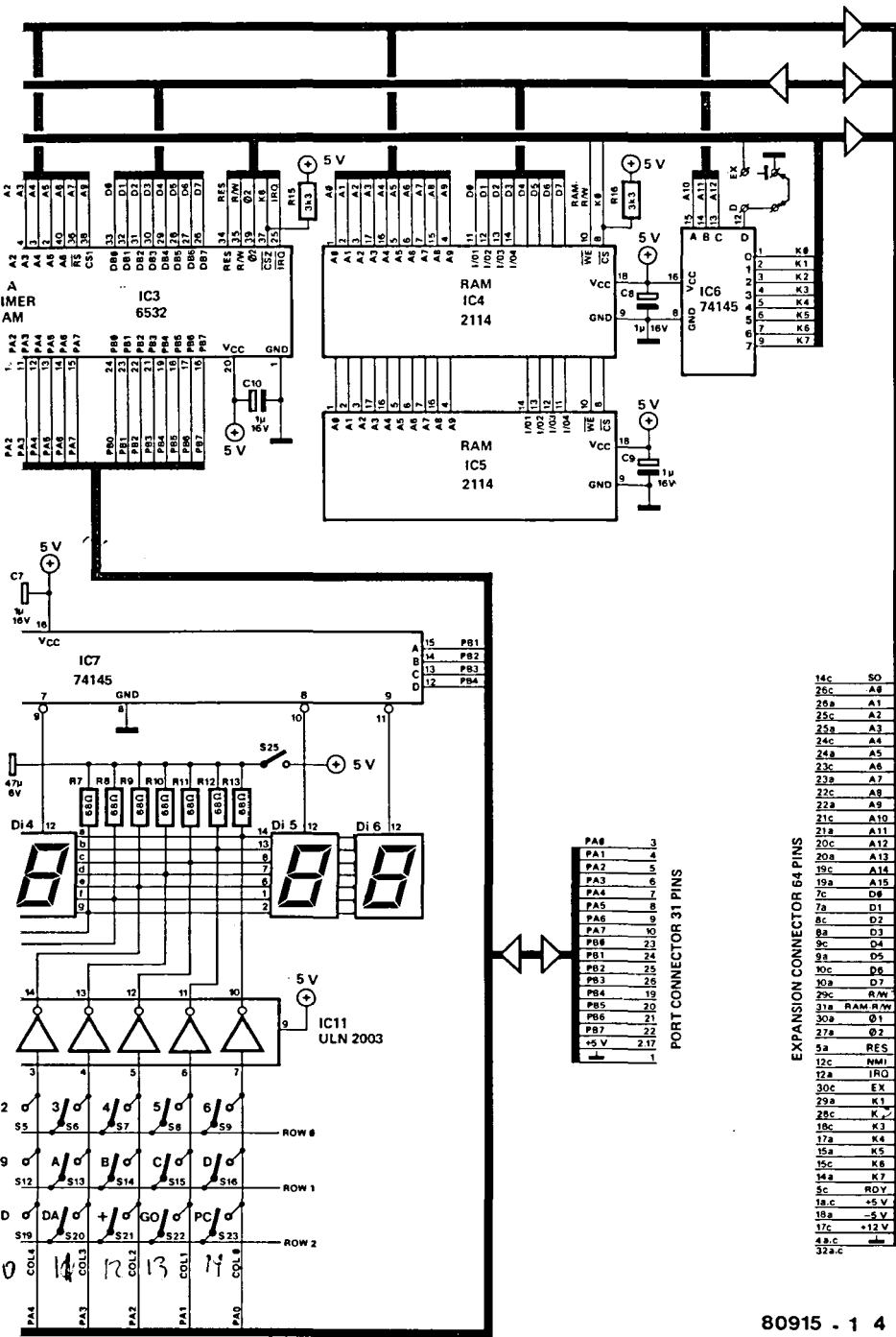
De geheugens krijgen ook nog het een en ander van de stuurbus. Selectiesignalen bijvoorbeeld. De EPROM en de RAM's hebben dezelfde adresbus-aansluitingen: A0 . . . A9. Hoe maakt men onderscheid tussen de twee geheugens? Dat gaat via de signalen  $\overline{CS}$ , van ChipSelect, die als leidingen K7 voor de EPROM resp. K0 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 Elektoor-notatie: 65K536.

Met 64 chipselectlijnen zijn dus alle geheugenblokken met tien adresaan-sluitingen te selekteren. In de standaarduitvoering van de junior-computer worden er maar acht gemaakt. Namelijk de signalen K0 . . . K7 van de adresdekoder IC6, die de drie eerstvolgende adreslijnen A10 . . . A12 krijgt toegevoerd. Drie van de acht selectiesignalen worden intern gebruikt. Naast de al besproken twee is dat K6, die de RAM in de PIA, IC3 selek-teert. Ter wille van het overzicht geven we nu een soort adresboek:

A15 . . . A13	A12	A11	A10	A9 . . . A0	aktief:	geheugenblok:
X	0	0	0	X	K0	1k RAM (IC4, IC5)
X	0	0	1	X	K1	1k extern
X	0	1	0	X	K2	1k extern
X	0	1	1	X	K3	1k extern
X	1	0	0	X	K4	1k extern
X	1	0	1	X	K5	1k extern
X	1	1	0	X	K6	RAM in PIA (IC3)
X	1	1	1	X	K7	1k EPROM (IC2)



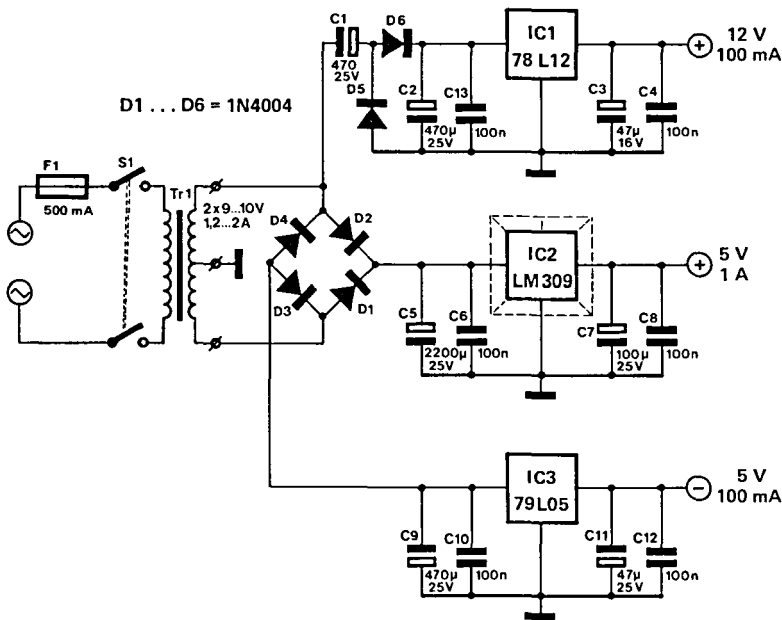


**PORT CONNECTOR 31 PINS**

PA8	3
PA1	4
PA2	5
PA3	6
PA4	7
PA5	8
PA6	9
PA7	10
PB8	23
PB1	24
PB2	25
PB3	26
PB4	19
PB5	20
PB6	21
PB7	22
+5 V	2.17
1	1

**EXPANSION CONNECTOR 64 PINS**

14c	50
26c	A8
25a	A7
25c	A2
25a	A3
24c	A4
24a	A5
23c	A6
23a	A7
22c	A8
22a	A9
21c	A10
21a	A11
20c	A12
20a	A13
19c	A14
19a	A15
7c	D8
7a	D1
8c	D2
8a	D3
9c	D4
9a	D5
10c	D6
10a	D7
29c	R/W
31a	RAM R/W
30a	01
27a	02
5a	RES
12c	NMI
12a	IRQ
30c	EX
29a	K1
28c	K2
16c	K3
17a	K4
15a	K5
15c	K6
14a	K7
5c	RDY
18c	+5 V
18a	-5 V
17c	+12 V
4a.c	32a.c



80915 - 1-5

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 combinaties van enen en nullen.

Van de 64k geheugenkapaciteit is dus 8k standaard-decodeerbaar. Voor een groter decodeerbereik is uitbreiding van de adresdecoder 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. Daàr komt u nog wel achter.

Voor het gebruik van de RAM is nóg een stuursignaal nodig. Een signaal dat onderscheid maakt tussen lezen en schrijven. Een RAM-R/W-signaal dus, dat 1 is bij lezen en 0 bij schrijven. Het signaal komt uit poort N6 en ontstaat uit een combinatie van het R/W-signaal van de  $\mu P$  en de  $\emptyset 2$ -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  $\mu P$  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 "pull-up-" 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 I/O-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 monitorprogramma 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 uitbreidingsconnector.

Verder bevat de stuurbus de kloksignalen  $\emptyset 1$  en  $\emptyset 2$ , waarvan de laatste nodig is voor het functioneren van de PIA. En een  $R/\bar{W}$ -lijn die niet door  $\emptyset 2$  wordt beïnvloed, zoals de RAM-R/ $\bar{W}$ . 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 S0, 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 adresdekoderbereik.

### *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 richtingsregister 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 A0 . . . A6. Het signaal  $\bar{R}\bar{S}$  (= 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. Adreslijn A7 fungeert als wisselwachter  $\bar{R}\bar{S}$ . Bij de RAM zijn twee chipselect-signalen betrokken: het uit de adresdekoder afkomstige signaal K6 dat als 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 A0, 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 besturingsfuncties, 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 geactiveerd 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 zo ja welke? Het programma herkent de toets en handelt overeenkomstig de toetsfunctie. Het effect van contactdender, dat is het herhaald contact-geen contact 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 toen 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 PA0 . . . 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 communicatie met toetsenbord en display in de vorm van "ieder op zijn beurt" (*multiplexing*) een elegante methode tot materiaalbesparing op. Minder hardware

dus ten koste van iets meer software. Want bezette geheugenplaatsen zijn goedkoper dan bezette IC-plaatsen.

Rest nog S24, 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 communiceren. 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 (op IC11 na. Die krijgt helemaal niets) 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. Wel 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 componenten 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. vijftienveertig 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 componenten. 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<sup>1</sup>, 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 vòòr montage van de onderdelen alle verticale doorverbindingen te controleren. Dat kan met een multimeter (ohm-meting). Of akoestisch, met de goedkope methode 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 methode 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 verticale draadverbinding (zie figuur 14b) of via de aansluitdraad van een component. N.B. Zet men de print vast in een bank-schroefje, 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 onderdelenplaattegrond 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 wipchakelaars en (eventueel) de 31-polige poortkonnektor. De onderkant is bestemd voor de montage van de weerstanden, condensatoren, 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 "componenten boven" is er hier sprake van de omgekeerde wereld, maar in elektronisch opzicht is er geen bezwaar tegen als de componenten "hangen" in plaats van "zitten".

---

<sup>1</sup> *In verband met de beperkte afmetingen van dit boek is de hoofdprint verkleind afgedrukt. Voor het zelf maken van printen zijn afdrucken 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 componenten worden in een bepaalde volgorde op de print gemonteerd. En wel als volgt (zie figuur 7 en de onderdelenlijst voor de hoofprint):

*Een.* De weerstanden R1 . . . R20 worden op de print gemonteerd. Na het solderen meteen de overtollige draadeindjes zo dicht mogelijk bij de las wegnippen. 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  $\Omega$ : oranje-oranje-bruin-(goud)  
68  $\Omega$ : 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 dood-enkele keer met 1% (bruin) of 2% (rood).

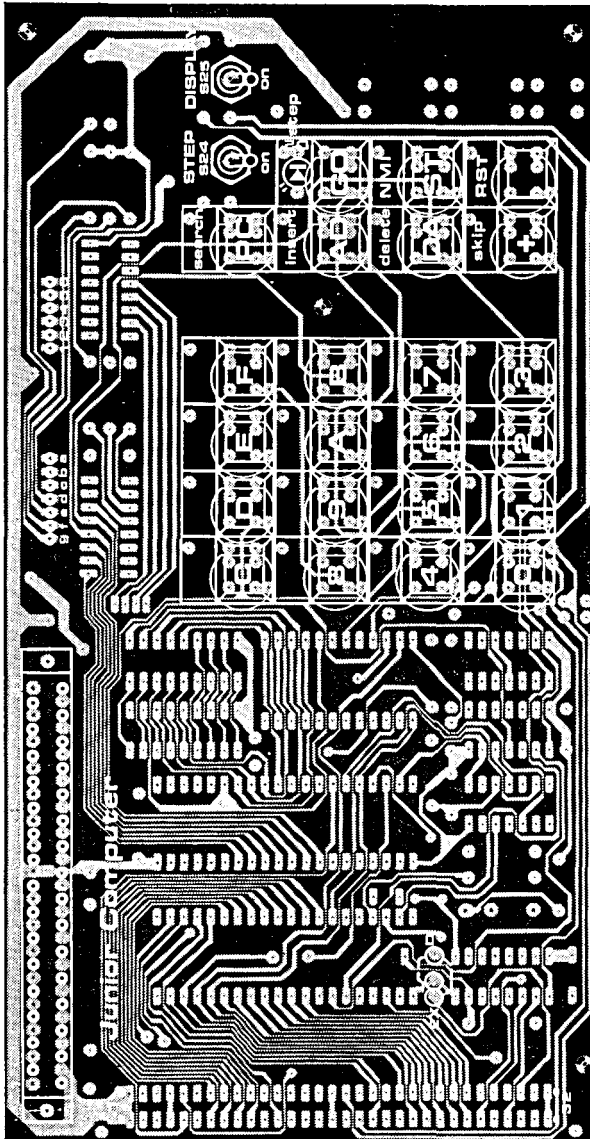
*Twee.* Bij de montage van de diode D1 moeten we goed letten op de polariteit. Meestal, zeg maar altijd is de katode-aansluiting (de punt van de driehoek) aangegeven door een ring. In geval van twijfel is het laatste woord aan de universeelmeter (ohm-meting uitvoeren met twee aansluitmogelijkheden van de meetpennen; bij de meeste multimeters is de min-pen de plus).

*Drie.* De condensatoren C1, C3 en C4 worden gemonteerd.

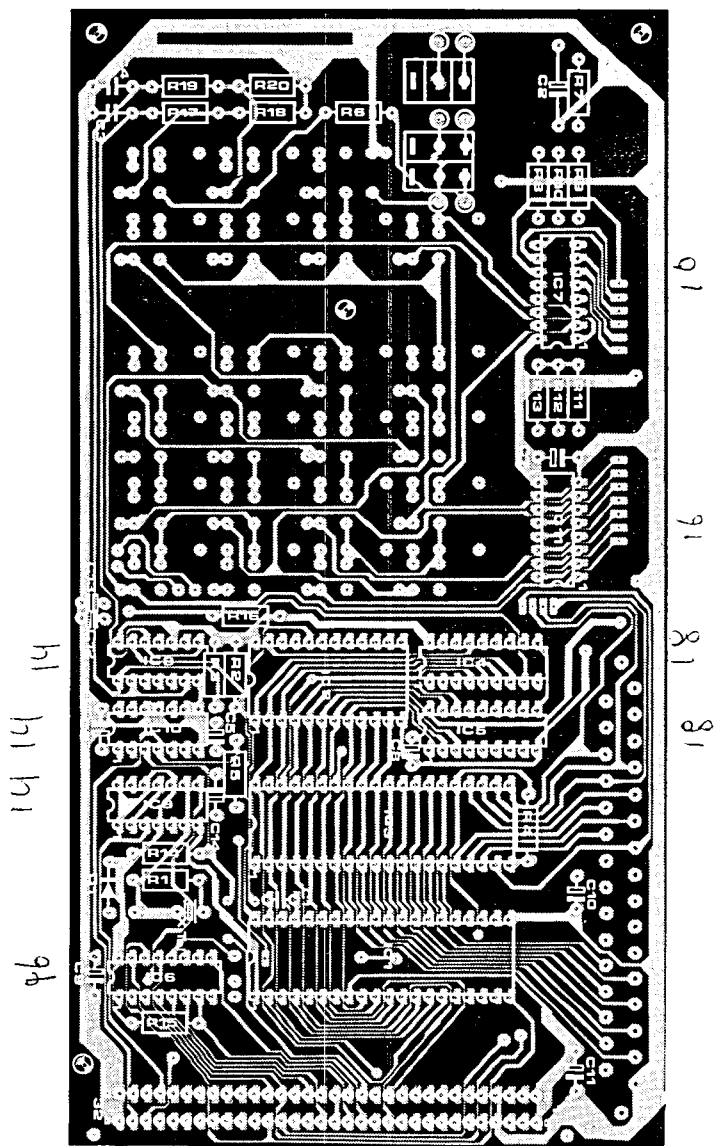
*Vier.* De elko's C2 en C5 . . . C14 gaan de print op. Kun je normaal gesproken een weerstand of condensator op twee manieren monteren, hier is het donders goed opletten geblazen. Elko's hebben een plus-aansluiting en een min-aansluiting. Op de componenten-plattegrond is de plus aangegeven met een niet opgevulde balk ( $\square$ ) en de min-aansluiting met een opgevulde balk ( $\blacksquare$ ). 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 direct 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.

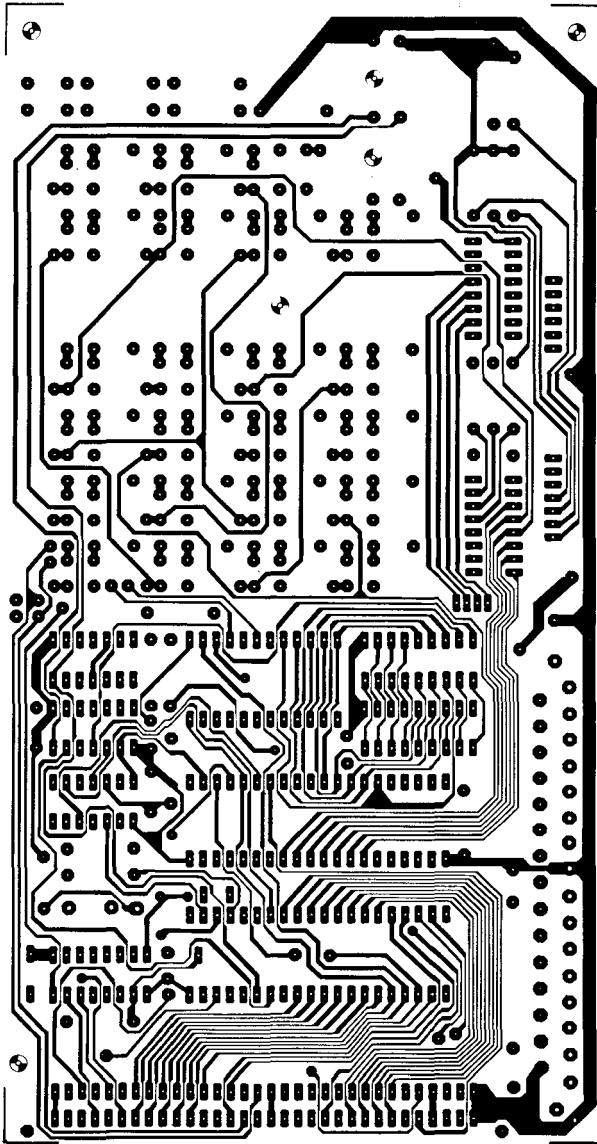
*Nog eens vijf.* Een IC kan op twee manieren op de print worden gesoldeerd of in een voetje worden gestoken. *Slechts één manier is de juiste.* Op de onderdelenplattegrond is een IC weergegeven met een rechthoek, een van de korte zijden vertoont een "putje". Een soortgelijk putje of markering van aansluitpen 1 toont het IC. De beide oriëntatiepunten moeten aan de



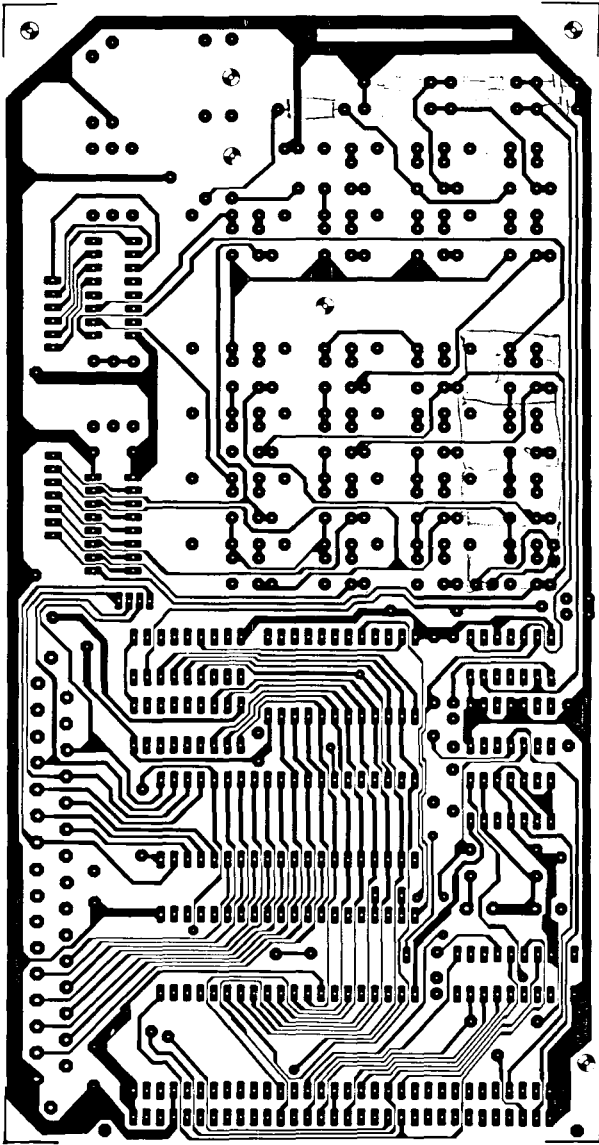
Figuur 6. Onderdelenplattegrond met koperbanenpatroon van de bovenkant van de dubbelzijdige hoofdprint (EPS 80089-1).



Figuur 7. Onderdelenplaattegrond met koperbanenpatroon van de onderkant van de hoofdprint. Aan deze kant zitten alle weerstanden, condensatoren, 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 componenten-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  $\Omega$   
R7, R8, R9, R10, R11, R12,  
R13 = 68  $\Omega$   
R17, R19 = 2k2  
R18, R20 = 68 k

### Kondensatoren:

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

### Halfgeleiders:

IC1 = 6502 (Rockwell, MOS)  
IC2 = 2708  
IC3 = 6532 (Rockwell, MOS)  
IC4, IC5 = 2114  
IC6, IC7 = 74145

IC8 = 556

9,65 IC9 = 74LS00, 7400, 74LS132  
9 IC10 = 74LS01, 7401  
1,10 IC11 = ULN2003 (Sprague)  
D1 = 1N4148  
D2 = LED rood

### Diversen:

4410 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 female 31-polig;  
DIN 41617 (optie) (poort-  
konnektor)

(bij afwezigheid van uitbreidings-  
konnektor:) minstens vier  
aansluitpennetjes  $\varnothing = 0,8$  mm,  
voor de voedingsaansluitingen  
1 print EPS 80089-1

z3, 30  
:menf 44  
:menf 43

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 L 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 contact is tussen het toetshuis en de print. Diode D2 wordt gemonteerd vòòr 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.

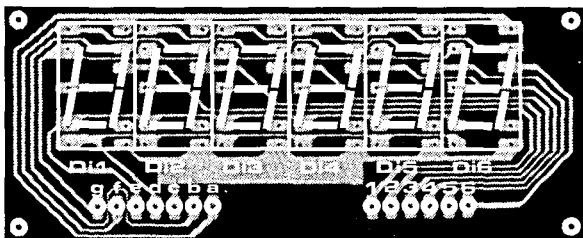
*Zo, naar de hoofdprint hebben we voorlopig geen omkijken meer. Of toch niet? Het kan geen kwaad om de print na te kijken op rondslingerend soldeertin, dat kan zorgen voor onbedoelde elektrische verbindingen. Ook de positie van de IC's en elko's nog eens goed nakijken!*

### *De display-print*

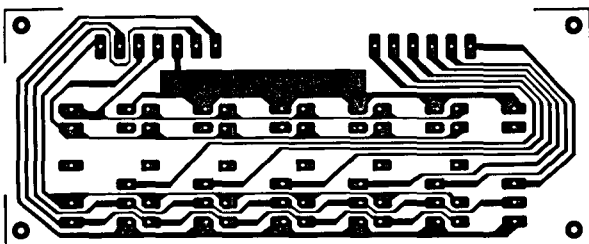
Daar zijn we eigenlijk gauw mee klaar. De onderdelenplattegrond staat in figuur 10, het koperbanenwerk in figuur 11. Eigenlijk gaat het maar om twee dingen: het monteren van de zes zevensgment-displays Di1 . . . Di6 en het monteren van deze print op de hoofdprint. De displays leveren geen problemen op 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 elf draadjes van 0,8 mm-montagedraad en van 1½ à 2 cm en soldeert die in de elf 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 elf 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 elf 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.

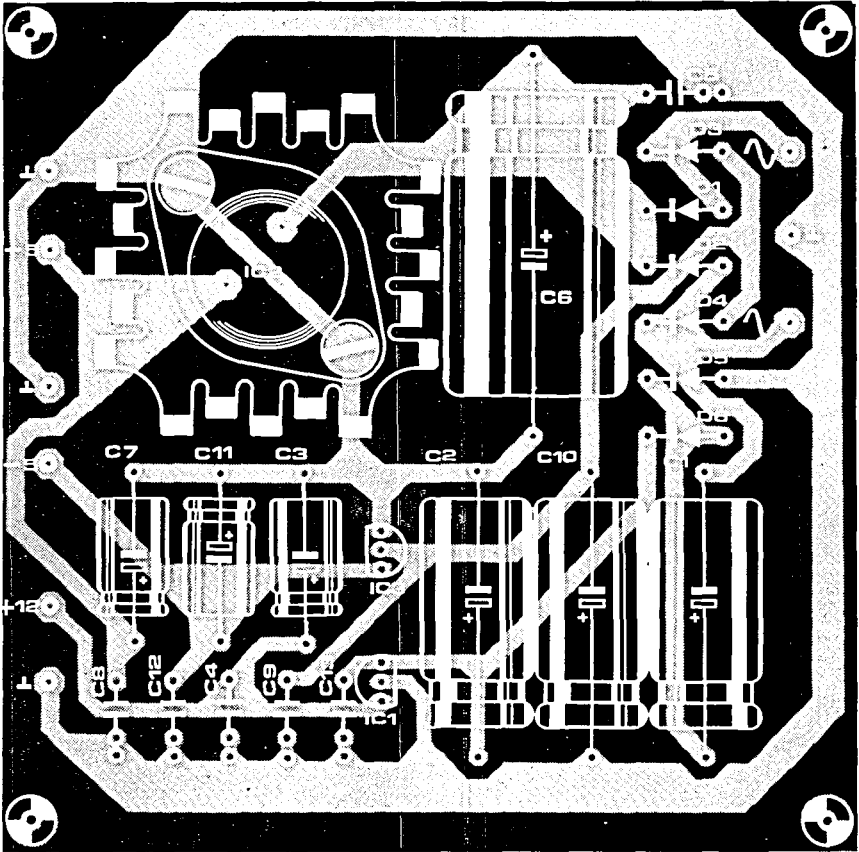
Onderdelenlijst bij de display-print van de junior-computer. De componenten-layout staat in figuur 10 en de koperlayout in figuur 11.

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. Onderdelenplaattegrond van de voedingsprint (EPS 80089-3).

**Onderdelenlijst bij de voedingsprint van de junior-computer.**

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

**Kondensatoren:**

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

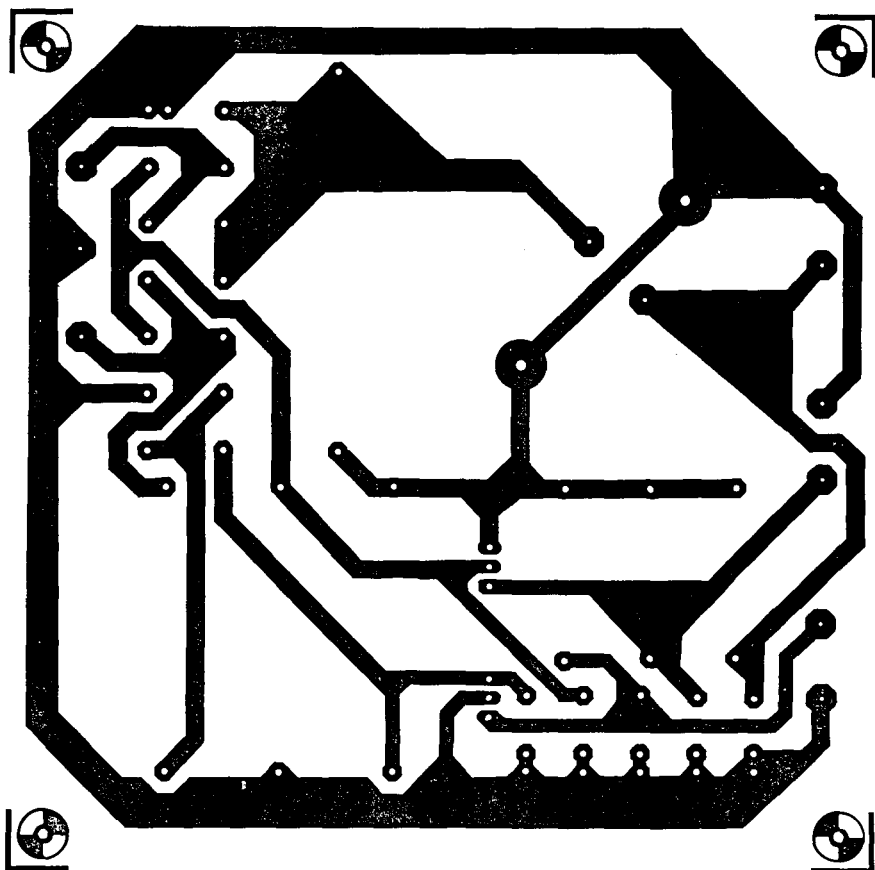
**Halfgeleiders:**

- 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



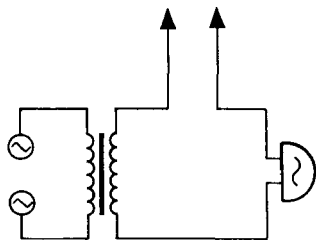
Figuur 13. Koperbanenpatroon van de voedingsprint.

*En dan nog . . .*

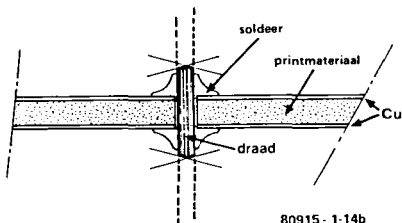
De drie printen zijn nu alle volgebouwd. Rest nog de aansluiting van de voedingstransformator Tr1 op de voedingsprint en, via de netschakelaar S1 en de zekering F1, op het lichtnet (netsteker nog niet in de kontaktdoos steken!) In afwachting van het definitieve inkasten kunnen de verbindingen provisorisch 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 methode 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 verticale draadbrug.

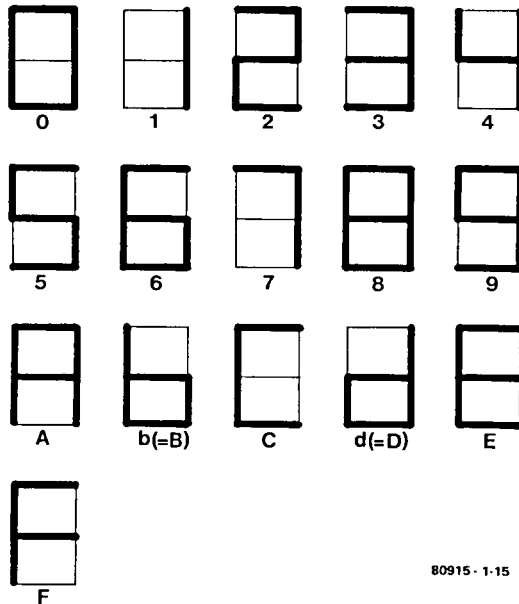
Eerst is het nogmaals grondig controleren van alle drie printen een zeer verstandige zaak. Laat er desnoods ook eens een ander naar kijken. Dan gaat de netsteker de muur in. *Let wel: de voeding is nog niet aangesloten op de hoofdprint!* De voeding wordt ingeschakeld met de netschakelaar S1 en we meten met een multimeter (DC-spanningsmeting) de drie voedingsspanningen ten opzichte van massa (= nul), op de aansluitpennen van de voedingsprint. De spanningen moeten binnen  $\pm 5\%$  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 combinaties 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 spronginstructie: 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.



80915 - 1-15

**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, ontstaan 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 contacten tussen het konnektor-“mannetje” (male) en het “vrouwtje”, en tussen IC en IC-voetje. Een grondige inspectie van met elkaar contact te maken delen is geboden. Eventueel reinigen met alcohol o.i.d. Zorg ervoor dat de beide delen ook mechanisch contact met elkaar hebben (aandrukken)
- (bij zelf gemaakte printen) Valse kopersporen (lees: kortsluitingen) als gevolg van onvolledige etsing of haren op de print-layout. Alles controleren 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 herhalingsfrequentie 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 sloop scherm doodse stilte dan staat de klok stil. Mogelijke verdachte componenten 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 wél sterk is? U kunt dan contact opnemen met de redactie 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 afstands-bussen) 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!!*

# 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:

11110111101

Dat 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  $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^{11}$  verschillende mogelijkheden, als we uitgaan van twee mogelijkheden (0 of 1) per cijfers. 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  $10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 \times 10 = 10^{11}$  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 = $1 \times 10^3 = 1000 \times 1$	1
900 = 9 honderdtallen = $9 \times 10^2 = 100 \times 9$	9
80 = 8 tientallen = $8 \times 10^1 = 10 \times 8$	8
+ 1 = 1 eenheden = $1 \times 10^0 = 1 \times 1$	1
+ <u>1981</u>	1981

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

1024 = 1 1024-tal = $1 \times 2^{10} = 1024 \times 1$	1
512 = 1 512-tal = $1 \times 2^9 = 512 \times 1$	1
256 = 1 256-tal = $1 \times 2^8 = 256 \times 1$	1
128 = 1 128-tal = $1 \times 2^7 = 128 \times 1$	1
0 = geen 64-tal = $0 \times 2^6 = 64 \times 0$	0
32 = 1 32-tal = $1 \times 2^5 = 32 \times 1$	1
16 = 1 16-tal = $1 \times 2^4 = 16 \times 1$	1
8 = 1 8-tal = $1 \times 2^3 = 8 \times 1$	1
4 = 1 4-tal = $1 \times 2^2 = 4 \times 1$	1
0 = geen 2-tal = $0 \times 2^1 = 2 \times 0$	0
+ 1 = 1 eenheid = $1 \times 2^0 = 1 \times 1$	1
+ <u>1981</u>	11110111101

1 = wel een bijdrage van die macht van 2  
 0 = geen bijdrage van die macht van 2

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  $2, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9$ , enzovoorts, dat er maar twee symbolen mogelijk zijn voor elk cijfer van het getal, namelijk 0 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 *binair 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 (0). 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 0. 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

0 = 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 0 tegenovergesteld definieert)

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 kan 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 acties. 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 gelijke getal 111011101 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<sup>1</sup>, schuilt het enorme vermogen van de computer om moeilijke problemen of handelingen te vertalen in een min of meer groot aantal doodsimplere 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: 0 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 zoets als bitpatronen. Denk daarbij aan een binair getal of aan de een of andere code (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 code van een bepaalde soort opdracht (instructie), 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 code waarmee de decimale cijfers 0 . . . 9 wordt vastgelegd. Over deze *BCD-code* spreken we later nog.

---

<sup>1</sup> 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 instructie.
- dan heb je de zogenaamde ASCII-kode van zeven bits voor het coderen 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 0 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 combinerend moet het duidelijk zijn dat de overgang van tweetallig naar zestientallig moet leiden tot een flinke reductie van het aantal cijfers dat nodig is om een binair getal of digitale kode weer te geven. We zullen zien dat de reductie 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 0 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=0000	4=4=0100	8= 8=1000	C=12=1100
1=1=0001	5=5=0101	9= 9=1001	D=13=1101
2=2=0010	6=6=0110	A=10=1010	E=14=1110
3=3=0011	7=7=0111	B=11=1011	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 aaneelkaar. Bijvoorbeeld:

$$(0) \underbrace{1101}_6 \underbrace{0101}_A \underbrace{0111}_B \underbrace{1100}_C = 6ABC_{16}$$

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 codes) 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:

$$C \times 16^0 + B \times 16^1 + A \times 16^2 + 6 \times 16^3 = 12 \times 1 + 11 \times 16 + 10 \times 256 + 6 \times 4096 = \text{decimaal } 27.324.$$

U kunt dit controleren 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 methoden aan om een decimaal getal om te zetten in een binair getal. Het getal wordt herhaald door twee. Deelt men een oneven getal door twee, dan levert dit een 1 op; een even getal geeft een 0. Het is mogelijk om decimale getallen direct 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:

$$\begin{array}{r} 129 \text{ eerste getal} \\ 243 \text{ tweede getal} \\ 1 \text{ overdracht} \\ + \text{---} \\ 372 \text{ resultaat = de som} \end{array}$$

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



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:

$$\begin{aligned} 1+0 &= 1 \\ 0+1 &= 1 \\ 1+1 &= 0, \text{ met carry, dus } 01+01=10 \\ 0+0 &= 0 \end{aligned}$$

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:

$$\begin{array}{r} 1984 \text{ aftrektal} \\ 199 \text{ aftrekker} \\ 11 \text{ geleend} \\ \hline 1785 \text{ resultaat = het verschil} \end{array}$$

Binair gaat het zo:

$$\begin{array}{r} 11000001 \text{ aftrektal} \\ 1111110 \text{ aftrekker} \\ 111111 \text{ geleend} \\ \hline 01000011 \text{ verschil} \end{array}$$

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

$$\begin{aligned} 0-0 &= 0 \\ 1-1 &= 0 \\ 1-0 &= 1 \\ 0-1 &= 1 \text{ met borrow, dus } 10-01=01 \end{aligned}$$

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:

$$\begin{array}{r} 233 \\ 147 \\ \hline 1631 \text{ x} \\ 932 \\ 233 \\ + \\ \hline 34251 \end{array}$$

Eerst is het resultaat van  $7 \times 233$  genoteerd; vervolgens het resultaat van  $4 \times 233$ , maar dan in zijn geheel een plaats naar links verschoven, zodat het in wezen gaat om  $40 \times 233$ . Tenslotte het resultaat van  $1 \times 233$ , twee plaatsen naar links verschoven; in feite dus  $100 \times 233$ . Het vermenigvuldigen van binaire getallen gaat precies zo. Zelfs nog eenvoudiger, omdat slechts vermenigvuldigen met 1 of 0 voorkomt. Een voorbeeld, waarbij de getallen 1011 en 1010 met elkaar worden vermenigvuldigd:

$$\begin{array}{r}
 1011 \quad (\text{kontrolle:=11}) \\
 1010 \quad (\text{kontrolle:=10}) \\
 \hline
 \phantom{1011}0000 \\
 1011 \\
 \phantom{1011}0000 \\
 1011 \\
 \phantom{1011}1 \\
 \hline
 1101110 \quad (\text{kontrolle:=110})
 \end{array}$$

Moet men hexadecimale getallen met elkaar vermenigvuldigen (of bij elkaar 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.

### *Binaire delen*

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

$$\begin{array}{r}
 17 \overline{)2091} \backslash 123 \\
 \underline{17} \phantom{0} \\
 39 \phantom{0} \\
 \underline{34} \phantom{0} \\
 51 \\
 \underline{51} \\
 0
 \end{array}$$

Binair gaat het net zo. Simpelers 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).

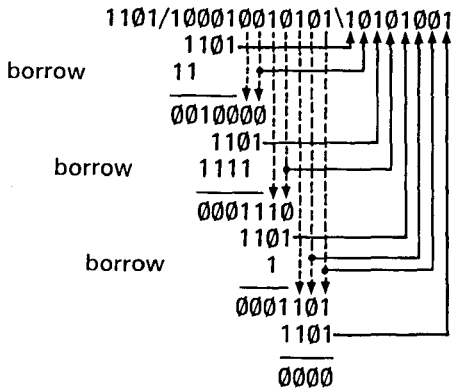
Is daarentegen de rest kleiner dan de deler, dan noteert men een nul en plakt er aan de rechter kant het volgende bit van het deeltal aan.

We geven nu een voorbeeld.

Het deeltal is 1000100101, decimaal 2197

De deler is 1101, decimaal 13

Het resultaat is 10101001, 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 methoden om negatieve getallen weer te geven. We beperken ons tot de bespreking van de *twee-komplement-methode (two's complement)*. Maar eerst even iets anders.

11111111		255
11111110		254
11111101		253
11111100		252
11111011		251
11111010		250
11111001		249
11111000		248
~		
01101011		107
01101010		106
01101001		105
01101000		104
01100111		103
~		
00001000		8
00000111		7
00000110		6
00000101		5
00000100		4
00000011		3
00000010		2
00000001		1
00000000		0

**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:

$$\begin{array}{r}
 \phantom{\text{borrow}} \phantom{00000000} \\
 \phantom{\text{borrow}} \phantom{00000000} \\
 \text{borrow } 00000000 \\
 \hline
 \phantom{\text{borrow}} 11111111 \quad - \\
 \phantom{\text{borrow}} 11111111 \quad = \text{nul min een} = \text{min een} = -1 \\
 \phantom{\text{borrow}} 00000001 \quad (\text{nog eens 1 aftrekken}) \\
 \hline
 \text{borrow } 11111110 \quad - \\
 \phantom{\text{borrow}} 00000001 \quad = -2 \\
 \phantom{\text{borrow}} \phantom{00000000} \quad (\text{nog eens 1 aftrekken}) \\
 \hline
 \text{borrow } \phantom{11111110} 1 \quad - \\
 \phantom{\text{borrow}} 11111101 \quad = -3
 \end{array}$$

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-methode, c.q. -notatie. Het is niet de bedoeling om u te vermoeien met allerlei getaltheorieën 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

01111111		127	\$7F
01111110		126	\$7E
01111101		125	\$7D
~			
00000111		7	\$07
00000110		6	\$06
00000101		5	\$05
00000100		4	\$04
00000011		3	\$03
00000010		2	\$02
00000001		1	\$01
00000000		0	\$00
11111111		-1	\$FF
11111110		-2	\$FE
11111101		-3	\$FD
11111100		-4	\$FC
11111011		-5	\$FB
11111010		-6	\$FA
11111001		-7	\$F9
11111000		-8	\$F8
~			
10000010		-126	\$82
10000001		-127	\$81
10000000		-128	\$80

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

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

### 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" code. 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 codes 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
11111110	- 2	FE
11111101	- 3	FD
11111100	- 4	FC
11111011	- 5	FB
11111010	- 6	FA
11111001	- 7	F9
11111000	- 8	F8
11110111	- 9	F7
11110110	-10	F6
11110101	-11	F5
11110100	-12	F4
11110011	-13	F3
11110010	-14	F2
11110001	-15	F1
11110000	-16	F0
11101111	-17	EF
11101110	-18	EE
11101101	-19	ED
11101100	-20	EC
11101011	-21	EB
11101010	-22	EA
11101001	-23	E9
11101000	-24	E8
11100111	-25	E7
11100110	-26	E6
11100101	-27	E5
11100100	-28	E4
11100011	-29	E3
11100010	-30	E2
11100001	-31	E1
11100000	-32	E0
11011111	-33	DF
11011110	-34	DE
11011101	-35	DD
11011100	-36	DC
11011011	-37	DB
11011010	-38	DA
11011001	-39	D9
11011000	-40	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 code van het getal 1981. Zoals we al weten is die:

11110111101

Bij de BCD-kode is het inzicht in de decimale waarde van het getal groter dan bij de binaire code. 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.

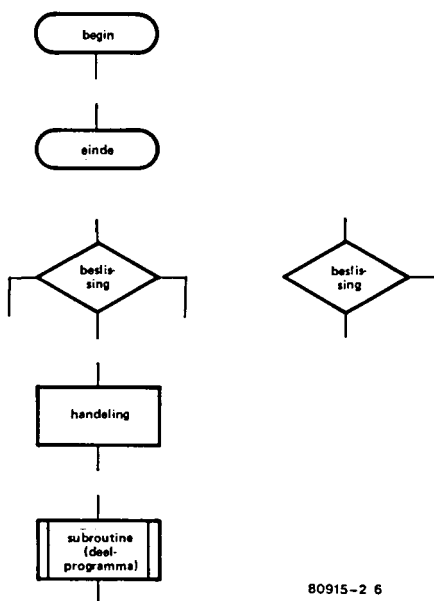
### Het probleem in kaart gebracht: het stroomdiagram

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 (instructies). Het vertalen van de richtlijnen in konkrete instructies 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 instructies voor de (junior-)computer.



80915-2 6

**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 functie 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*

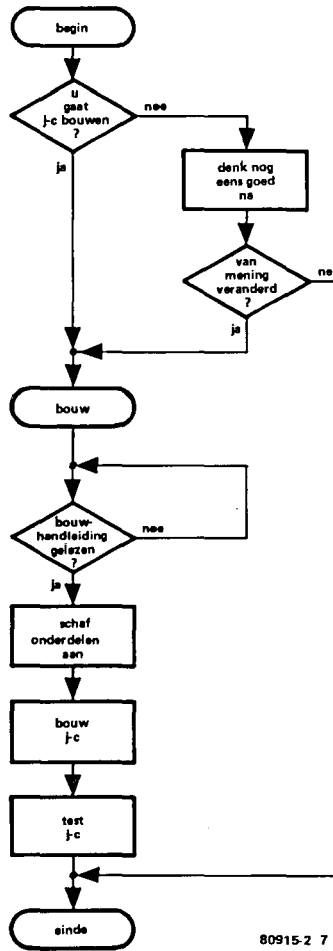
Ter illustratie van het begrip stroomdiagram geven we nu een min of meer uitgewerkt voorbeeld van zo'n stroomdiagram. Het staat in figuur 7 en gaat over de oplossing van een "probleem" dat u als u dit leest misschien al heeft "opgelost": de bouw van de junior-computer, zoals dat in hoofdstuk 1 is beschreven.

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.

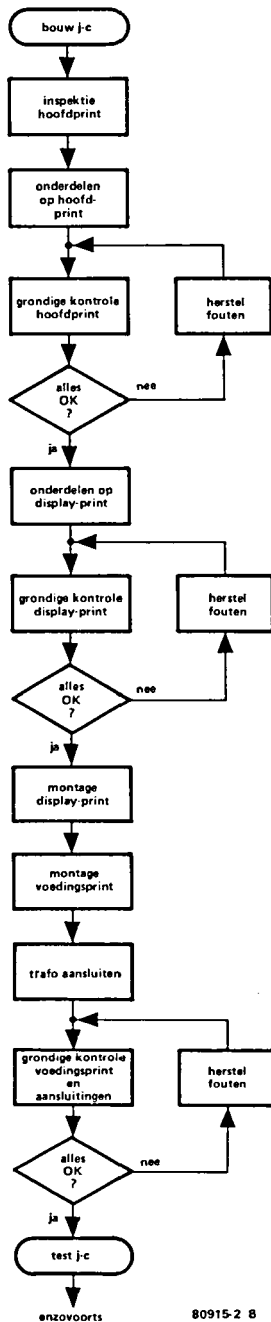
We gaan nog een stapje verder. Tot de handeling "onderdelen op hoofdprint" van figuur 8 hoort de montage van de elko C12. Bij deze laatste handeling hoort het stroomdiagram van figuur 9. Een drietal handelingen



80915-2 7

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

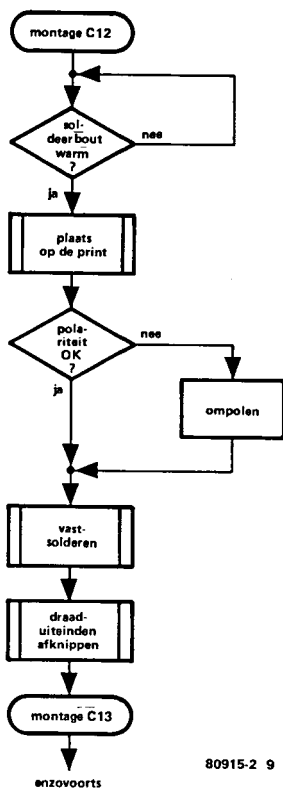
van "montage C12" is in de vorm van een subroutine gegoten. Dat houdt verband met het feit dat de handelingen in kwestie ook van toepassing zijn op een groot aantal andere, op printen te monteren onderdelen. In dat geval hebben subroutines belangrijke voordelen boven normale routines. Het waarom staat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



enovoorts

80915-2 8

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



80915-2 9

**Figuur 9.** Een deel van het stroomdiagram van de handeling "onderdelen op hoofd-print" 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  $110101010$ ?
- 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  $10101010 + 1010101$ ?
- 17). Hoeveel is  $01110100 - 1101$ ?
- 18). Hoeveel is  $11110000 - 1111$ ?
- 19). Hoeveel is  $10111000 - 10000001$ ?
- 20). Hoeveel is  $10101111 - 10101111$ ?
- 21). Hoeveel is  $100 - 1111011$ ?
- 22). Hoeveel is  $11110001 \times 01111$ ?
- 23). Hoeveel is  $101 \times 11111111$ ?
- 24). Hoeveel is  $1010 \times 1010$ ?
- 25). Hoeveel is  $11 \times 11111111$ ?
- 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  $10011011110110010 : 1001110$ ?
- 30). Hoeveel is  $\$B9A0 : \$0B$ ?

## Antwoorden

- 1). 2F
- 2). 1F
- 3). 147
- 4). 1AA
- 5). 2
- 6). 0B
- 7). 000100110010
- 8). 101000000010100
- 9). 0000001101010110
- 10). 1010010101100001
- 11). 1010101110111010
- 12). 100010110
- 13). 101110010
- 14). 100000000
- 15). 11111111
- 16). 11111111
- 17). 1100111
- 18). 11100001
- 19). 00110111
- 20). 0
- 21). 100001001 (-247 in 9 bits twee-komplementnotatie)
- 22). 11100001111
- 23). 10011111011
- 24). 1100100
- 25). 1011111101
- 26). 101010111011101000 = \$2AEE8
- 27). 10010011
- 28). 100000000
- 29). 1111111111
- 30). 1000011100000 = \$10E0

# 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 geactiveerd. Als gevolg daarvan wordt door de computer gekeken of een van de toetsen is ingedrukt. Gaat het om een van de toetsen 0...F, dan wordt de toetswaarde in beeld gebracht.

De vier linker displays geven hexadecimaal de adressen weer. In principe zijn alle adressen van 0000 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 020018, op het volgende adres A9, enzovoorts. We gaan dan als volgt te werk:

				adres	data			
RST				xxxx	xx			
AD				xxxx	xx			
0	2	0	0	0200	xx			
DA				0200	xx			
		1	8	0200	18	0200	1 8	CLC
+		A	9	0201	A9	0201	A 9	LDA
+		0	3	0202	03	0202	0 3	
+		6	9	0203	69	0203	6 9	ADC
+		0	4	0204	04	0204	0 4	
+		8	D	0205	8D	0205	8 D	STA
+		0	0	0206	00	0206	0 0	
+		0	3	0207	03	0207	0 3	
+		4	C	0208	4C	0208	4 C	JMP
+		0	0	0209	00	0209	0 0	
+		0	3	020A	03	020A	0 3	
AD						020B	X X	
1	A	7	A	1A7A	xx	020C	X X	
DA				1A7A	xx	1A79	X X	
		0	0	1A7A	00	1A7A	0 0	
+		1	C	1A7B	1C	1A7B	1 C	
						1A7C	X X	

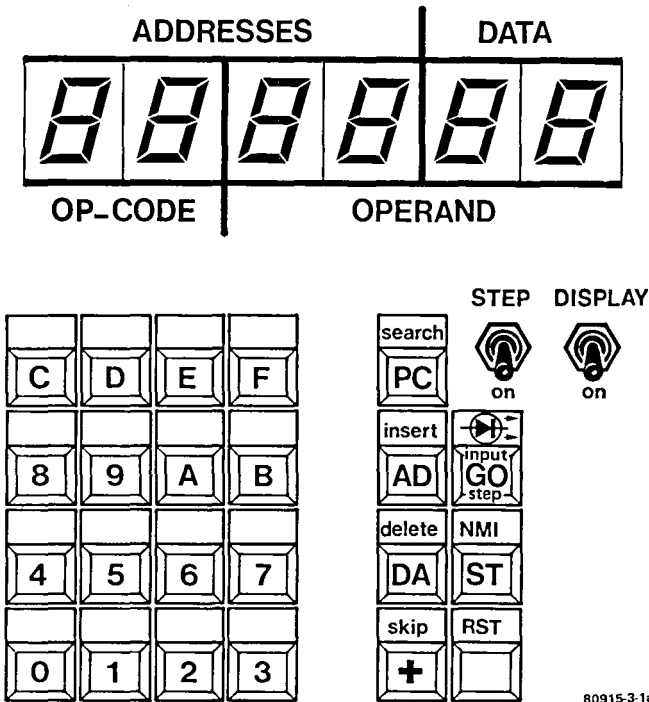
Dat is een hele mondvol. Wat is er allemaal gebeurd? Van alles. Eerst is via het indrukken van RST de monitor aangeropen. De adres- en data-displays geven dan geen kruisjes aan maar willekeurige data (X = dont care, "doet er niet toe" of: "kan van alles zijn"). Door het indrukken van AD geven we te kennen dat we een adres willen opgeven waarop data moet worden geplaatst. 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 ("geincrementeerd", 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 wél nodig als men verderop of verder terug data wil ingeven, bijvoorbeeld op adres 1A7A. 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.



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 RST. Roept het monitorprogramma aan. Activeert displays.
- toets AD. Zorgt voor de adresmode, voor het ingeven van een of meerdere adressen.
- toets DA. Brengt de junior-computer in de datamode.
- toets +. Indrukken verhoogt het laatst ingegeven adres met één. Mode (adres- of data-) blijft ongewijzigd.

N.B. Bij het ingeven van adressen en data verschijnen de hexadecimale "onderdelen" daarvan rechts naar links in het display. Dus bij het ingeven van adres 1A7A: Na 1 : XXX1; na A : XX1A; na 7 : X1A7 en ten slotte na A : 1A7A.

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

Op adres 0200 staat 18, kode voor de instructie CLC, CLear Carry. Vervolgens gaat de computer te rade bij adres 0201. Vindt daar A9, de kode voor LDA, LoaD Accumulator immediate. Laden waarmee? Dat staat op het volgende adres, 0202. De accu wordt dus geladen met 03. Op adres 0203 staat 69. Niet negenezestig maar de hexadecimale kode voor de ADC-instructie, ADd memory to Accumulator with Carry oftewel tel de inhoud van de geheugenplaats met het eerstvolgende adres op de bij de accu-inhoud. De inhoud van adres 0204 is 07, de accu bevatte 03. De nieuwe accu-inhoud zou best wel eens 0A kunnen zijn.

We zijn zeer benieuwd wat er op adres 0205 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, 0206 en 0207; de laatste adresbyte, 00, gevolgd door het eerste, 03. Het opbergadres is dus 0300. Op 0208 staat 4C, kode voor JMP, JuMP, oftewel spring naar een ander adres, dat in de geheugenplaatsen 0209 en 020A staat. Er wordt gesprongen naar adres 0300. Daar was het resultaat van de optelling opgeborgen: 0A is de kode voor ASL-A: Analogic Shift Left Accumulator, schuif de inhoud van de accu één plaats naar links.

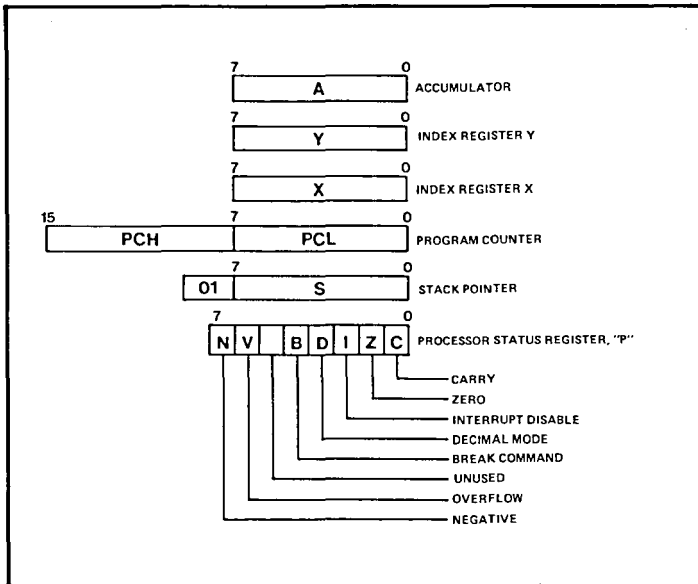
Het "hoe" en "waarom eigenlijk" van dit programma is hier nog niet aan de orde. Waar het om gaat is dat we met allerlei soorten data te maken krijgen, die naast- en doorelkaar worden gebruikt. Bepaalde data vormt de kode van een bepaalde instructie. Een instructie heeft betrekking op bepaalde data. We 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-instructies en operanden zijn gelijksoortig gekodeerd en liggen beide in een geheugen opgeslagen. De bijbehorende kringloop(cyclus) is de volgende: kode voor de (volgende) instructie ophalen uit het geheugen → instructie dekodieren (wat moet er worden gedaan?) → operand halen, indien van toepassing (waar "laat je de instructie op los"? → instructie uitvoeren → adres vaststellen waarop de volgende instructie voor de computer ligt te wachten → deze instructie ophalen → en ga zo maar door. De kreet "programma" zegt nu misschien iets meer: het gaat om een dwingende volgorde van activiteiten.

## Programmeermodel: het "software-dashboard"

Voordat we uitgebreid duiken in de software-mogelijkheden van de junior-computer, zoals de instructieset en de adresseermogelijkheden geven we eerst een overzicht van de interne registerstructuur, de architectuur 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 juistere: 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.

Het blok ALU is de rekeneenheid, Arithmetic (and) Logic Unit. Hier vinden alle rekenkundige en logische operaties (bewerkingen) plaats. De ALU heeft twee acht-bitsingangen en één acht-bitsuitgang. De ALU werkt nauw samen met de accu; in de laatste staan de te manipuleren data en de data als gevolg van het manipuleren.

Dan het (processor-) *statusregister*. Dit register, P, geeft de toestand aan van uitgangdata van de ALU in de vorm van "flags", dat zijn logische toestanden van flipflops. De laatste kunnen geset (1) of gereset (0) zijn, evenzo de bij een flipflop horende vlag.

Eén zo'n vlag is de carry, C. Hij wordt uitgehangen (logisch 1) zodra er, bijvoorbeeld bij een optelling van twee getallen in de ALU 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. Er zijn instructies waarmee het mogelijk is om de toestand van het P-register te onderzoeken. Hierdoor is het voor de gebruiker mogelijk om korrigerend of modifierend op te treden.

De programmateller, PC, is een zestien-bitsregister. Hierin staat het adres van de geheugenplaats waar de volgende instructie 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 instructies 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 instructies. Ze zijn bij bepaalde adresseertechnieken (geïndexeerd en indirect) 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 activiteiten.

## Adresseerrepertoire

Maar liefst dertien verschillende adresseermogelijkheden staan de gebruiker van de junior-computer ter beschikking. Dát is het sterke punt van de 6502- $\mu$ P. De instructieset, het opdrachtenrepertoire is niet zo bijster groot: 56 stuks. De combinatie van een beperkt aantal "krachtige" (slimme) instructies 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 instructies betekent ook: weinig te onthouden voor de gebruiker. Alle instructies zijn gekenmerkt door een (hexadecimale) machinecode en door drie hoofdletters, waarvoor het woord *mnemonics* van toepassing is (afkomstig van de Griekse mythologische figuur Mnemosyne, de personificatie van het geheugen). Een soort instructie-steno dus.

## Immediate-addressing

### *Onmiddellijke adressering*

Immediate adressering heeft betrekking op interne registers. Welk register, dat zit in de mnemonics verpakt. Ze bestaan uit twee bytes; het eerste voor de instructie-omschrijving (de *opcode*), het tweede bevat door de gebruiker opgegeven 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:  $A+M+C \rightarrow A$ . Tel bij de bestaande accu-inhoud de geheugeninhoud op, waarvan het adres onmiddellijk volgt 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-instructie (Clear-Carry). Laten we eens zo'n optelprogramma bekijken:

CLC           clear carry (C = 0)

LDA # 13   laad accu met 13

ADC # 08   tel daar 08 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 instructiekodes op (zie tabel achterin het boek). We vinden: CLC = 18; LDA # = A9; ADC # = 69; BRK = 00 en kiezen als startadres 0100. We gaan daarna

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

				adres:	display:	
RST	AD			xxxx	xx	
0	1	0	0	0100	xx	
DA		1	8	0100	18	CLC
+		A	9	0101	A9	LDA #
+		1	3	0102	13	
+		6	9	0103	69	ADC #
+		0	8	0104	08	
+		0	0	0105	00	BRK
AD				0105	00	
1	A	7	E	1A7E	xx	
DA		0	0	1A7E	00	
+		1	C	1A7F	1C	
AD						
0	1	0	0	0100	18	
GO				0107	xx	programmastart
AD				0107	xx	
0	0	F	3	00F3	21	resultaat

Een paar dingen zullen u vast nog niet duidelijk zijn. Waarom die plotselinge sprong van adres 0105 naar 1A7E? En waarom staat het resultaat op adres 00F3? Na de BRK op 0105 is het eigenlijke programma voltooid. Wat volgt is een aantal huishoudelijke zaken dat te maken heeft met het monitorprogramma. Na afwikkeling van de BRK-instructie worden de adressen 1A7E en 1A7F behandeld. In de bijbehorende geheugenplaatsen staat het startadres (1C00) van een deelprogramma van de monitor, de *save-subroutine*. Die zorgt ervoor dat de inhoud van elk intern  $\mu$ P-register, dus de inhoud van het "software-dashboard" op bepaalde geheugenplaatsen wordt opgeborgen. En wel als volgt:

Op adres 00EF de inhoud van PCL,  
 op adres 00F0 de inhoud van PCH,  
 op adres 00F1 de inhoud van P,  
 op adres 00F2 de inhoud van S,  
 op adres 00F3 de inhoud van A,  
 op adres 00F4 de inhoud van Y en  
 op adres 00F5 de inhoud van X.

Op 00F3 staat de inhoud van A, in ons geval het resultaat, 21, 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 instructie daarvoor is:

SBC#[byte] met als gevolg:  $A - M - \bar{C} \rightarrow A$ . Trek van de bestaande accu-inhoud de geheugeninhoud (M) af, waarvan het adres onmiddellijk volgt.

Er wordt in twee-komplement gewerkt (zie hoofdstuk 2). Ook de inhoud  $\bar{C}$  (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  $\bar{C}=0$ ,  $C=1$  worden gemaakt. Dat doet de instructie SEC. Het aftrekprogramma ziet er als volgt uit:

SEC            SEC → kode 38                    SBC #08    SBC → kode E9  
LDA #13    LDA → kode A9                    BRK        BRK → kode 00

Ook nu kiezen we startadres 0100. We toetsen als volgt:

				adres	data	
RST	AD			xxxx	xx	
0	1	0	0	0100	xx	
DA		3	8	0100	38	SEC
+		A	9	0101	A9	LDA #
+		1	3	0102	13	
+		E	9	0103	E9	SBC #
+		0	8	0104	08	
+		0	0	0105	00	BRK
AD				0105	00	
1	A	7	E	1A7E	xx	} zie opm.
DA		0	0	1A7E	00	
+		1	C	1A7F	1C	
AD						
0	1	0	0	0100	38	programmastart
GO				0107	xx	
AD				0107	xx	
0	0	F	3	00F3	0B	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 00F3 het resultaat van het programma: 0B, afgetrokken van 13 (decimaal:19) levert 0B (decimaal: 11).

### Logische instructies

We behandelen in het kader van immediate adresseren nog een aantal instructies die te maken hebben met de L van de inmiddels bekende ALU. Logische instructies dus.

Het OR-ren is een bekende logische operatie. De instructie ORA # [byte] heeft tot gevolg:  $AYM \rightarrow A$ . De kode is 09. Laten we aan de hand van een programmaatje eens bekijken wat er precies gebeurt:

LDA #AA laad de accu met AA

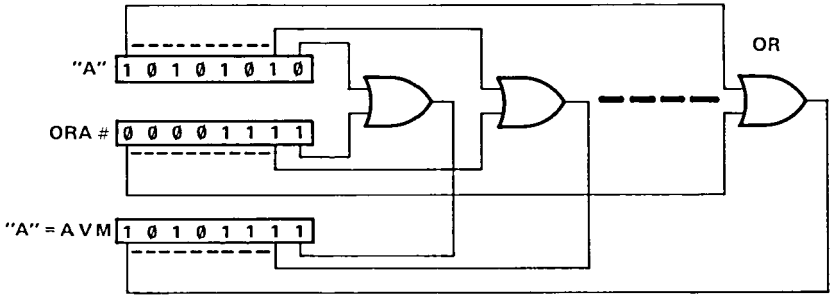
ORA #0F voer bit voor bit de OR-functie uit met 0F

BRK stop

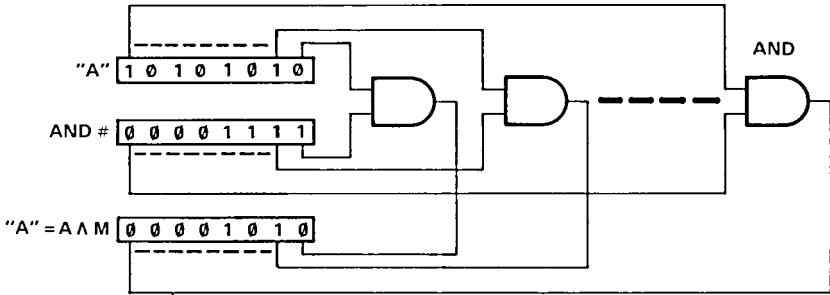
AA is binair: 10101010

0F is binair: 00001111

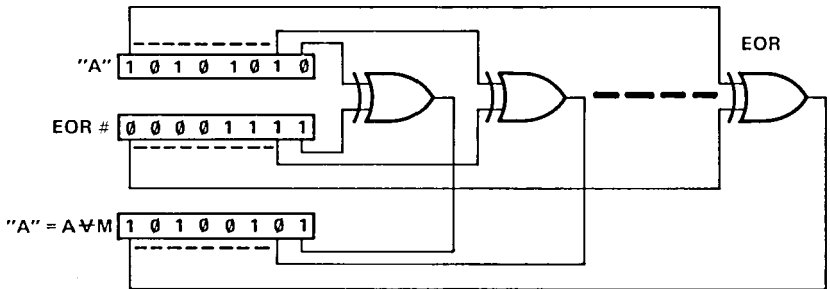
resultaat: 10101111 (AF)



80915-3-2a



80915-3-2b



80915-3-2c

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

Op bits met gelijke positie wordt de OR-functie losgelaten. Het resultaat komt in de accu te staan. Figuur 2a geeft met hardware (OR-poorten) weer wat er gebeurt.

Een andere logische operatie: AND. De instructie AND# heeft als gevolg  $A \wedge M \rightarrow A$ . De code is 29. Ook hier eerst een programma:

LDA#AA laad accu met AA

AND#0F voer bit voor bit de AND-functie uit met 0F

BRK stop

AA is binair: 10101010  
 0F is binair: 00001111  
 resultaat: 00001010 (0A)

Op bits met gelijke positie wordt de AND-functie 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-)functie. De instructie EOR # heeft als gevolg:  $A \vee M \rightarrow A$ . De code is 49. Een programma:

LDA #AA laad accu met AA  
 EOR #0F voer bit voor bit de EOR-functie uit met 0F  
 BRK stop

AA is binair: 10101010  
 0F is binair: 00001111  
 resultaat: 10100101 (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:

RST	AD			xxxx	xx		
0	1	0	0	0100	xx		
DA		A	9	0100	A9	LDA #	begin 1
+		A	A	0101	AA		
+		0	9	0102	09	ORA #	
+		0	F	0103	0F		
+		0	0	0104	00	BRK	einde 1
+		A	9	0105	A9	LDA #	begin 2
+		A	A	0106	AA		
+		2	9	0107	29	AND #	
+		0	F	0108	0F		
+		0	0	0109	00	BRK	einde 2
+		A	9	010A	A9	LDA #	begin 3
+		A	A	010B	AA		
+		4	9	010C	49	EOR #	
+		0	F	010D	0F		
+		0	0	010E	00	BRK	einde 3
AD							
0	1	0	0	0100	A9	startadres 1	
GO				0106	AA	stop 1	
AD							
0	0	F	3	00F3	AF	resultaat 1	
AD							
0	1	0	5	0105	A9	startadres 2	
GO				010B	AA	stop 2	

AD	0	0	F	3	00F3	0A	resultaat 2
AD	0	1	0	A	010A	A9	startadres 3
GO					0110	xx	stop 3
AD	0	0	F	3	00F3	A5	resultaat 3

Door het inbouwen van drie onderbrekingen (BRK) kunnen we de drie programmadelen achtereenvolgens uitvoeren. Er wel op letten dat telkens vóór GO het juiste startadres wordt ingegeven!

De drie behandelde logische instructies 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-instructies. Acht stuks hebben we er leren kennen: LDA#, LDX#, LDY#, ADC#, SBC#, ORA#, AND# en EOR#. Daarnaast de instructies CLC, SEC en BRK, die bij het implied adresseren nog uitgebreid zullen worden behandeld.*

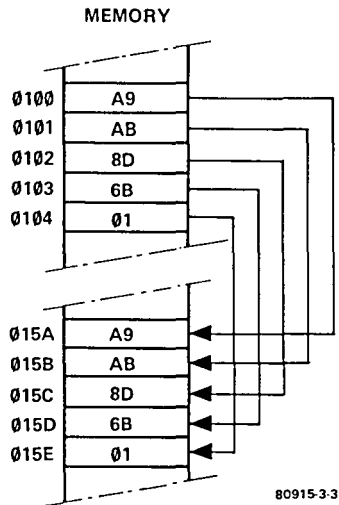
### Absolute adressering

Bij onmiddellijke of immediate adressering werd (operand-) data die direct volgde op de opcode van de instructie direct in een intern register van de  $\mu P$  gezet. Dus in A, X of Y. Dat hoeft niet zo. Het kan ook via het geheugen van de microcomputer. Direct na de opcode volgt dan in twee stappen (bytes) het adres van de geheugenlokatie die aan bod is. Laten we gewoon eens een programmaatje bekijken:

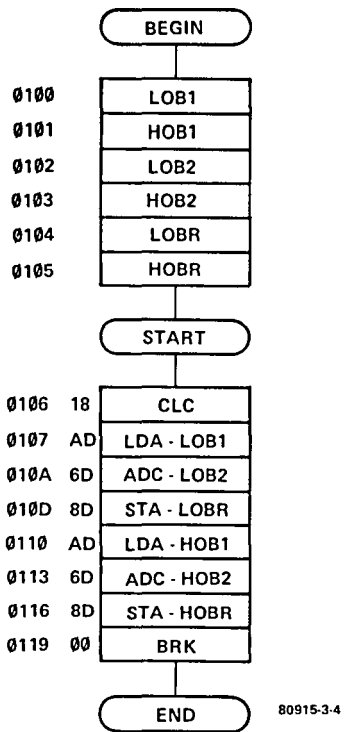
LDA-0100 laad accu met de inhoud van plaats 0100  
 STA-015A berg accu-inhoud op op plaats 015A  
 LDA-0101 laad accu met de inhoud van plaats 0101  
 STA-015B berg accu-inhoud op op plaats 015B  
 LDA-0102 laad accu met de inhoud van plaats 0102  
 STA-015C berg accu-inhoud op op plaats 015C  
 LDA-0103 laad accu met de inhoud van plaats 0103  
 STA-015D berg accu-inhoud op op plaats 015D  
 LDA-0104 laad accu met de inhoud van plaats 0104  
 STA-015E berg accu-inhoud op op plaats 015E

N.B. Het streepje tussen de mnemonics en het adres geeft het absolute adresseringskarakter aan.

Wat is er gebeurd? De inhoud van de plaatsen 0100 . . . 0104 is gekopieerd op de plaatsen 015A . . . 015E. 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.



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 instructie LDA is een oude bekende, zij het nu zonder #. De instructie 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 instructies drie bytes. Het eerste byte is de kode voor de instructie, 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 instructies 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 (01 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 0100 zijn gereserveerd voor de zo juist genoemde bytes, als grondstof of eindprodukt van het programma. Het eigenlijke programma speelt zich af van 0106 t/m 0119. 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 04EF, getal 2 23AB. Daar gaat ie:

(STEP: OFF; DISPLAY: ON)

RST	AD			xxxx	XX	
1	A	7	A	1A7A	XX	
DA		0	0	1A7A	00	} STEP-voorbereiding
+		1	C	1A7B	IC	
++				1A7D	XX	
+				1A7E	00	} BRK-voorbereiding
+				1A7F	IC	
AD				1A7F	IC	
0	1	0	0	0100	XX	
DA		E	F	0100	EF	LOB1
+		0	4	0101	04	HOB1
+		A	B	0102	AB	LOB2
+		2	3	0103	23	HOB2
+				0104	XX	plaats voor LOBR
+				0105	XX	plaats voor HOBR
+		1	8	CLC	0106	18 clear carry-vlag
+		A	D	LDA-	0107	AD
+		0	0		0108	00
+		0	1		0109	01
+		6	D	ADC-	010A	6D
+		0	2		010B	02
+		0	1		010C	01

+	8	D	STA-	010D	8D	} resultaat (=LOBR) naar adres 0104
+	0	4		010E	04	
+	0	1		010F	01	
+	A	D	LDA-	0110	AD	} HOB1 in A
+	0	1		0111	01	
+	0	1		0112	01	
+	6	D	ADC-	0113	6D	} A+HOB2→A (carry-vlag telt mee)
+	0	3		0114	03	
+	0	1		0115	01	
+	8	D	STA-	0116	8D	} resultaat (=HOBR) naar adres 0105
+	0	5		0117	05	
+	0	1		0118	01	
+	0	0	BRK	0119	00	eind programma
AD						
0	1	0	6	0106	18	startadres
GO				011B	xx	programma loopt
AD						
0	1	0	4	0104	9A	resultaat: LOBR
+				0105	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: 04EF. Binair: 00000100 11101111

Getal 2: 23AB. Binair: 00100011 10101011

←HOB→ ←LOB→

Eerst wordt op adres 0100 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
+	<hr/>	
	110011010	LOBR
carry	←9→←A→	

Bij het opbergen van LOBR (op 0104) en het binnenhalen van HOB1 in de accu blijft de carry-vlag zoals hij was. We gaan weer optellen:

	00000100	HOB1
	00100011	HOB2
	1	carry van LOBR (nu helemaal rechts!)
	111	carry
+	<hr/>	
	000101000	HOBR
carry	2 8	

Het resultaat zien we op 0104 en 0105. Door het aanroepen van

de adressen 0104 en 0105 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 verwijzadres (1C00). Meer hierover later.

We hadden het over absoluut adresseren, weet u nog wel? Het optelprogramma van daarnet gebruikte bijna uitsluitend instructies met absolute adressering. Heel mooi heeft u de bijbehorende afwikkeling van een instructie in drie fasen kunnen zien: opcode — rechter adresbyte — linker adresbyte.

Er zijn nog veel meer instructies 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 instructies:

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 instructies wordt de geheugeninhoud van M met 1 (00000001) verhoogd (INC-) of verlaagd (DEC-).

logische instructies:

ORA- kode 0D (A ∨ M → A) OR memory with accumulator  
AND- kode 2D (A ∧ 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 #)

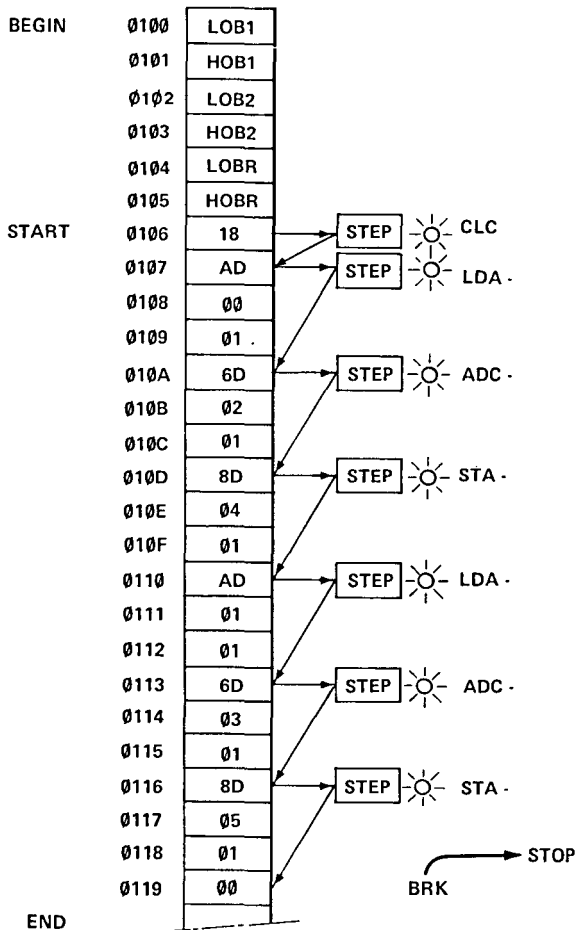
*Voordat we aan weer een nieuwe adresseermogelijkheid beginnen eerst een intermezzo: een nieuwe programmeermogelijkheid.*

## 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-instructie 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 00 en 1A7B met 1C — een puur huishoudelijke kwestie. Stel we hebben net zo als eerder het programma ingetoetst, inclusief 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



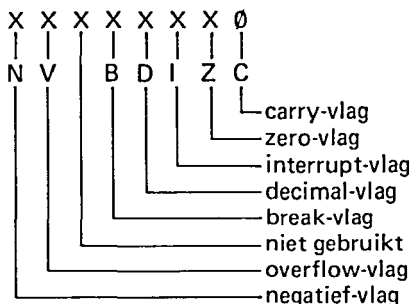
80915-3-5

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

STEP wordt één instructie afgewerkt. Figuur 5 illustreert dit. We zien een stukje geheugen met het optelprogramma uit "absoluut adresseren".

Goed, we zijn op 0106, 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 00F1. Wat ligt er nou meer voor de hand als daar eens te gaan kijken? Dus toets AD 00F1 in. Wat zien we op de displays? De adresdisplays wijzen 00F1 aan. En de data-displays?

Even geduld a.u.b. Eerst iets anders. Het byte in het P-register en dus ook op adres 00F3 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 gedecodeerd 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-instructie is uitgevoerd.

Er is via AD etc. een uitstapje gemaakt en we willen weer terug naar het adres waar de volgende instructie (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 instructie, althans de instructie- of opcode ligt opgeslagen.

Na het indrukken van PC verschijnt er 0107 AD in het display; AD is de opcode van LDA-. Drukken we STEP. We zien nu: 010A 6D. Inmiddels is dan de voorgaande LDA-instructie uitgevoerd, dus LOB1=EF moet in de accu staan. Even checken: de inhoud van A is gered op adres 00F3. Toets: AD, 0, 0, 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).

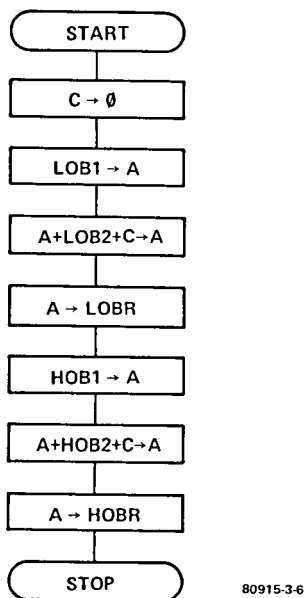
### *Huishoudelijk intermezzo*

In het vervolg zullen alle (voorbeeld-)programma's worden gegeven in drie vormen: 1. De "grove" flowchart (zie bijvoorbeeld figuur 6, van het optelprogramma). 2. De uitgewerkte flowchart, met opcodes (zie bijvoorbeeld figuur 4, van het optelprogramma), en 3. Het toetsprogramma, met displays.

## Zero page addressing

### *Adresseren op pagina nul*

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



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

hiervan doordat het byte ADH altijd nul is (00). Het adresbereik doorloopt dus de mogelijkheden: 0000 . . . 00FF. Deze 256 verschillende adressen horen bij wat we noemen pagina nul. De volgende 256 adressen horen bij pagina 1: 0100 . . . 01FF. Dat gaat zo door tot pagina FF, met de adressen FF00 . . . 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 0 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-instructie het linker byte 00 is hoeven we dat niet meer op te geven. Met als gevolg twee bytes in plaats van drie bytes per instructie. En dat spaart geheugenplaatsen!

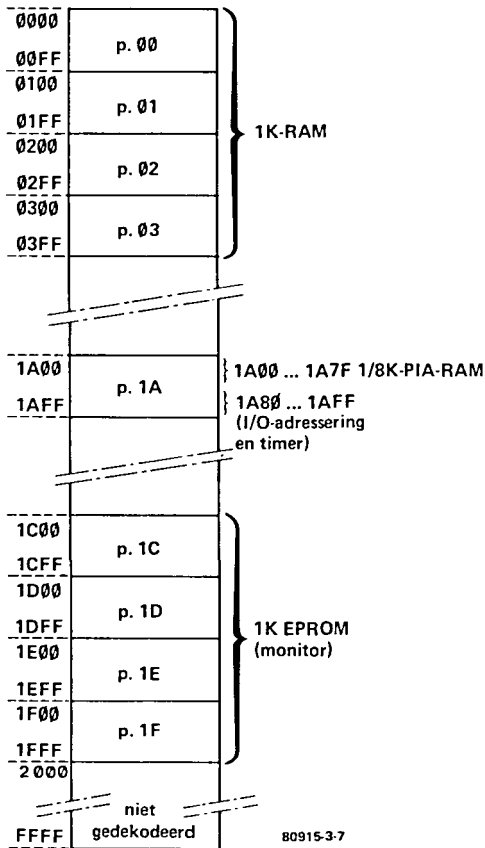
Pagina-nul-instructies bestaan dus uit twee bytes: de opcode, gevolgd door het rechter adresbyte ADL. De mnemonics van de opcode eindigt op Z, van Zero, nul. Het instructie-overzicht ziet er bekend uit:

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 0 tot en met F.

STAZ kode 85 ( $A \rightarrow M$ ) store accumulator in memory

STXZ kode 86 ( $X \rightarrow M$ ) store index X in memory

STYZ kode 84 ( $Y \rightarrow M$ ) store index Y in memory

rekenkundige (arithmetic) instructies:

ADCZ kode 65 ( $A+M+C \rightarrow A$ ) add memory to accumulator with carry

SBCZ kode E5 ( $A-M-C \rightarrow A$ ) subtract memory from accu w. carry

INCZ kode E6 ( $M+1 \rightarrow M$ ) increment memory by one

DECZ kode C6 ( $M-1 \rightarrow M$ ) decrement memory by one

logische instructies:

ORAZ kode 05 ( $A \vee M \rightarrow A$ ) OR memory with accumulator

ANDZ kode 25 ( $A \wedge M \rightarrow A$ ) AND memory with accumulator

EORZ kode 45 ( $A \vee M \rightarrow A$ ) Exclusive OR memory with accumulator

## Relative addressing

### *relatieve adressering*

Deze adresseringsmethode wordt toegepast bij vertakkings- of *sprong-instructies*. De Engelsen en Amerikanen spreken van branch-instructies. Twee soorten zijn er: voorwaardelijke (conditional) en onvoorwaardelijke (unconditional). Bij onvoorwaardelijke instructies 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 spronginstructies zijn er:

1. BCC opcode 90 Branch on Carry Clear. Spring als C=0 (gereset)  
BCS opcode B0 Branch on Carry Set. Spring als C=1 (geset)
2. BNE opcode D0 Branch Not Equal. Spring als Z=0 (gereset)  
BEQ opcode F0 Branch on EQual. Spring als Z=1 (geset)
3. BPL opcode 10 Branch on Plus. Spring als N=0 (gereset)  
BMI opcode 30 Branch on MInus. Spring als N=1 (geset)
4. BVC opcode 50 Branch on oVerflow Clear. Spring als V=0 (gereset)  
BVS opcode 70 Branch on oVerflow Set. Spring als V=1 (geset)

Met deze instructies zullen we programma's gaan maken. De onvoorwaardelijke instructies komen later in dit hoofdstuk aan bod.

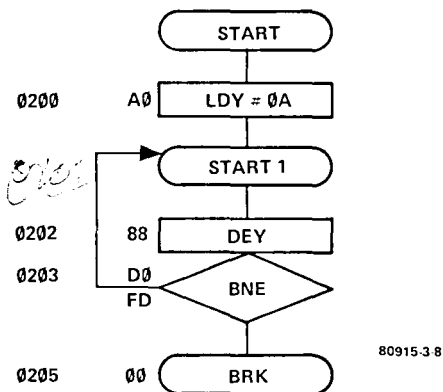
Om te beginnen het programma waarvan het stroomdiagram in figuur 8 staat. Op het startadres, 0200 wordt het Y-register geladen met 0A en via de DEY-instructie wordt de inhoud van Y met 1 verlaagd. Dus 0A wordt 09. Dan komt BNE. Afhankelijk van de toestand van de nulvlag Z wordt teruggegaan naar DEY. Het Y-register bevat bij de eerste BNE 09, 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 zolang met 1 te verlagen dat 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(D0) 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 1111101 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:

0200	A0	LDY #
0201	0A	
0202	88	DEY
0203	D0	BNE
0204	FD	aantal stappen
0205	00	BRK

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



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

Direkt na het afwerken van adres 0204 staat de programmateller PC al op het volgende adres 0205 gericht. En het is de programmateller, waarop de stappen van 0204 betrekking hebben. Het na de opcode van een voorwaardelijke instructie 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 00...7F) tot maximaal 128 stappen terug (-128; hexadecimaal FF...80). 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 instructie die volgt op de voorwaardelijke spronginstructie. 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 spronginstructie. 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 spronginstructies staat (03) en meteen daarna het rechter byte van het doeladres (02). De twee adresbytes verschijnen in het adresdisplay en het datadisplay laat de berekende offset (FD) zien:

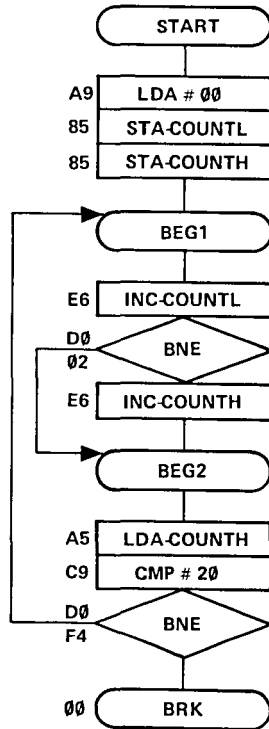
AD				XXXX XX
1	F	D	5	1FD5 D8
GO				0000 00
0	3	0	2	0302 (FD) - offset noteren!
RST				

Men kan tussen twee berekeningen in het display 000000 maken door een willekeurige kommandoets in te drukken.

Na het indrukken van RST heeft de computer dit rekenprogramma verlaten. Nu kunnen we het programma van figuur 8 intoetsen:

AD						
0	2	0	0	0200	XX	
DA		A	0	0200	A0	LDY #
+		0	A	0201	0A	
+		8	8	0202	88	DEY
+		D	0	0203	D0	BNE
+		F	D	0204	FD	offset
+		0	0	0205	00	BRK

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.



COUNTL = 0000  
COUNTH = 0001

80915-3-9

Figuur 9. Stroomdiagram van een teller-programma ("software-teller").

### Een software-teller

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

Het gaat om een teller die hexadecimaal van 0000 tot 2000 telt en als de maat vol is stopt. Het stroomdiagram staat in figuur 9. Het getal 2000 beslaat 16 bits; twee geheugenplaatsen zijn er nodig om de tellerstand vast te leggen. Op plaats COUNTH (adres 0001) staat het linker byte, op plaats COUNTL (adres 0000) het rechter byte.

Begonnen wordt met het laden van 00 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 spronginstructie, 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 instructie CMP vergeleken met de eindwaarde, 20. In het kort alvast iets over deze instructie, later meer. De instructie (opcode C9) beïnvloedt ondermeer de nulvlag. Deze wordt 1 als de inhoud van A (dus van COUNTH) gelijk is aan 20 en blijft nul zolang dat niet het geval is. Is de inhoud van COUNTH gelijk aan 20, 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 00. Dat heeft ophogen van COUNTH tot gevolg (bij de bovenste BNE wordt dan niet gesprongen) en wel zolang COUNTH lager dan 20 is. Het toetsen van het programma gaat zo:

AD				xxxx	xx	
1	F	D	5	1FD5	D8	startadres offset-berekening
GO				0000	00	
1	8	1	C	181C	02	offset eerste BNE
2	0	1	6	2016	F4	offset tweede BNE
RST	AD					
0	2	1	0	0210	xx	startadres programma
DA		A	9	0210	A9	LDA #
+		0	0	0211	00	
+		8	5	0212	85	STA-
+		0	0	0213	00	
+		8	5	0214	85	STA-
+		0	1	0215	01	
+		E	6	0216	E6	INC-
+		0	0	<sup>12</sup> 0217	00	
+		D	0	<sup>11</sup> 0218	D0	BNE
+		0	2	<sup>10</sup> 0219	02	offset
+		E	6	<sup>9</sup> 021A	E6	INC-
+		0	1	<sup>8</sup> 021B	<sup>1</sup> 01	

+	A	5	7	021C	<sup>2</sup>	A5	LDA-
+	0	1	6	021D		01	
+	C	9	5	021E		C9	CMP #
+	2	0	4	021F		20	operand CMP #
+	D	0	3	0220		D0	BNE
+	F	4	2	0221		F4	offset
+	0	0	1	0222		00	BRK
AD				0222		xx	
0	2	1	0	0210		A9	startadres
GO				0212		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 02.

Voelen we nu de CMP-instructie nader aan de tand. Hij zorgt voor het vergelijken van de accu-inhoud met data. In de immediate-versie (CMP#; opcode C9) is het eerstvolgend byte de data waarmee wordt vergeleken, de referentiedata. Er zijn nog meer adresseermogelijkheden voor CMP.

Er wordt A-M gedaan: van de accu-inhoud wordt de referentiedata afgetrokken zonder rekening te houden met de carry-vlag. De accu-inhoud blijft na afloop ongewijzigd. Twee mogelijkheden zijn er voor A-M: nul of niet; A is gelijk aan M of ongelijk.

Drie vlaggen worden beïnvloed door het resultaat:

a) de N-vlag wordt: geset (1) als  $A < M$   
(geheugensteuntje: voor negatieve getallen is het linker bit 1, voor positieve 0)

b) de nulvlag Z wordt: geset (1) als  $A = M$

c) de carryvlag C wordt: geset (1) als  $A \geq M$

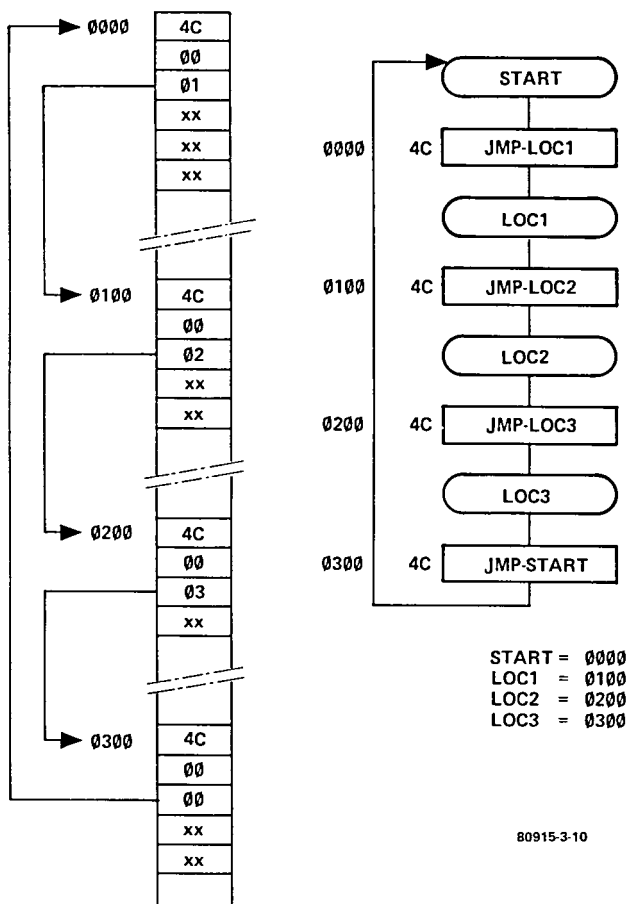
Halen we het voorwaardelijke instruktierepertoire voor de geest, dan weten we nu drie dingen:

- Met een CMP-instructie kan worden vastgesteld of de accu-inhoud kleiner is dan bepaalde data of niet; met een aansluitende BMI-instructie kan worden gekeken of er afhankelijk daarvan wordt gesprongen of niet.
- Na een CMP weet men of de accu-inhoud gelijk is aan bepaalde data of niet; met een aansluitende BEQ-instructie kan worden gekeken of er afhankelijk daarvan wordt gesprongen of niet.
- Na een CMP-instructie weet men of  $A \geq$  bepaalde data is of niet; met een aansluitende BCS-instructie kan worden gekeken of er afhankelijk daarvan wordt gesprongen of niet.

*Tot zover voorlopig de voorwaardelijke spronginstructies.*

### *Onvoorwaardelijke spronginstructies*

Doorliepen de programma's tot nu toe geheugenplaatsen met netjes op elkaar aansluitende adressen (met uitzondering van de voorwaardelijke spronginstructies, 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 spronginstructies.

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 instructie in huis:

**JMP-.** Het is een onvoorwaardelijke spronginstructie (JuMP=spring) met absolute (let op het liggend streepje achter de mnemonics) of (nog te bespreken) indirecte adressering. Voor absolute adressering is de opcode 4C, voor indirecte 6C. Een 3-byte-instructie dus, waarvan de laatste twee het sprongadres bevatten. Figuur 10 geeft stroomdiagram en geheugenkaartjes van een programmaatje met uitsluitend JMP-instructies. Vanuit START wordt in een bepaalde volgorde achtereenvolgens en periodiek vier adressen doorlopen: "vlooiën springen". Flauw eigenlijk. Maar nu het serieuze werk.

## JSR en RST – 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 dát 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 instructies plus operanden (de subroutine, het deelprogramma) eenmalig in een bepaald stuk geheugen te laden en als de handeling is vereist via een onvoorwaardelijke spronginstructie JSR te springen naar het startadres van de subroutine. Om na het afwikkelen ervan via een onvoorwaardelijke spronginstructie 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 structuur overzichtelijker wordt.

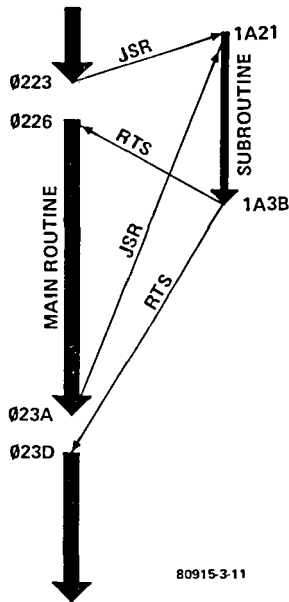
Een programmavoorbeeld. Enwel dat van figuur 11. We zien een subroutine op de geheugenplaatsen 1A21 . . . 1A3B, die twee keer wordt aangeroepen. Op 0223 staat de opcode van een JSR, op 0224 (ADL) en 0225 (ADH) het startadres van de subroutine. De computer is op de hoogte van het adres (0225) dat als laatste voor de sprong is afgehandeld; dat adres staat in wat heet de stack of stapel, waarover we het nog zullen hebben. Op 023A vindt de tweede JSR plaats; op de adressen 023B en 023C staat dezelfde adresdata als op 0224 en 0225. Een RTS leidt ertoe dat de draad van het hoofdprogramma weer wordt opgenomen op adres 023D. 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



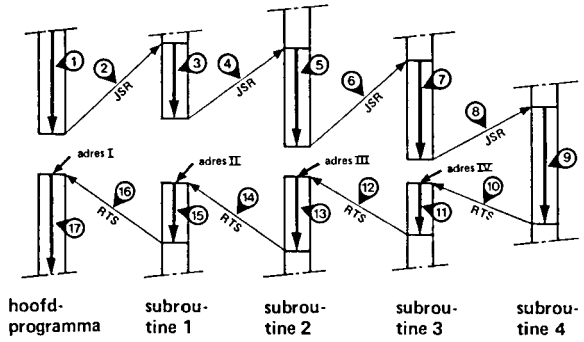
Figuur 11. Hoofdprogramma ↔ subroutine. Vice versa, dus een retour en nooit een enkele reis!

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.

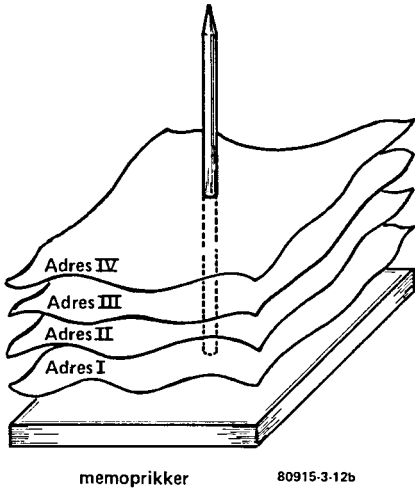
Het aantal verpleegsters met cacaobusjes is beperkt (ze worden steeds kleiner), het aantal kubussen is beperkt (denk aan de afmetingen van het kind), en ook het aantal keren dat men een JSR kan loslaten, de *nestdiepte* is beperkt. Iedere JSR moet een keer worden gevolgd door een RTS; is het aantal JSR's 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 welke 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 spronginstructie. Dat bewuste adres is als laatste op de stapel gekomen, ligt dus "boven op" de stapel.

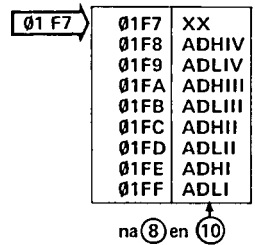
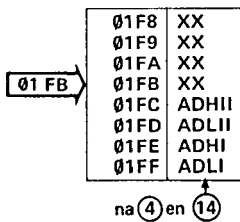
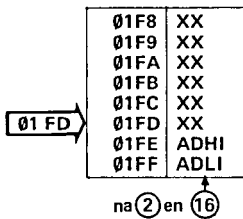
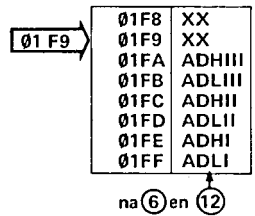
Dus wat het laatste binnenkwam wordt het eerst afgewerkt. Vergelijk het met de afwas. eerst worden borden afgespoeld, dan volgt het eigenlijke afwassen. Na het afspoelen staan de borden op een stapel. Het bovenste bord is als laatste afgespoeld en wordt als eerste afgewassen (tenminste: bij



80915-3-12a



80915-3-12b



80915-3-12c

**Figuur 12.** Het nesten van subroutines (a). De toestand van de stapel (stack) doet daarbij sterk denken aan een memoprikket (b). De toestand van de stapel en van de stapelwijzer (stack pointer) op acht tijdstippen, die ook in a zijn aangegeven (c).

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 sprongen. 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.

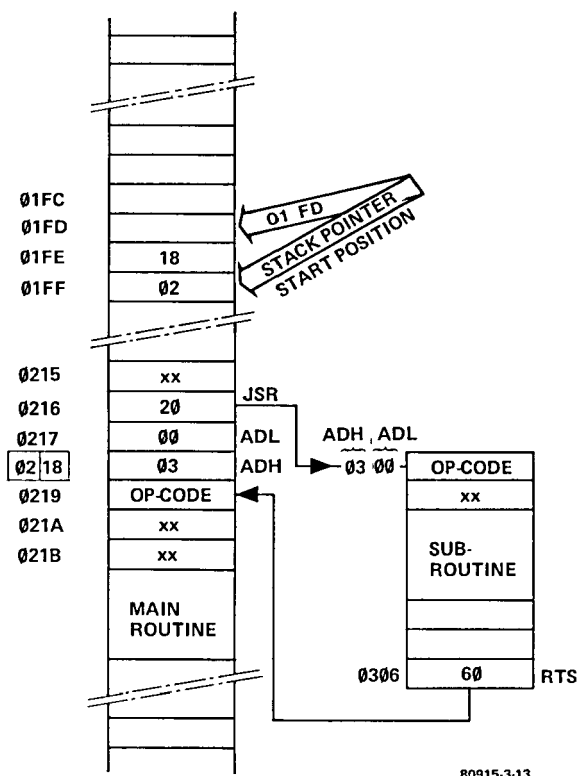
Als stapel kunnen de 256 geheugenplaatsen van pagina 1 worden gebruikt, met adressen 01FF . . . 0100. Er is dus plaats voor maximaal 128 terugkeeradressen. Een geheugenkaart loopt zoals gebruikelijk van boven naar beneden met toenemend adres. De stapelplaatsen worden van onder naar boven met terugkeeradresinformatie geladen (vergelijk figuur 12b!), te beginnen met de eerste adressen 01FF en 01FE, 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 rechter byte ADL en vervolgens het linker byte ADH 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 wél te zien in het programmavoorbeeld van figuur 13.

Dit programma heeft een voorgeschiedenis die loopt tot en met adres 0215. Dan, op 0216 gebeurt het: daar staat 20, de opcode van JSR. Springen waarheen? Op de adressen 0217 en 0218 staat waarheen gesprongen wordt; op 0217 het rechter byte (00) en op 0218 het linker byte (03) van het startadres van de subroutine. Op de stapel in pagina 1 van het geheugen staat op 01FF 02 (rechter byte) en op 01FE 18 (linker 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 0300 met de opcode van de eerste instructie. Daarop volgen er meer; welke precies is hier niet interessant. In ieder geval komt op 0306 60, opcode van de instructie RTS, een een-byte-instructie 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 0219 gaat de opcode van een nieuwe instructie 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 spronginstructies aan bod komen. We leren meteen een paar kunstjes kennen uit het rijke repertoire van het monitorprogramma.*



80915-3-13

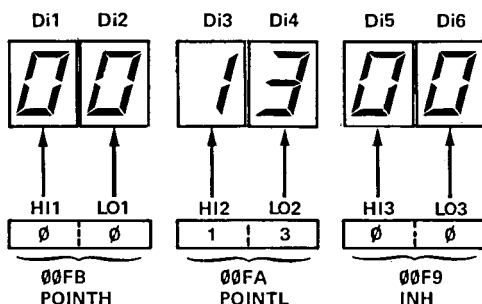
Figuur 13. Programmavoorbeeld met een subroutine.

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.



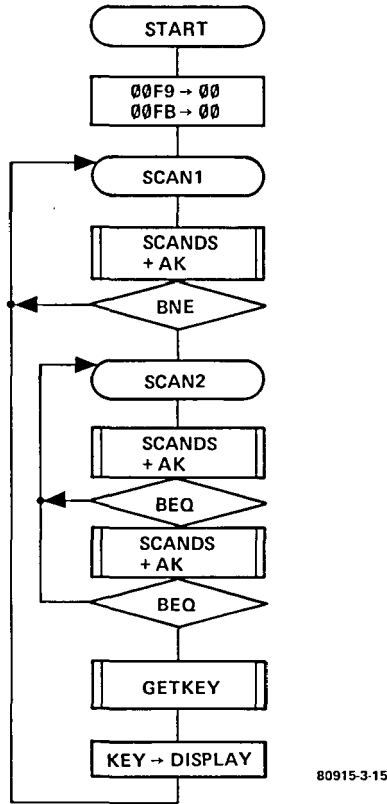
80915-3-14

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

De zevensegment-displays moeten de getallen 0 . . . 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 00FB in zevensegmentcode omgezet en aan Di1 toegevoerd, die gedurende een zekere tijd tot oplichterij wordt aangespoord. Dan worden de rechter vier bits van 00FB na omzetting in zeven segmenten op Di2 gezet en Di2 enige tijd ingeschakeld. Op dezelfde manier komt de inhoud van 00FA en 00F9 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-instructie al dan niet worden besloten tot een sprong.

Zodra een toets is ingedrukt moet de hexadecimal toetswaarde worden omgezet in de passende zevensegmentcode. 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 continu 00 aangeven. Daarom worden aan het begin van het programma de geheugenplaatsen 00FB en 00F9 met 00 geladen. Eerst wordt programmadeel SCAN1 afgehandeld. Dit bevat een SCANDS + AK-subroutine, waarmee de inhoud van geheugenplaatsen 00F9, 00FA en 00FB wordt gedisplaysed en waarmee gekeken wordt of een toets is ingedrukt. Via de voorwaardelijke instructie 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 dendereffect van de ingedrukte toets tot het verleden behoort. Zou men deze 'software-debouncing' (geprogrammeerde dender-onderdrukking) niet hebben

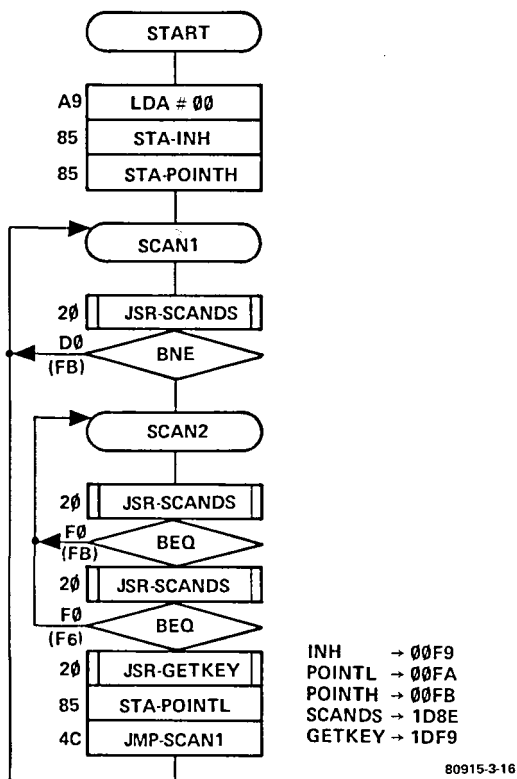


**Figuur 15.** Stroomdiagram van de monitor-routine die zorgt voor toetsdetectie, toetsherkenning en de weergave op het display van de toetswaarde.

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-instructies 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				0000	00	
0	9	0	6	0906	<u>FB</u>	→ offset noteren!
0	E	0	B	0E0B	<u>FB</u>	→ offset noteren!
1	3	0	6	1306	<u>F6</u>	→ offset noteren!
RST						
AD						
0	2	0	0	0200	XX	
DA		A	9	0200	A9	LDA #
+		0	0	0201	00	
+		8	5	0202	85	STAZ
+		F	9	0203	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.

## Implied addressing

### *Ingebouwde adressering*

Vijftientig instructies hebben betrekking op "interne aangelegenheden" van de microprocessor. Bijvoorbeeld interne dataverhuizing van het ene naar het andere register. Of instructies die de voortgang van het programma beïnvloeden. Er zijn geen een of twee bytes nodig voor het vastleggen van de operand; die zit in de opcode van de instructie ingebouwd. Het gaat hier dan ook om een-byte-instructies. Een paar kennen we al. Hier volgen ze allemaal:

BRK	kode 00	BReaK
CLC	kode 18	Clear Carry
CLD	kode D8	Clear Decimal mode
CLI	kode 58	Clear Interrupt flag
CLV	kode B8	Clear oVerflow flag
DEX	kode CA	DEcrement X-register by one
DEY	kode 88	DEcrement Y-register by one
INX	kode E8	INcrement X-register by one
INY	kode C8	INcrement Y-register by one
NOP	kode EA	No OPeration (een loze instructie; geheugensteuntje: NOPpes)
PHA	kode 48	Push Accumulator on stack
PHP	kode 08	Push Processor-status on stack
PLA	kode 68	Pull Accumulator from stack
PLP	kode 28	Pull Processor-status from stack
RTI	kode 40	ReTurn from Interrupt
RTS	kode 60	ReTurn from Subroutine
SEC	kode 38	SEt Carry flag
SED	kode F8	SEt Decimal Flag
SEI	kode 78	SEt Interrupt flag
TAX	kode AA	Transfer Accumulator to X-register
TAY	kode A8	Transfer Accumulator to Y-register
TSX	kode BA	Transfer Stackpointer to X-register
TXA	kode 8A	Transfer X-register to Accumulator
TXS	kode 9A	Transfer X-register to Stack pointer
TYA	kode 98	Transfer Y-register to Accumulator

Nu wat meer bijzonderheden van een aantal ervan.

### *SED en CLD*

Met de instructies 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-instructie. 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=0).

### *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 instructie 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 instructie op de plaatsen met een, twee of drie opeenvolgende NOP's. Vallen er bij het debuggen instructies weg dan kan men de geheugenplaatsen vullen met (de opcode van) een NOP. Dus toch niet helemaal een loze instructie.

### *Push-Pull-Transfer*

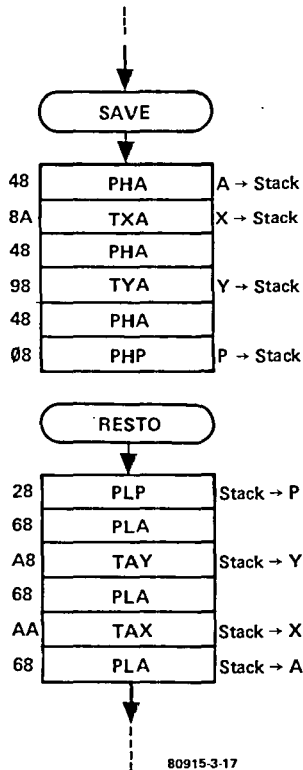
Alle implieed instructies waarvan de mnemonics ("instructie-steno") beginnen met P of T betreffen intern datatransport van of naar A, X, Y, S of P, het processor-statusregister. Er zijn heel wat toepassingen van dit intern data-stuivertje-wisselen.

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 (in S) 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

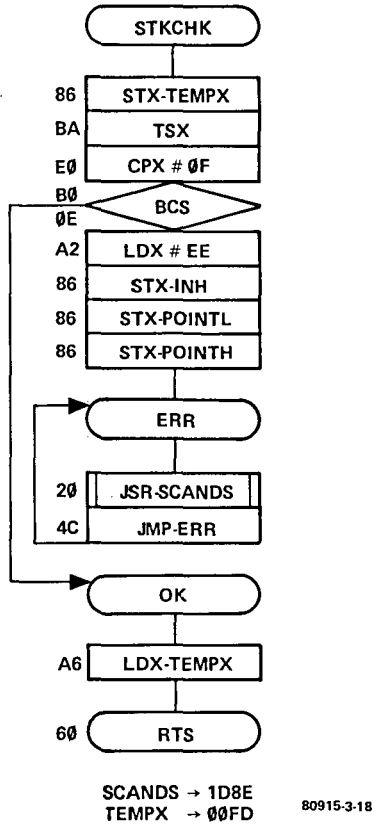
Het had ook anders gekund:

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-instructies respectievelijk pull-instructies.

De instructies 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 vóór 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 vóór de sprong en een herstelprogramma RESTO direct 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.** Stroomdiagram van het programma STKCHK, waarmee het overschrijden van het maximale aantal stapelplaatsen wordt vastgesteld met een foutmelding.

De stapelwijzer (het interne register S) staat gericht op het adres van de eerstvolgende lege plaats van de stapel. Na het aanroepen van de monitor van de junior-computer (het indrukken van RST) staat de stapelwijzer automatisch gericht op adres 01FF op pagina 1 van het RAM-geheugen. Het is mogelijk om daarvan af te wijken, dus om de wijzer te richten op een ander adres in pagina 1. Dat gaat zo:

LDX# 81 stop 81 in het X-register  
 TXS breng X naar de stack pointer S

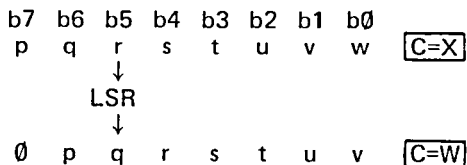
De stack pointer staat nu gericht op adres 0181 van pagina 1 van het RAM-geheugen. Adres 01FF is de laagste stapelplaats en als je ergens anders begint is de kans groter dat het maximale aantal stapelplaatsen wordt overschreden. Dit kan worden signaleerd door opname in het programma van de routine STCHK volgens figuur 18. Zodra de stapelwijzer boven het hoogste adres (0100) van pagina 1 uitwijst ontstaat een foutmelding (ERROR) in de vorm van EEEEEEE op het display. Pas na het

indrukken van de toets RST keert de junior-computer terug naar het monitorprogramma. Het programma van figuur 18 is vooral handig bij grote programma's. We komen er nog op terug. Evenals op verscheidene implied instructies, die hier nog niet zijn besproken.

### *Accu-adressering*

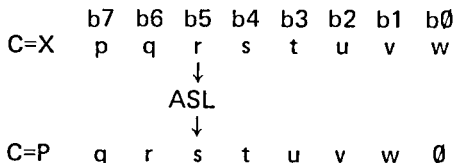
Het gaat om een variant op de ingebouwde adressering. Om vier een-byte-instructies die de accu-inhoud manipuleren. Om ASL, LSR, ROL en ROR. Schuiven (shift) en draaien (rotate). De toepassing zit in allerlei rekenkundige bewerkingen.

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



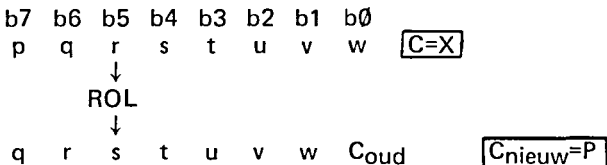
Bovenaan staat de bitpositie (bee-zoveel). De letters P . . . W stellen een bepaald bitpatroon voor van enen en nullen. We zien dat alle bits naar rechts schuiven; het meest linkse bit wordt ∅ en bit b0=W bepaalt de carry-vlag. Is W ∅ dan wordt de carry gereset, bij een 1 voor W geset. De N-vlag blijft ∅. Acht keer achter elkaar een LSR levert een accu-inhoud ∅∅∅∅∅∅∅∅ op. Uitsluitend dan wordt de Z-vlag geset.

**ASL**—Arithmetic Shift Left (opcode 0A). De bits schuiven nu naar links:



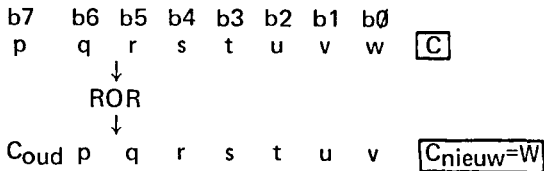
Alle bits schuiven een plaatsje naar links; het meest rechtse bit wordt ∅ en bit b7 (was P) bepaalt de carry-vlag: 1 bij P=1 en ∅ bij P=∅. De N-vlag wordt geset zodra bit zeven 1 (Q) is. De Z-vlag wordt geset bij een accu-inhoud ∅∅∅∅∅∅∅∅, die bijvoorbeeld optreedt na acht opeenvolgende ASL's.

**ROL**—ROTate Left (opcode 2A). Eens kijken:



Ook hier schuiven de bits een plaats naar links op. Het verschil met ASL is dat nu bit b0 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 0 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-instructies van daarnet kunnen ook worden losgelaten op de combinatie: carry-vlag-inhoud van een geheugenplaats. Het gaat dan om instructies met absolute adressering of varianten daarvan.

### *Schuiven en draaien — een voorbeeld*

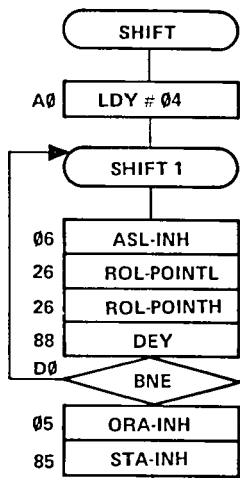
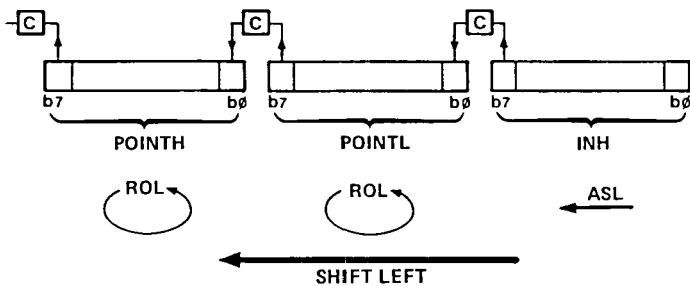
We weten al dat ingetoetste data bij de junior-computer van rechts naar links in het display doorschuift. We weten ook van figuur 14 dat de weer te geven data ligt opgeslagen in een drietal display-buffers: de geheugenplaatsen met adressen 00FB, 00FA en 00F9. Het doorschuiven van weer te geven informatie is in figuur 19 in beeld gebracht. Zodra een nieuwe toets wordt ingedrukt start de routine SHIFT van het monitorprogramma. 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 beetje 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 06) en wordt bit nul 0. Vervolgens laat de monitor een ROLZ (opcode 26) los op POINTL; de b0 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 0 geworden. De laatste twee rechthoeken van figuur 19 betreffen het gelijk maken van deze vier bits aan de toetswaarde van de ingedrukte toets.

*Nu volgt een programma waarbij gebruik wordt gemaakt van een aantal zojuist besproken instructies 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



80915-3-19

Figuur 19. Programma waarmee ingetoetste data van links naar rechts het display inschuift. Deze routine maakt ook deel uit van het monitorprogramma ("de monitor").

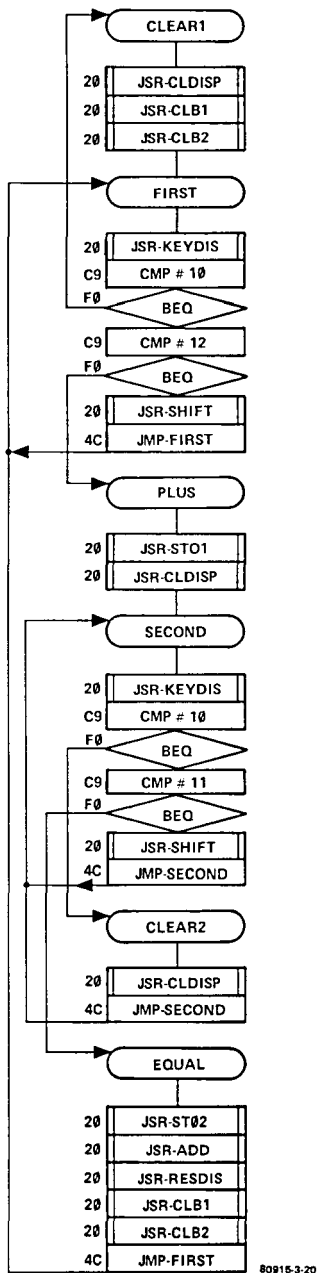
papier voor een paar dubbeltjes. Een speciaal optelprogramma maken lijkt op het te lijf gaan van een mug met een kanon. Is dat ook. Maar door dit te doen leren we het kanon te gebruiken voor meer ingewikkelde, vreedzame toepassingen.

We willen als volgt optellen:

$$\begin{array}{cccccccc}
 X & X & X & X & X & X & & + & X & X & X & X & X & X & = & X & X & X & X & X & X \\
 \text{eerste getal} & & & & & & & & \text{tweede getal} & & & & & & & \text{resultaat} & & & & & & 
 \end{array}$$

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 0 ... 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 10 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 sub-routines worden er gebruikt; de deelprogramma's staan in figuur 21.

Nog een paar opmerkingen vooraf. Er zijn  $4 \times 3 = 12$  geheugenplaatsen (buffers) nodig om de diverse data in kwijt te raken. Naam en adres van elk volgt hieronder:

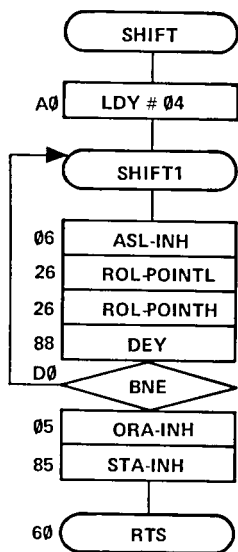
POINTH 00FB	POINTL 00FA	INH 00F9	display-buffers
B12 0002	B11 0001	B10 0000	buffers voor het eerste getal
B22 0005	B21 0004	B20 0003	buffers voor het tweede getal
R2 0008	R1 0007	R0 0006	buffers voor het resultaat van de optelling

waarbij elke buffer twee displays bedient. De volgorde van links naar rechts is gelijk aan de volgorde waarin men een getal opschrijft.

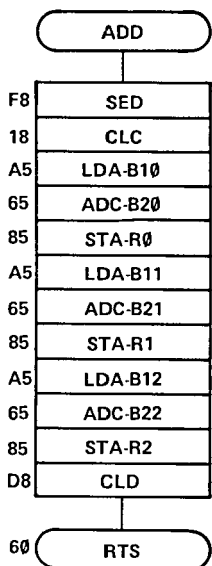
Het programma (figuur 20). Begonnen wordt met CLEAR1, met als gevolg dat display- en getalbuffers met 00000000 worden gevuld (zie ook de figuren 21d, 21e en 21f). Dan volgt FIRST: meteen springt het programma naar de subroutine KEYDIS (figuur 21c), waarin gebruik wordt gemaakt van SCANDS en GETKEY, oude 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#10 wordt er vervolgens getest of de CLEAR-toets soms is ingedrukt (dat is dus AD ). De instructie CPM#12 test of + is ingedrukt. Gaat het niet om deze stuurtoetsen dan blijven cijfertoetsen ( 0 . . . 9 ) of = als enige over. In SHIFT (figuur 21a) wordt dan het getal de display-buffer 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 display-buffers (= het eerste getal) naar de buffers B12-B11-B0. In de display-buffers 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: Weer 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.

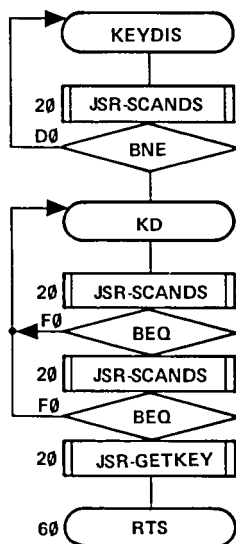
Er is nog een andere mogelijkheid: toets = (DA) is ingedrukt. Dit be-



80915-3-21a



80915-3-21b



80916-3-21c

Figuur 21a t/m 21i. Zeven 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.

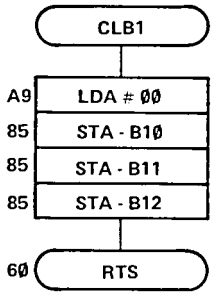
## Absolute Indexed, X Addressing

### *Absolute X-geïndexeerde adressering*

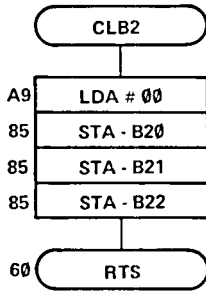
Als eerste van de geïndexeerde adresseringsmogelijkheden komt de boven genoemde aan de orde. Inderdaad, een hele mondvol, de terminologie wordt er niet eenvoudiger op. Het programmeren wél.

Wat was absolute adressering ook weer? Instructies met absolute adressering bestaan uit drie bytes. Het eerste byte is voor de opcode van de instructie; de volgende twee bytes (ADL, ADH) geven het adres waar de data is waarop de instructie wordt losgelaten.

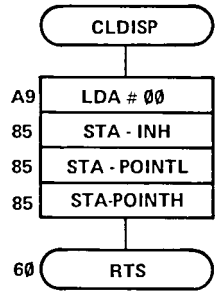
Bij absolute X-geïndexeerde adressering gaat het om een variant van



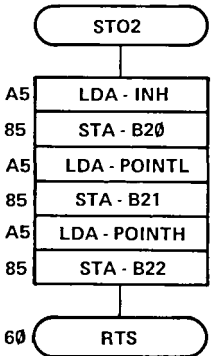
80915-3-21d



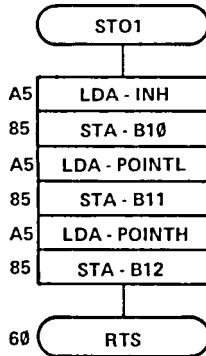
80915-3-21e



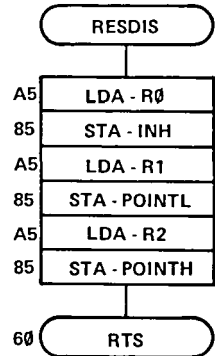
80915-3-21f



80915-3-21g



80915-3-21h



80915-3-21i

B10 → 0000  
 B11 → 0001  
 B12 → 0002

B20 → 0003  
 B21 → 0004  
 B22 → 0005

R0 → 0006  
 R1 → 0007  
 R2 → 0008

INH → 00F9  
 POINTL → 00FA  
 POINTH → 00FB

SCANDS → 1D8E  
 GETKEY → 1DF9

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 instructie. Waar dan wel? De operand-data van de instructie is te vinden op het adres dat gelijk is aan het opgegeven adres *plus* de inhoud van het X-register.

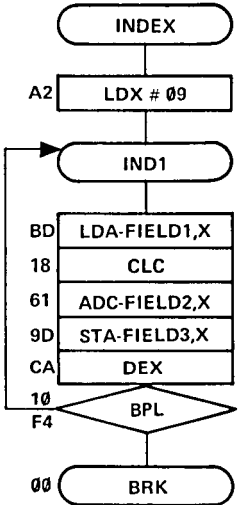
Vergelijk het met de huiselijke betekenis van een adres. De post is niet geadresseerd op "Dorpsstraat 106", maar op "Dorpsstraat 100, drie huizen verder" (even nummering). De drie is te vergelijken met de inhoud van het X-register.

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:	0120	FIELD2:	0011	FIELD3:	0300
0120	01	0011	10	0300	11
0121	02	0012	20	0301	22
0122	03	0013	30	0302	33
0123	04	0014	40	0303	44
0124	05	0015	50	0304	55
0125	06	0016	60	0305	66
0126	07	0017	70	0306	77
0127	08	0018	80	0307	88
0128	09	0019	90	0308	99
0129	0A	001A	A0	0309	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.



80915-3-22

**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 (binair optellen) 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 09. Dan de al beschreven optelling en dan wordt de inhoud van X met één verlaagd. De eerste optelling heeft dus betrekking op adres 0120+09=0129 van FIELD1, 0011+09=001A van FIELD2 en 0300+09=0309 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 instructie-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 binair optellen
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

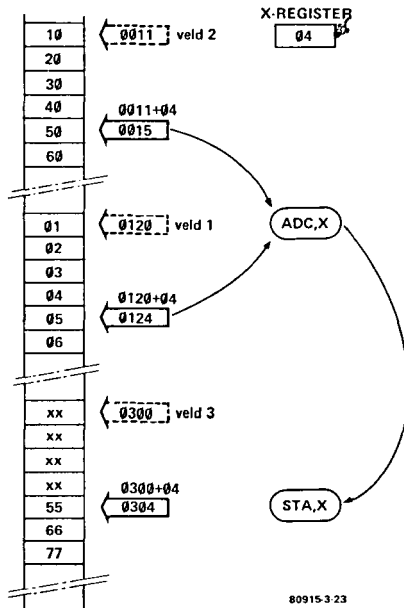
En dan hebben we nog maar een keer opgeteld, dus een regel van de tabel uitgevoerd. Bij een korte tabel van het voorbeeld, met velden van tien geheugenplaatsen vergt de eigenlijke opteloperatie al  $10 \times 10 = 100$  geheugenplaatsen. Vergelijk dat nou eens met het programma van figuur 22: met kop 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 04 van het X-indexregister.

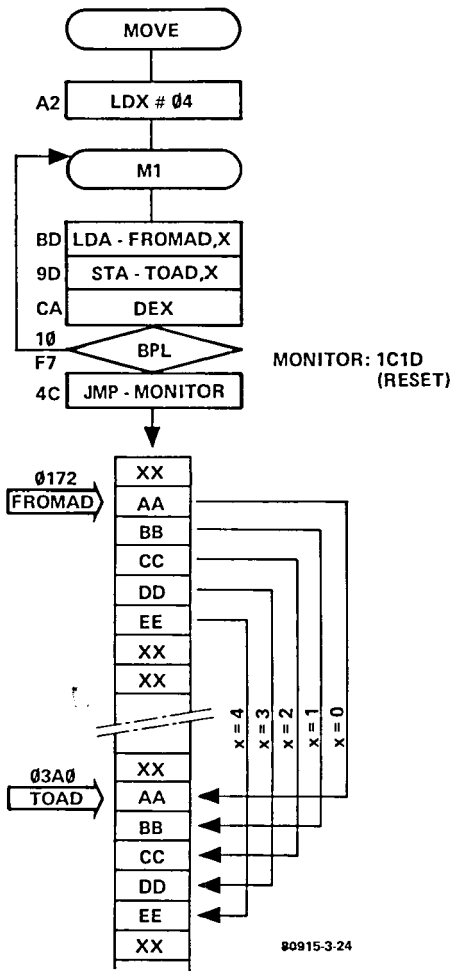
De instructies met absolute X-geïndexeerde adressering zijn terug te vinden in het instructie-overzicht achterin dit boek.

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 0172) naar het andere (TOAD, adres 03A0). 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 04 (LDX#04); na een datatransport (eerst adres 0172+04=0176) wordt de inhoud ervan met 1 verlaagd. Dat gaat zo door tot dat X nul is; dan is alle data verhuisd, of juist: 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	
0	2	0	0	0200	xx
DA		A	2	0200	A2 LDX#
+		0	4	0201	04
+		B	D	0202	BD LDA-,X
+		7	2	0203	72 ADL van FROMAD
+		0	1	0204	01 ADH van FROMAD
+		9	D	0205	9D STA-,X
+		A	0	0206	A0 ADL van TOAD
+		0	3	0207	03 ADH van TOAD
+		C	A	0208	CA DEX
+		1	0	0209	10 BPL
+		F	7	020A	F7 offset
+		4	C	020B	4C JMP
+		3	3	020C	33 ADL van monitor-adres
+		1	C	020D	1C ADH van monitor-adres
AD				020D	1C
0	2	0	0	0200	A2
GO				0200	A2 programmastart



Figuur 23. Een momentopname van het programma van figuur 22, voor de situatie: X=04.



Figuur 24. Programma voor het verhuizen van een datablok.

Aan het eind van het programma springen we naar de monitor (vektor 1C33); na de uitvoering van het programma verschijnt het startadres in beeld.

N.B. Het X-register bestaat uit acht bytes. Op de aangegeven manier kunnen dus datablokken van maximaal 256 bytes worden getransporteerd.

## 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.

Dat is zeer handig. Bij het nesten van subroutines bijvoorbeeld. Meer hierover in volgende hoofdstukken. De opcodes staan in het instructie-overzicht, achterin het boek. Let op! Niet alle instructies zijn in beide versies (X- en Y-geïndexeerd) beschikbaar. Sommige alleen X- en LDX-alleen in Y-.

### **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: 00. Met als voordeel dat de instructie 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. De opcodes van de instructies met deze vorm van adresseren vinden we weer achterin het boek.

### **Zero Page Indexed, Y Addressing**

#### *Y-geïndexeerd adresseren op pagina nul*

Zie boven, met Y in plaats van X. Uit het instructie-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 directe adressering. Daar komt nu verandering in.

### **Indirekte adressering**

*De laatste drie van de dertien adresseringsmogelijkheden komen nu aan bod. Het einde van dit hoofdstuk is in het zicht.*

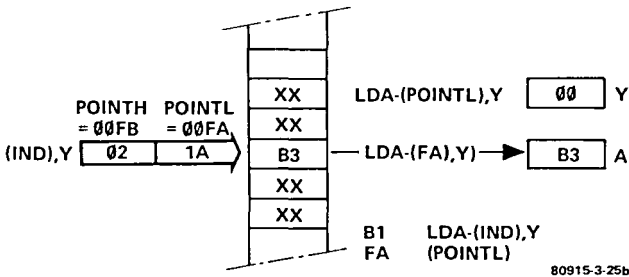
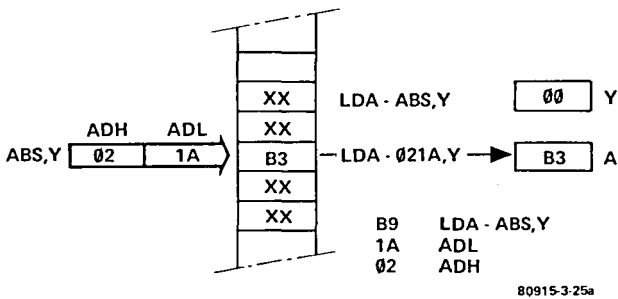
Indirekte adressering is geen eenvoudige zaak om uit te leggen. We hebben er zo lang mogelijk mee gewacht, maar nu is het dan zo ver. Hebben we het echter door, dan wordt het programmeren een stuk eenvoudiger; korter en overzichtelijker.

Een groot voordeel van indirecte 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 adresseringsmethode is het grote succes van de 6502-microprocessor te danken.

### **Indirect Indexed Addressing**

#### *Indirekte 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 021A. In het Y-register staat 00. Het laad-programmadeel ziet er zo uit:



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

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

Hierdoor komt de inhoud van adres 021A (B3) in de accu terecht. Immers,  $Y=00$ , dus het operand-adres is  $021A+00=021A$ . 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 indirecte geïndexeerde adressering, met (een deel van) een adres als operand is de data waarop de instructie 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 instructie te bewerken data zich bevindt.

Terug naar figuur 25b. De instructie die het laden van de accu met de inhoud van geheugenplaats 021A tot gevolg heeft is een instructie met indirecte 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 indirecte adres.

Het linker byte van het indirecte adres is 00; de geheugenplaats bevindt zich namelijk op pagina nul. Het adres 00FA noemen we POINTL. Op dat adres ligt het rechter byte van het effectieve adres. Het linker byte van het effectieve adres ligt één geheugenplaats verder, op adres 00FB, eveneens op pagina nul; dit adres is POINTH gedoopt.

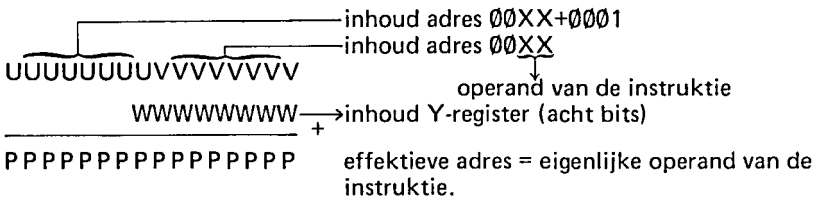
We weten van figuur 25a dat het effectieve adres 021A moet zijn, dus de inhoud van POINTL is 1A en van POINTH 02.

We zien dat achter de opcode van de instructie (in ons geval LDA-) FA staat. Dus slechts één van de vier bytes van de twee adressen waar het effectieve 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 indirecte 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 effectieve adres. Hebben we gezegd. Dat is alleen waar als de inhoud van register Y 00 is. En dat is ie in figuur 25b. Het effectieve 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 wordt opgeteld bij byte U.

Ofschoon voor het vastleggen van het effectieve adres in feite twee adressen nodig zijn (namelijk 00XX en 00XX+0001) spreken we van *het* indirecte adres in plaats van *de* indirecte adressen. Op het indirecte adres ligt een direkt oftewel absoluut adres, dat na rekening (optelling) te hebben gehouden met Y overgaat in het effectieve adres.

Instructies met indirecte Y-geïndexeerde adressering zijn twee bytes lang. Het eerste byte bevat de opcode, het tweede het rechter byte van een adres op pagina nul. Een overzicht van instructies, met opcodes en met deze vorm van adressering staat achterin het boek. U zult zien dat de mnemonics van de instructies allemaal al minstens een keer eerder aan de orde zijn gekomen.

### *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  $255 \times 256 = 65.280$  bytes mogelijk is? Niet gek. En bovendien een goed voorbeeld van wat er mogelijk is met indirecte Y-geïndexeerde adressering. Want daar hebben we het nog steeds over.

Als overture 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 0200, het laatste 02FF, 255 geheugenplaatsen verder. Het beginadres komt te staan op twee adressen op pagina nul; op adres BEG 00 en op adres BEG+1 02. Het eerste datablok (dat overigens op pagina 02 staat) moet worden verhuisd naar pagina A8, met beginadres A800 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 00 en op adres MOV+1 A8. De diverse adressen op pagina nul met de namen van de op het adres verblijvende bytes zijn:

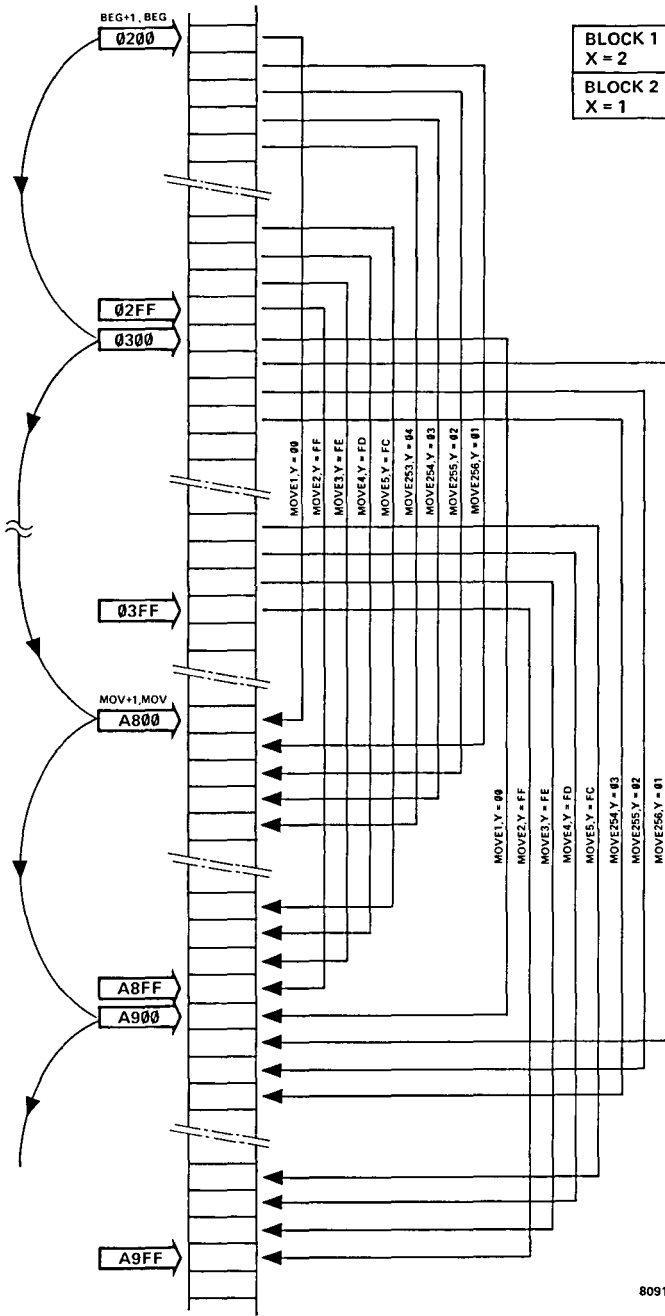
BEG	op adres 0000. Inhoud heet FRADL en is gelijk aan 00
BEG+1	op adres 0001. Inhoud heet FRADH en is gelijk aan 02
MOV	op adres 0002. Inhoud heet TOADL en is gelijk aan 00
MOV+1	op adres 0003. Inhoud heet TOADH en is gelijk aan A8
BLOCKS	op adres 0004. Inhoud heet N en is gelijk aan 02 (= aantal te verhuizen datablokken)

Die namen zitten logisch inelkaar, dachten we: FR slaat op FFrom 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 dat de indirecte adressen 0000 . . . 0004 laadt met de juiste data; het stroomdiagram ervan staat in figuur 27. Nadat de eigenlijke taken van DETMOV 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.

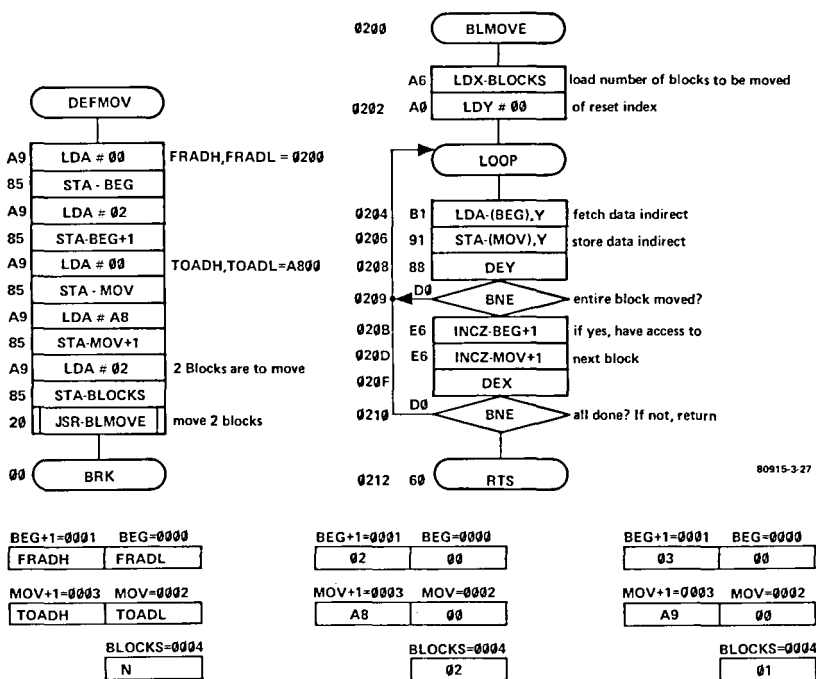
Het X-register is geprogrammeerd als blokteller; het wordt geladen met de inhoud van BLOCKS. Het Y-register doet dienst als echt index-register; de inhoud ervan geeft aan hoe ver de verhuizing van data binnen een blok is gevorderd. Is de inhoud van Y 21, dan is de 33-ste geheugenplaats van het blok aan bod. Aan het begin van BLMOVE is Y geladen met 00. Het transport van data verloopt met de accu als tussenstation; de hierbij betrokken instructies LDA en STA zijn de versies met indirecte geïndexeerde adressering.

Na een datatransport wordt de inhoud van Y met 1 verlaagd (DEY); Y doorloopt achtereenvolgens de waarden 00, FF, EE, . . . , 02, 01, 00, FF, FE, enzovoorts. Via een spronginstructie 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 verlaagd; ook de inhoud van de blokteller Y 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 DETMOV en volgt ter plaatse een BRK: het programma is klaar.



80915-3-26

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 indirecte Y-geïndexeerde adressering.

Nu nog wat meer details. Zoals al gezegd bevatten indirecte adressen zoals BEG & BEG+1, en MOV & MOV+1 directe adressen die met inachtnaam van de inhoud van het Y-indexregister van belang zijn voor de transport-instructies binnen het programmadeel LOOP. Het gaat telkens om twee opeenvolgende adressen op pagina nul. Reden genoeg voor de introductie van een nieuwe kreet: de *address pointer ofwel adreswijzer*. Dat is een patroon van zestien bits dat ontstaat door het aaneelkaar plakken van twee bytes, de inhoud van het indirecte adres. De wijzer staat gericht op het directe ofwel absolute adres dat wordt gevormd door de inhoud van het indirecte 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 02FF en A8FF en op een ander moment op 03FF en A9FF.

De dynamische pointer van het vertrekveld wijst op het adres met de eigenlijke operand van de instructie LDA; die van het aankomstveld op het adres met de eigenlijke operand van de instructie STA.

## Indexed Indirect Addressing

### *X-geïndexeerde indirecte adressering*

Let op het verschil met de vorige adresseringsmethode: "indirekt" en "geïndexeerd" hebben stuivertje gewisseld en "Y-" is vervangen door "X-". Dat stuivertje wisselen betekent indexering bij het vaststellen van het indirecte adres; het indirecte adres bevat het effectieve adres, dat de eigenlijke operand-data van de instructie bevat.

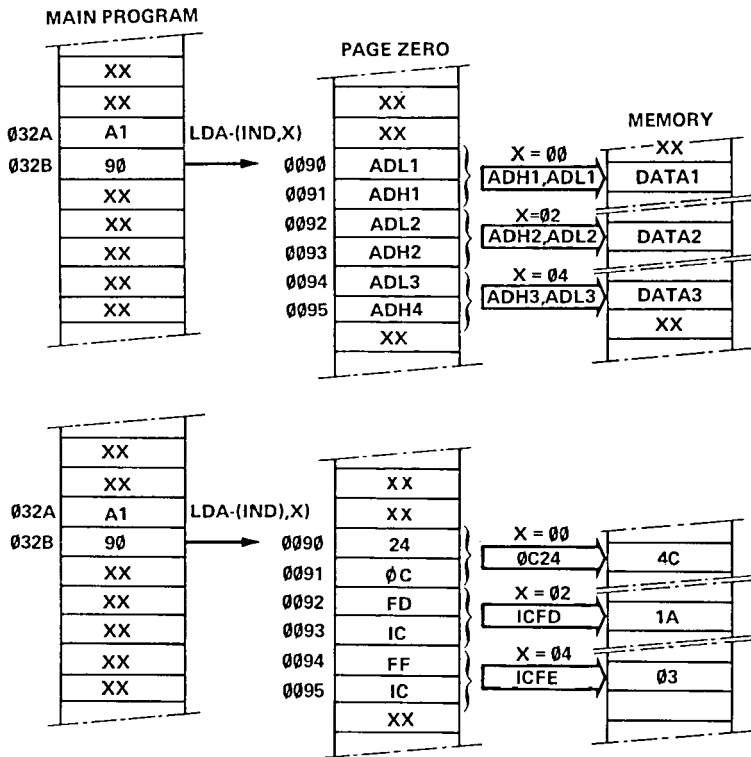
Het naadje van de kous: ook hier gaat het om instructies van twee bytes lang. Op de opcode volgt een byte van een adres op pagina nul. Maar nu komt het grote verschil: het eigenlijke indirecte adres ontstaat uit het tweede byte van de instructie en uit de inhoud van het X-register: bij dat tweede byte 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 indirecte adres, waarvan het linker byte 00 is vanwege de adressering op pagina nul. De tweede helft van het indirecte adres (dat het linker byte bevat van het effectieve 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 indirecte en geïndexeerde adressering zijn:

- Bij indirecte Y-geïndexeerde adressering ligt het indirecte adres direct vast via het tweede byte van de instructie; het effectieve adres ontstaat indirect uit de inhoud van het indirecte adres, namelijk met inachtnaam van de inhoud van het Y-register (indexering).
- Bij X-geïndexeerde indirecte adressering volgt het indirecte adres indirect uit het tweede byte van de instructie, namelijk met inachtnaam van de inhoud van het X-register (indexering); het effectieve adres volgt direct uit de inhoud van het indirecte 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 instructie LDA in zijn werk gaat onder gebruikmaking van X-geïndexeerde indirecte 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 concrete 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.

Op adres 032A van MAIN PROGRAM staat A1, de opcode van LDA. De notatie is: LDA-(IND,X). (Bij indirecte Y-geïndexeerde adressering zou dat



80915-3-28

Figuur 28. Een voorbeeld van X-geïndexeerde indirecte adressering. Deze vorm van adressering wordt betrekkelijk weinig toegepast.

zijn LDA-(IND),Y, met opcode B1). Op het volgende adres staat 90. Daarbij wordt opgeteld de inhoud van Y. Is deze 00, dan is het effectieve adres te vinden op de geheugenplaatsen 0090 (ADL1) en 0091 (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=01. We komen dan op pagina nul op de geheugenplaatsen 0091 en 0092 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 02 of in ieder geval met even bedragen te wijzigen.

Bij X=02 ligt het effectieve adres in 0092 en 0093; de adreswijzer ADH2, ADL2 staat gericht op DATA2. Bij X=04 ligt het effectieve adres in 0094 en 0095; de adreswijzer ADH3, ADL3 staat gericht op geheugenplaats DATA3 van MEMORY.

Het nut van deze adresseringsmethode 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.

X-geïndexeerde indirecte adressering wordt overigens betrekkelijk weinig toegepast. Bij de bespreking van het I/O-gebeuren, in boek 2 komen we er nog een keer op terug. Een instructie-overzicht met opcodes ziet men achterin het boek.

## Onderbreking van het lopende programma: NMI, IRQ en RESET

### *Interrupts en (nogmaals) indirecte 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 actie onderneemt. Actie, 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 van 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 essentiële programma-informatie zoals de inhoud van de programmateller PC en van het programma-statusregister P tijdelijk opgeslagen op de stack (stapel).

Bij een subroutine na een interrupt gaat het precies zo. Alleen veroorzaakt nu niet de instructie JSR de sprong naar een subroutine, maar een bepaald elektrisch signaal op een pen van het microprocessor-IC. Dus in het geval van een interrupt maken niet 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 programma-voortgang:

1. *Interrupt Request, IRQ* ofwel onderbrekingsaanvraag. 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 "aanvraag" te negeren, naast zich neer te leggen. Dat gaat via het beïnvloeden van de interrupt-vlag I door een van de beide instructies:

CLI (opcode 58) waardoor I=0. 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  $\mu P$  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-instructies of, nog vollediger, hardware-spronginstructies, die bij een NMI een onvoorwaardelijk karakter hebben en bij een IRQ een voorwaardelijk karakter. Op het punt van de hardware is er nog een vermeldenswaardig verschil tussen de beide interrupt-mogelijkheden. Een NMI wordt veroorzaakt door de overgang van logisch 1 naar 0 op de NMI-ingang; dus bij een snelle daling tot 0 volt van de elektrische spanning. Bij een IRQ moet de IRQ-aansluiting spanning voeren, logisch 1 zijn, wil er eventueel een onderbreking optreden.

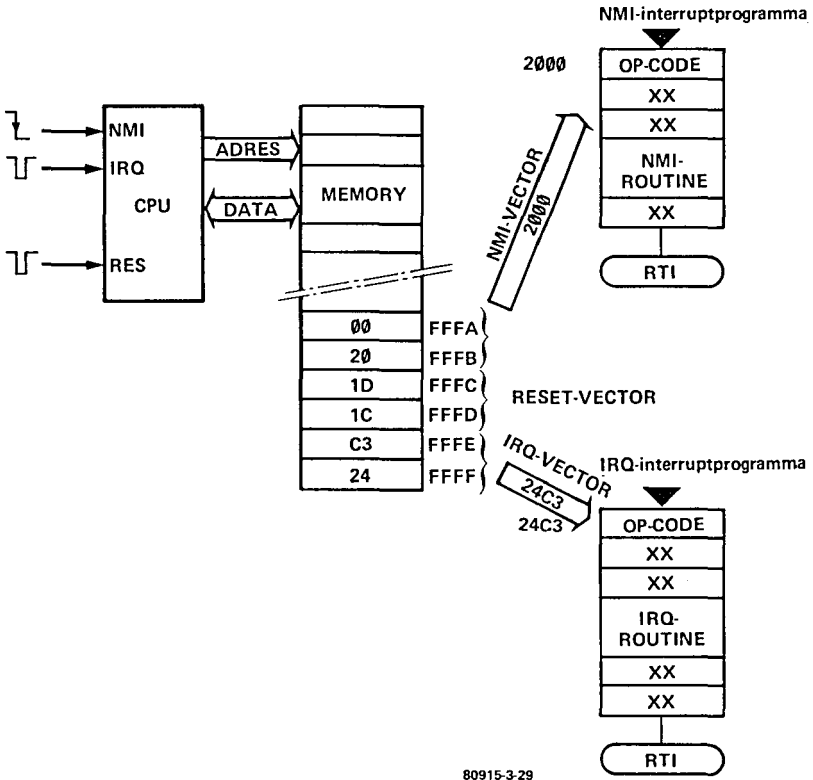
### *NMI-afwikkeling*

Zodra een NMI optreedt gaat de microprocessor te rade bij de geheugenplaatsen met adressen FFFA en FFFB; dat wil zeggen nadat de lopende programma-instructie is afgewerkt. Te rade voor wat? Hij moet weten op welke geheugenplaats de eerste instructie van de af te handelen interrupt-subroutine op hem ligt te wachten. In figuur 29 is dat adres 2000. 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 instructie voor instructie 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 instructie RTI, met opcode 40.

### *IRQ-afwikkeling*

Voert de IRQ-aansluiting gedurende een voldoende lange tijd spanning, dan is er sprake van interruptie-aanvraag. 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 instructie 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 0) wordt I 1 gemaakt. Dit om te voorkomen dat een interrupt-subroutine niet 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 instructie RTI (*niet* RTS).

Het nul maken van de vlag I kan via de instructie CLI of RTI. Er moet daarbij op worden gelet dat de IRQ-aansluiting van de microprocessor 1 is 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 "vlaggenreger")? 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 instructie die aan bod was ook daadwerkelijk aan bod is.

### *Reset*

De computer (microprocessor) kan op nog een derde manier kwa software via hardware worden beïnvloed, 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).

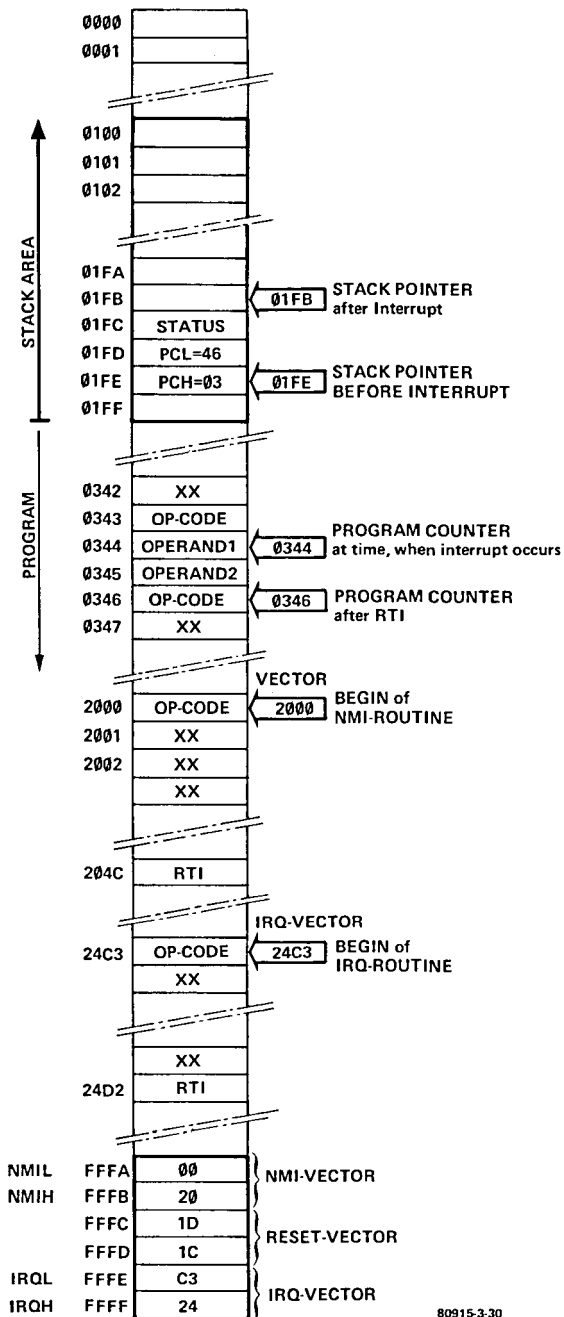
Bij de junior-computer wijst de reset-vektor altijd naar het adres 1C1D, dat hoort bij het monitorprogramma (systeem-monitor). Drukt men, bijvoorbeeld na het inschakelen van de junior-computer op de toets RST dan ontstaan via-via een RES voor de microprocessor en gaat het op naar de geheugenplaatsen 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  $8 \times 4 = 32$  pagina's 00...1F zijn volledig gedecodeerd.

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 0 of 1 zijn, dus in dit geval is 11111111 gelijk aan XXX11111, bijvoorbeeld 00011111, 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 01 is gebruikt als stack; pagina 03 bevat het te onderbreken programma. Verder zien we dat de IRQ-vektor wijst naar 24C3, het start-adres van de IRQ-subroutine en dat de NMI-vektor wijst op het adres (2000) waar de NMI-subroutine begint.

Stel het programma is bezig met geheugenplaats 0343. Er wordt een opcode van een instructie 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, huis-houdelijke akties in detail te beschrijven, maar waar het om gaat is dat het linker byte PCH van de programmateller (03) op de stack komt op die plaats die door de stapelwijzer is aangewezen. Dat is namelijk de eerst-



80915-3-30

**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 verlaagd en gaat het rechter byte PCL de stapel op. Na de daarop volgende verhoging met 1 van de stapelwijzer staat deze gericht op adres 01FC. Daar gaat de inhoud van het P-register naar toe. Hierna zoekt de stapelwijzer het weer eentje hoger op en staat dan gericht op 01FB.

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 2000. 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 0343 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 instructie met indirecte adressering JMP. Nu volstaat de opmerking dat via *een* NMI-vektor *verscheidene* NMI-subroutines kunnen worden aangeroepen.

Net zo als binnen een subroutine verschillende subroutines kunnen worden "besprongen" kan met meervoudige interrupts binnen een interrupt-subroutine verschillende interrupt-routines worden aangesproken.

### *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 instructie RTI op de terugkeer naar het onderbroken programma. In het voorbeeld van figuur 30 staat een RTI op adres 204C. Er gebeurt nu van alles. Eerst wordt de stapelwijzer met 1 verhoogd en wijst dus op adres 01FC. 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 verhogen van de stapelwijzer (01FD) krijgt PCL van de programmateller zijn oude waarde terug; nogmaals met 1 ophogen van de stapelwijzer (01FE) levert de oude waarde van PCH. Daarmee is het terugkeeradres bekend. In het geval van figuur 30 is dat 0346. We zijn terug in het onderbroken programma. Dat zich vervolgens verder afwickelt.

### *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"-instructies, die de inhoud van zo'n intern register op de stapel zetten. Ze moeten er dan natuurlijk wel weer af kunnen worden gehaald. En dat gaat met

"pull"-instructies.

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

```
SAVE PHA red accu-inhoud op de stapel
      (waarna de stapelwijzer met 1 wordt verlaagd)
      TXA breng inhoud X-register naar de accu
      PHA red accu-inhoud op de stapel
      (waarna de stapelwijzer met 1 wordt verlaagd)
      TYA breng inhoud Y-register naar de accu
      PHA red accu-inhoud op de stapel
      . . . eerste instructie interrupt-subroutine
      .
      .
      .
RESTO PLA breng 't bovenste van de stapel naar de accu
      (waarna de stapelwijzer met 1 wordt verhoogd)
      TAY breng accu-inhoud naar Y-register
      PLA breng 't bovenste van de stapel naar de accu
      (waarna de stapelwijzer met 1 wordt verhoogd)
      TAX breng accu-inhoud naar X-register
      PLA breng 't bovenste van de stapel naar de accu
      (herstel oorspronkelijke accu-inhoud)
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.

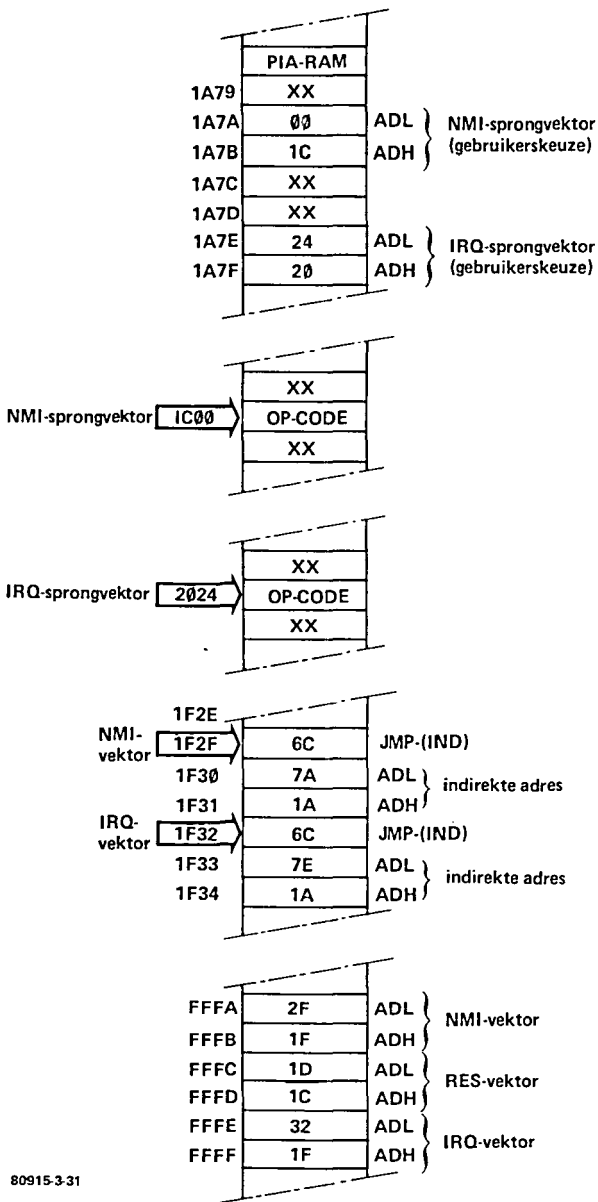
*Als laatste, dertiende adresseringsmogelijkheid komt nu de indirecte adressering aan de orde. Het gaat om slechts één instructie, namelijk JMP-indirekt. Maar wèl erg belangrijk, omdat we nu eindelijk aan de weet komen hoe het komt dat met één NMI verscheidene interrupt-subroutines kunnen worden bereikt.*

## Indirekte adressering

*Na JMP-indirekt alle dertien goed (behandeld)*

De instructie JMP kennen we al. Althans in de versie met absolute (direkte) adressering. Het is de instructie om onvoorwaardelijk te springen naar een ander deel van het geheugen, waar dan nieuwe instructies liggen. Het is een instructie 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 indirecte 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 instructie 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



80915-3.31

**Figuur 31.** Het verwerken ("bedienen" of "servicen") van interrupts via instructies JMP met indirecte adressering. De inhoud van het indirecte adres wijst op het effectieve (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 indirecte sprong bekijkt de  $\mu$ P automatisch twee opeenvolgende geheugenplaatsen; de eerste bevat de ADL, de tweede de ADH van de sprongvektor.

De notatie van de instructie 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 indirecte sprong naar de adressen die de sprongvektor vastleggen, en van een *effektieve sprong* naar het sprongadres, waarop de sprongvektor staat gericht.

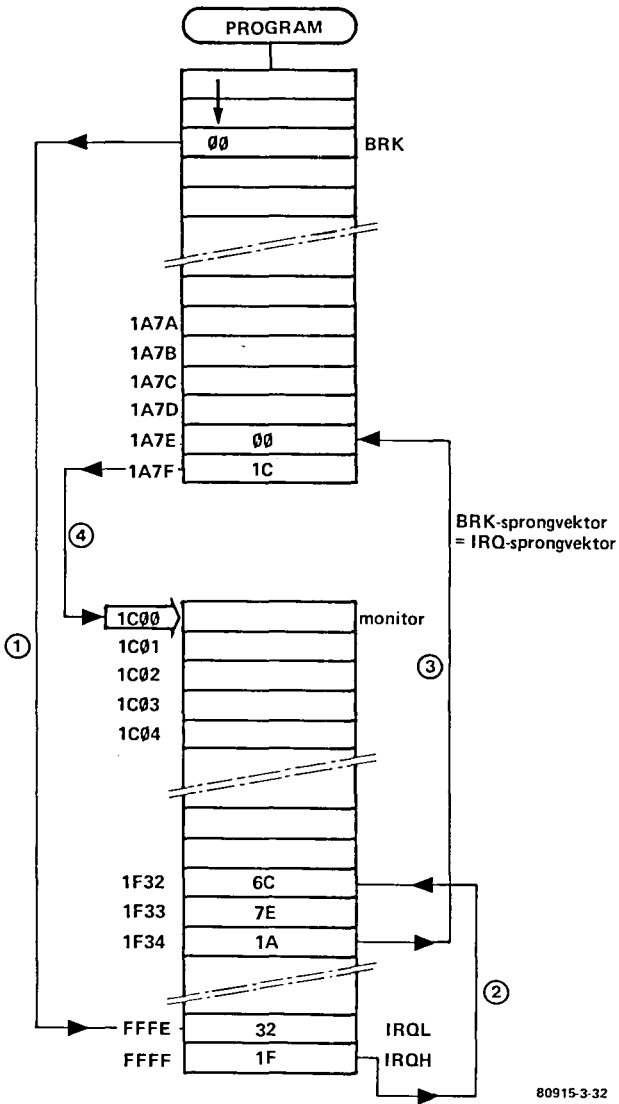
Wat is het nut van dit indirecte springen bij het afhandelen van een NMI? Laten we dat eens gaan bekijken aan de hand van figuur 31. Stel er is een NMI. De computer kijkt dan op de adressen FFFA en FFFB, op zoek naar de NMI-vektor. Die blijkt 1F2F te zijn. Dat is nu niet het startadres van een subroutine, maar een adres waarop de opcode staat van de instructie JMP-(IND). De inhoud van IND is 1A7A (= de inhoud van de adressen 1F30 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-sprongvectoren moeten kunnen worden bereikt. Wil dat allemaal tot een goed einde komen dat is het van belang dat voordat de NMI optreedt de indirecte adressen met sprongvectoren 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 1C00. 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 2024.

De clou van de sprongvectoren in (PIA-)RAM is dat het effectieve 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 indirecte adres wordt geladen met een nieuw effectief (direct, absoluut) sprongadres dat hoort bij een andere interrupt-subroutine. Vooral het gebruik van BRK's wordt hierdoor buitengewoon interessant.

Een instructie JMP met indirecte adressering heeft nog een andere nuttige toepassing, die niets met interrupts heeft te maken. Iets heel anders dus. Een programma kan op de indirecte adressen na (die komen dan in RAM, zoals in figuur 31 op pagina 1A) al geheel in EPROM worden gezet; er kan dan van hartelust worden geëxperimenteerd met de effectieve adressen, namelijk via een JMP-(IND) en verschillende sprongvectoren.



**Figuur 32.** De instructie BRK is de software-versie van de hardware-instructie 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 standaard situatie 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"-instructies NMI en IRQ. Het kan ook via 100% software. Dus met zo'n ouderwetse instructie met opcode en zo. Dat is de instructie BRK, met opcode 00, 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 00, treedt de IRQ-situatie op. Na een BRK gaat het dus naar 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 indirecte 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-instructie is het "debuggen", foutloos maken van een gebruikersprogramma. We gaan er hier niet in detail op in (dat komt wel in boek 2), 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 controle blijkt dat de eerste helft van een programma goed is en dat vanaf een zekere plaats verkeerde instructies, met van de juiste afwijkende instructielengte zijn gebruikt, of instructies 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 instructies 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. Zoals gezegd, meer hierover in *Junior-computer 2*.

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 0.

*Zo, dat was dan hoofdstuk 3. Het was de laatste bladzijden wel erg veel theorie, 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).*

# 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 ook 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 instructies. 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 0200 tot en met 0248 : hoofdprogramma (figuur 20)  
adressen 0249 tot en met 0258 : subroutine SHIFT (fig. 21a)  
adressen 0259 tot en met 026E : subroutine ADD (figuur 21b)  
adressen 026F tot en met 0281 : subroutine KEYDIS (fig. 21c)  
adressen 0282 tot en met 028A : subroutine CLB1 (fig. 21d)  
adressen 028B tot en met 0293 : subroutine CLB2 (fig. 21e)  
adressen 0294 tot en met 029C : subroutine CLDISP (fig. 21 f)  
adressen 029D tot en met 02A9 : subroutine STO2 (fig. 21g)  
adressen 02AA tot en met 02B6 : subroutine STO1 (fig. 21h)  
adressen 02B7 tot en met 02C3 : subroutine RESDIS (fig. 21i)

### *Vorbereiding toetsprogramma*

Terug naar het probleem van daarnet van het alles nog niet precies weten. Neem nou adres 0206. 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 "papierene" 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 instructie automatisch het adres van de volgende geheugenplaats bevat. En als op de bewuste geheugenplaats geen instructie 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 instructie NOP. Dezelfde instructie kunnen we op een aantal plaatsen invoegen (al in de papieren fase) om het vervelende effect van vergeten instructies en/of operanden, kortom het effect 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+0001, 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	
0	2	0 0	0200	xx	
DA		2 0	0200	20	JSR- <b>CLEAR1</b>
+		9 4	0201	94	ADL van CLDISP
+		0 2	0202	02	ADH van CLDISP
+		2 0	0203	20	JSR-
+		8 2	0204	82	ADL van CLB1
+		0 2	0205	02	ADH van CLB1
+		2 0	0206	20	JSR-
+		8 B	0207	8B	ADL van CLB2
+		0 2	0208	02	ADH van CLB2
+		2 0	0209	20	JSR- <b>FIRST</b>
+		6 F	020A	6F	ADL van KEYDIS
+		0 2	020B	02	ADH van KEYDIS
+		C 9	020C	C9	CMP #
+		1 0	020D	10	met 10
+		F 0	020E	F0	BEQ
+		F 0	020F	F0	F0 plaatsen terug is CLEAR1
+		C 9	0210	C9	CMP #
+		1 2	0211	12	met 12
+		F 0	0212	F0	BEQ <sup>1</sup>
+		0 6	0213	06	(offset) 06 plaatsen verder is PLUS
+		2 0	0214	20	JSR-
+		4 9	0215	49	ADL van SHIFT
+		0 2	0216	02	ADH van SHIFT
+		4 C	0217	4C	JMP- (onvoorwaardelijke sprong)
+		0 9	0218	09	ADL van FIRST
+		0 2	0219	02	ADH van FIRST
+		2 0	021A	20	JSR- <b>PLUS</b>
+		A A	021B	AA	ADL van STO1
+		0 2	021C	02	ADH van STO1
+		2 0	021D	20	JSR-
+		9 4	021E	94	ADL van CLDISP
+		0 2	021F	02	ADH van CLDISP
+		2 0	0220	20	JSR- <b>SECOND</b>
+		6 F	0221	6F	ADL van KEYDIS
+		0 2	0222	02	ADH van KEYDIS
+		C 9	0223	C9	CMP #
+		1 0	0224	10	met 10
+		F 0	0225	F0	BEQ
+		0 A	0226	0A	0A plaatsen verder (offset) is CLEAR2
+		C 9	0227	C9	CMP #
+		1 1	0228	11	met 11
+		F 0	0229	F0	BEQ
+		0 C	022A	0C	(offset); 0A plaatsen verder is EQUAL
+		2 0	022B	20	JSR-
+		4 9	022C	49	ADL van SHIFT
+		0 2	022D	02	ADH van SHIFT
+		4 C	022E	4C	JMP-
+		2 0	022F	20	ADL van SECOND
+		0 2	0230	02	ADH van SECOND

+	2	0	0231	20	JSR-	CLEAR2
+	9	4	0232	94	ADL van CLDISP	
+	0	2	0233	02	ADH van CLDISP	
+	4	C	0234	4C	JMP-	
+	2	0	0235	20	ADL van SECOND	
+	0	2	0236	02	ADH van SECOND	
+	2	0	0237	20	JSR-	EQUAL
+	9	D	0238	9D	ADL van STO2	
+	0	2	0239	02	ADH van STQ2	
+	2	0	023A	20	JSR-	
+	5	9	023B	59	ADL van ADD	
+	0	2	023C	02	ADH van ADD	
+	2	0	023D	20	JSR-	
+	B	7	023E	B7	ADL van RESDIS	
+	0	2	023F	02	ADH van RESDIS	
+	2	0	0240	20	JSR-	
+	8	2	0241	82	ADL van CLB1	
+	0	2	0242	02	ADH van CLB1	
+	2	0	0243	20	JSR-	
+	8	B	0244	8B	ADL van CLB2	
+	0	2	0245	02	ADH van CLB2	
+	4	C	0246	4C	JMP-	
+	0	9	0247	09	ADL van FIRST	
+	0	2	0248	02	ADH van FIRST	
N.B. Laatste regel van hoofdprogramma						
+	A	0	0249	A0	LDY #; subroutine	SHIFT
+	0	4	024A	04	04 in Y-indexregister	
+	0	6	024B	06	ASLZ	SHIFT1
+	F	9	024C	F9	INH op adres 00F9	
+	2	6	024D	26	ROLZ	
+	F	A	024E	FA	POINTL op adres 00FA	
+	2	6	024F	26	ROLZ	
+	F	8	0250	FB	POINTH op adres 00FB	
+	8	8	0251	88	DEY verlaag inhoud Y met 1	
+	D	0	0252	D0	BNE	
+	F	7	0253	F7	F7 plaatsen terug is SHIFT1	
+	0	5	0254	05	ORAZ	
+	F	9	0255	F9	bit voor bit ORren met INH	
+	8	5	0256	85	STAZ	
+	F	9	0257	F9	inhoud van accu naar INH (adr 00F9)	
+	6	0	0258	60	RTS terugkeer naar hoofdprogramma	
+	F	8	0259	F8	SED decimaal rekenen subroutine	ADD
+	1	8	025A	18	CLC	
+	A	5	025B	A5	LDAZ	
+	0	0	025C	00	ADL van B10 (adres 0000); B10 in accu	
+	6	5	025D	65	ADCZ	
+	0	3	025E	03	ADL van B20 (adres 0003); accu = B10 + B20	
+	8	5	025F	85	STAZ	
+	0	6	0260	06	ADL van R0; accu → R0	
+	A	5	0261	A5	LDAZ	
+	0	1	0262	01	ADL van B11 (adres 0001); B11 in accu	
+	6	5	0263	65	ADCZ	

+	0	4	0264	04	ADL van B21 (adres 0004); accu = B11 + B21
+	8	5	0265	85	STAZ
+	0	7	0266	07	ADL van R1 (adres 0007); accu → R1
+	A	5	0267	A5	LDAZ
+	0	2	0268	02	ADL van B12 (adres 0002); B12 in accu
+	6	5	0269	65	ADCZ
+	0	5	026A	05	ADL van B22 (adres 0005); accu = B12 + B22
+	8	5	026B	85	STAZ
+	0	8	026C	08	ADL van R2 (adres 0008); accu → R2
+	D	8	026D	D8	CLD herstel situatie van binair rekenen
+	6	0	026E	60	RTS terugkeer naar hoofdprogramma
+	2	0	026F	20	JSR- subroutine <b>KEYDIS</b>
+	8	E	0270	8E	ADL van SCANDS } in monitor
+	1	D	0271	1D	ADH van SCANDS } (1D8E)
+	D	0	0272	D0	BNE
+	F	B	0273	FB	(offset); FB plaatsen terug is KEYDIS
+	2	0	0274	20	JSR- <b>KD</b>
+	8	E	0275	8E	ADL van SCANDS } in monitor
+	1	D	0276	1D	ADH van SCANDS } (1D8E)
+	F	0	0277	F0	BEQ
+	F	B	0278	FB	(offset); FB plaatsen terug is KD
+	2	0	0279	20	JSR-
+	8	E	027A	8E	ADL van SCANDS } in monitor
+	1	D	027B	1D	ADH van SCANDS } (1D8E)
+	F	0	027C	F0	BEQ
+	F	6	027D	F6	(offset); F6 plaatsen terug is KD
+	2	0	027E	20	JSR-
+	F	9	027F	F9	ADL van GETKEY } in monitor
+	1	D	0280	1D	ADH van GETKEY } (1DF9)
+	6	0	0281	60	RTS terugkeer naar hoofdprogramma
+	A	9	0282	A9	LDA #; subroutine <b>CLB1</b>
+	0	0	0283	00	00 → accu
+	8	5	0284	85	STAZ
+	0	0	0285	00	accu → B10 (= 00)
+	8	5	0286	85	STAZ
+	0	1	0287	01	00 → B11
+	8	5	0288	85	STAZ
+	0	2	0289	02	00 → B12
+	6	0	028A	60	RTS terugkeer naar hoofdprogramma
+	A	9	028B	A9	LDA #; subroutine <b>CLB2</b>
+	0	0	028C	00	00 → accu
+	8	5	028D	85	STAZ
+	0	3	028E	03	00 → B20
+	8	5	028F	85	STAZ
+	0	4	0290	04	00 → B21
+	8	5	0291	85	STAZ
+	0	5	0292	05	00 → B22
+	6	0	0293	60	RTS terugkeer naar hoofdprogramma
+	A	9	0294	A9	LDA #; subroutine <b>CLDISP</b>
+	0	0	0295	00	00 → accu

+	8	5	0296	85	STAZ
+	F	9	0297	F9	00 → INH (adres 00F9)
+	8	5	0298	85	STAZ
+	F	A	0299	FA	00 → POINTL (adres 00FA)
+	8	5	029A	85	STAZ
+	F	B	029B	FB	00 → POINTH (adres 00FB)
+	6	0	029C	60	RTS terugkeer naar hoofdprogramma
+	A	5	029D	A5	LDAZ; subroutine <b>STO2</b>
+	F	9	029E	F9	INH (adres 00F9) → accu
+	8	5	029F	85	STAZ
+	0	3	02A0	03	accu (= INH) → B20 (adres 0003)
+	A	5	02A1	A5	LDAZ
+	F	A	02A2	FA	POINTL (adres 00FA) → accu
+	8	5	02A3	85	STAZ
+	0	4	02A4	04	accu (= POINTL) → B21 (adres 0004)
+	A	5	02A5	A5	LDAZ
+	F	B	02A6	FB	POINTH (adres 00FB) → accu
+	8	5	02A7	85	STAZ
+	0	5	02A8	05	accu (= POINTH) → B22 (adres 0005)
+	6	0	02A9	60	RTS terugkeer naar hoofdprogramma
+	A	5	02AA	A5	LDAZ; subroutine <b>STO1</b>
+	F	9	02AB	F9	INH (adres 00F9) → accu
+	8	5	02AC	85	STAZ
+	0	0	02AD	00	accu (= INH) → B10 (adres 0000)
+	A	5	02AE	A5	LDAZ
+	F	A	02AF	FA	POINTL (adres 00FA) → accu
+	8	5	02B0	85	STAZ
+	0	1	02B1	01	accu (= POINTL) → B11 (adres 0001)
+	A	5	02B2	A5	LDAZ
+	F	B	02B3	FB	POINTH (adres 00FB) → accu
+	8	5	02B4	85	STAZ
+	0	2	02B5	02	accu (= POINTH) → B12 (adres 0002)
+	6	0	02B6	60	RTS terugkeer naar hoofdprogramma
+	A	5	02B7	A5	LDAZ; subroutine <b>RESDIS</b>
+	0	6	02B8	06	R0 (adres 0006) → accu
+	8	5	02B9	85	STAZ
+	F	9	02BA	F9	accu (= R0) → INH (adres 00F9)
+	A	5	02BB	A5	LDAZ
+	0	7	02BC	07	R1 (adres 0007) → accu
+	8	5	02BD	85	STAZ
+	F	A	02BE	FA	accu (= R1) → POINTL (adres 00FA)
+	A	5	02BF	A5	LDAZ
+	0	8	02C0	08	R2 (adres 0008) → accu
+	8	5	02C1	85	STAZ
+	F	B	02C2	FB	accu (= R2) → POINTH (adres 00FB)
+	6	0	02C3	60	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 programma-deel 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 bytes-begroting 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 economisch 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 standaard-uitvoering ten dienste staat voor eigen programma's beslaat de adressen 0000 . . . 03FF. Dat zijn vier pagina's, elk van 256 bytes:

pagina 00 : 0000 . . . 00FF

pagina 01 : 0100 . . . 01FF

pagina 02 : 0200 . . . 02FF

pagina 03 : 0300 . . . 03FF

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 00E1 . . . 00FF. Een paar van die geheugenplaatsen kennen we al, zoals 00F9 (label INH), 00FA (POINTL) en 00FB (POINTH). En verder de plaatsen 00EF . . . 00F5, 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 zelf te reserveren, indien men gebruik maakt van adressering op pagina nul. Pagina 01 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 1C00 . . . 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 (026F . . . 0281), 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 1A00 . . . 1A7F; dat is de eerste helft van pagina 1A.

Net als bij pagina nul van de standaard-RAM geldt ook hier een bespreking: 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 (1A80 . . . 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 spronginstructie 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 spronginstructie, 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				0000	00	
0	E	0	0	0E00	F0	F0 op adres 020F
1	2	1	A	121A	06	06 op adres 0213
2	5	3	1	2531	0A	0A op adres 0226
2	9	3	7	2937	0C	0C op adres 022A
5	2	4	B	524B	F7	F7 op adres 0253
7	2	6	F	726F	FB	FB op adres 0273
7	7	7	4	7774	FB	FB op adres 0278
7	C	7	8	7C78	F6	F6 op adres 027D

RST

N.B. Tussen twee offsetberekeningen kan men door het indrukken van een willekeurige kommando-toets het display 000000 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 spronginstructie, 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-komplementnotatie 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  $-40$ .

### 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	0	0	1A7A 00
	+	1	C	1A7B 1C
				1A7C XX
				1A7D XX
		0	0	1A7E 00
		1	C	1A7F 1C

Wat we in feite hebben gedaan is het laden van de geheugenplaats met adres 1A7E met 00 en van plaats 1A7F met 1C. Deze geheugenplaatsen (op pagina 1F, in de PIA-RAM) bevatten het effectieve adres (1C00, in het monitor-programma) waarnaar wordt gesprongen zodra een BRK optreedt, en dat is de software-versie van de hardware-instructie 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 effectieve adres 1C00. 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 toetsdetectie, 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-instructies, dan hoeven de plaatsen 1A7E en 1A7F niet te worden geladen. Ziet men af van het gebruik van de toets ST en van stapvoorstep-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. Via RST bereikt men adres 1C1D van het monitorprogramma direct, via een NMI of IRQ (=BRK) wordt 1C1D bereikt nadat een programmadeel (met startadres 1C00) 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 1C1D na een RST . Na de warming up en nadat het programma korrek is ingetoetst kunnen we het gaan uitvoeren:

```
AD 0 2 0 0 (ingeven startadres)
GO (programmastart)
2 4 5 6 002456
+ 000000
4 1 3 2 004132
DA (= ) 006588
AD (=  CLEAR ) 000000
1 9 8 5 3 1 198531
+ 000000
8 3 2 7 0 2 832702
DA 031233
AD 000000
```

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.

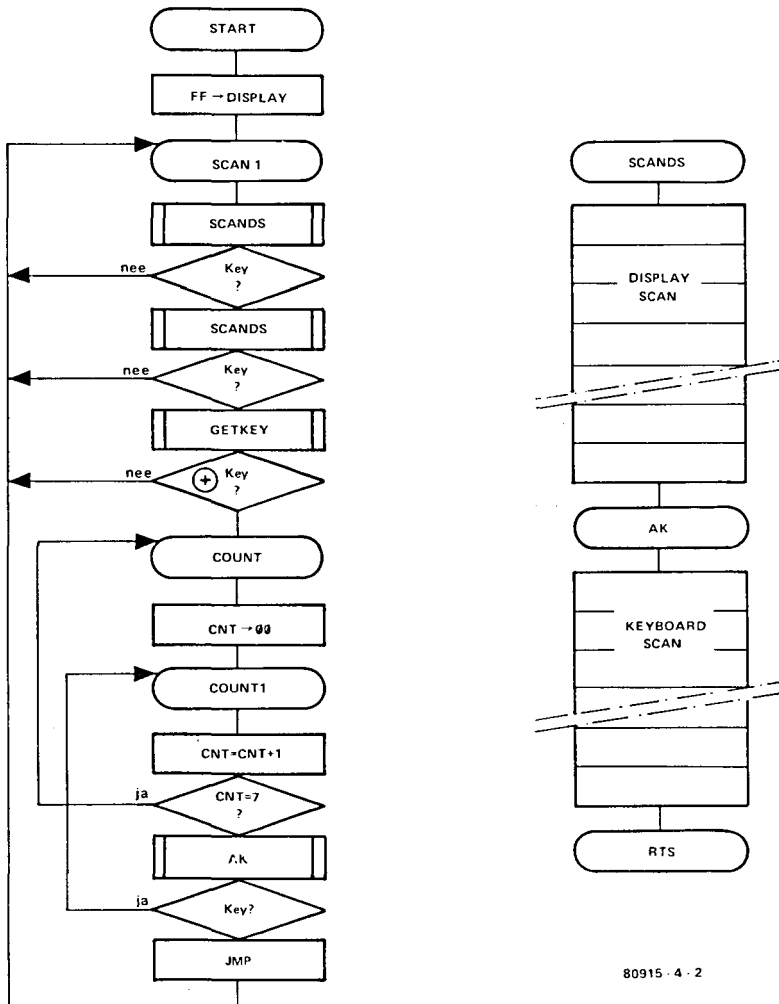
## De junior-computer als dobbelsteen

### *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 (01, 02, 03, 04, 05, of 06) verschijnt op de middelste twee displays (die normaal het rechter adresbyte ADL aangeven). De linker en rechter twee displays laten "FF" zien.

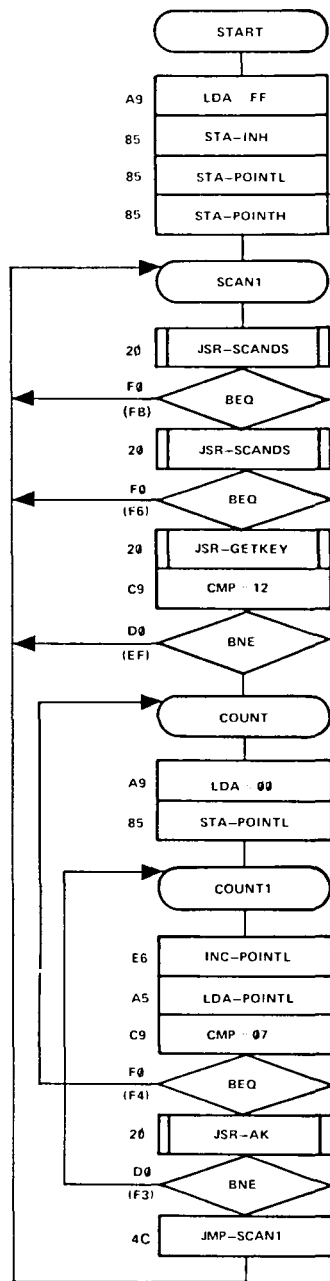
Het globale stroomdiagram van de software-dobbelsteen staat in figuur 2. Er is dankbaar gebruik gemaakt van de monitor-routines SCANDS, AK en GETKEY, die we in hoofdstuk 3 ook al een paar keer zijn tegengekomen. Wat gebeurt er allemaal in figuur 2? Begonnen wordt met het vullen van het display met FFFFFF. 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 wél + , dan begint het eigenlijke periodieke tellen (COUNT). Uit het gedetailleerde stroomdiagram van figuur 3 blijkt dat begonnen wordt met het via de accu laden van de display-buffer POINTL (voor de middelste



80915 - 4 - 3

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

twee displays) met 00. 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 07. De eerste keer is dat natuurlijk niet het geval; na zeven keer doorlopen van COUNT1 wel. Dreigt de tellerstand 07 te worden dan gaat het terug naar COUNT: de tellerstand wordt weer 00 gemaakt.

Bij een tellerstand ongelijk aan 07 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 07 in zijn werk gaat. Deze stand (CNT) gaat de accu in en wordt dan via een CMP-instructie vergeleken met 07.

Ook in figuur 3 stellen we vast dat het programmadeel COUNT1 uit een handvol instructies bestaat. Elke instructie vergt een paar mikrosekonden (zie het instructie-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 0200 genomen. Ook hier is het toetsengebeuren rijkelijk voorzien van commentaar. De programmastart gaat als volgt:

AD 0 2 0 0 GO

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

toetsen				adres	data	
RST		AD		xxxx	xx	
0	2	0	0	0200	xx	
DA		A	9	0200	A9	A9 LDA IMM <span style="border: 1px solid black; padding: 2px;">START</span>
+		F	F	0201	FF	FF → accu
+		8	5	0202	85	STA Z
+		F	9	0203	F9	FF → INH (00F9)
+		8	5	0204	85	STA Z
+		F	A	0205	FA	FF → POINTL (00FA)
+		8	5	0206	85	STA Z
+		F	B	0207	FB	FF → POINTH (00FB)
+		2	0	0208	20	JSR- <span style="border: 1px solid black; padding: 2px;">SCAN1</span>
+		8	E	0209	8E	ADL } van SCANDS (monitor) ADH } (adres 1D8E)
+		1	D	020A	1D	
+		F	0	020B	F0	BEQ
+		F	B	020C	FB	(offset); FB plaatsen terug is SCAN1
+		2	0	020D	20	JSR-
+		8	E	020E	8E	ADL } van SCANDS (monitor) ADH } (adres 1D8E)
+		1	D	020F	1D	
+		F	0	0210	F0	BEQ

+	F	6	0211	F6	F6 plaatsen terug is SCAN1	
+	2	0	0212	20	JSR-	
+	F	9	0213	F9	ADL } van GETKEY (monitor)	
+	1	D	0214	1D	ADH } (adres 1DF9)	
+	C	9	0215	C9	CMP IMM	
+	1	2	0216	12	met 12	
+	D	0	0217	D0	BNE	
+	E	F	0218	EF	EF plaatsen terug is SCAN1	
+	A	9	0219	A9	LDA IMM	
+	0	0	021A	00	00 → accu	
+	8	5	021B	85	STA Z	
+	F	A	021C	FA	00 → POINTL (00FA)	
+	E	6	021D	E6	INC Z	
+	F	A	021E	FA	POINTL + 1 → POINTL	
+	A	5	021F	A5	LDA Z	
+	F	A	0220	FA	POINTL → accu	
+	C	9	0221	C9	CMP IMM	
+	0	7	0222	07	met 07	
+	F	0	0223	F0	BEQ	
+	F	4	0224	F4	F4 plaatsen terug is COUNT	
+	2	0	0225	20	JSR-	
+	B	1	0226	B1	ADL } van AK (monitor)	
+	1	D	0227	1D	ADH } (adres 1DB1)	
+	D	0	0228	D0	BNE	
+	F	3	0229	F3	F3 plaatsen terug is COUNT1	
+	4	C	022A	4C	JMP-	
+	0	8	022B	08	ADL } van SCAN1	
+	0	2	022C	02	ADH }	
AD						
0	2	0	0	0200	A9	startadres
GO						programmastart
+				FF04	FF	de teerling is geworpen!
+				FF01	FF	
+				FF06	FF	
+				FF02	FF	

enzovoorts

**Figuur 4. Het toetsprogramma van de software-dobbelsteen.**

### Instruktielengte meten via software

Instructies 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							
		0	1	2	3	4	5	6	7
Linker hexadecimaal teken	0	BRK (1)	ORA (IND,X) (2)				ORA Z (2)	ASL Z (2)	0
	1	BPL (2)	ORA (IND),Y (2)				ORA Z,X (2)	ASL Z,X (2)	1
	2	JSR (3)	AND (IND,X) (2)				AND Z (2)	ROL Z (2)	2
	3	BMI (2)	AND (IND),Y (2)				AND Z,X (2)	ROL Z,X (2)	3
	4	RTI (1)	EOR (IND,X) (2)				EOR Z (2)	LSR Z (2)	4
	5	BVC (2)	EOR (IND),Y (2)				EOR Z,X (2)	LSR Z,X (2)	5
	6	RTS (1)	ADC (IND,X) (2)				ADC Z (2)	ROR Z (2)	6
	7	BVS (2)	ADC (IND),Y (2)				ADC Z,X (2)	ROR Z,X (2)	7
	8	BCC (2)	STA (IND,X) (2)				STY Z (2)	STA Z (2)	8
	9	BCC (2)	STA (IND),Y (2)				STY Z,X (2)	STA Z,X (2)	9
	A	LDY IMM (2)	LDA (IND,X) (2)	LDX IMM (2)			LDY Z (2)	LDA Z (2)	A
	B	BCS (2)	LDA (IND),Y (2)				LDY Z,X (2)	LDA Z,X (2)	B
	C	CPY IMM (2)	CMP (IND,X) (2)				CPY Z (2)	CMP Z (2)	C
	D	BNE (2)	CMP (IND),Y (2)				CMP Z,X (2)	DEC Z,X (2)	D
	E	CPX IMM (2)	SBC (IND,X) (2)				CPX Z (2)	SBC Z (2)	E
	F	BEQ (2)	SBC (IND),Y (2)				SBC Z,X (2)	INC Z,X (2)	F

**Figuur 5.** Deze opcodetabel is de compacte versie van de opcodetabel volgens Aanhangel 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 Aanhangel 1, achterin dit boek staat een overzicht van alle 256 mogelijke combinaties van twee hexadecimale tekens met – indien van toepassing – de bijbehorende mnemonics plus adresseringsmethode (dat laatste uitsluitend voor instructies met meerdere adresseringsmogelijkheden). Aanhangel 1 is nog eens compact 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-instructies, 74 opcodes van 2-byte-instructies, 48 opcodes van 3-byte-instructies 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-instructie of
- de opcode van 2-byte-instructie of
- de opcode van een 3-byte-instructie 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 “instruktietometer” 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 instruktie-lengte; X = 00 betekent geen opcode (“nul-byte-instructie”), X = 01 een 1-byte-instructie, X=02 een 2-byte-instructie en X=03 een 3-byte-instructie. 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 functie. Indien namelijk een ingetoetst byte niet tot de uitzonderingen

Rechter hexadecimaal teken

		Rechter hexadecimaal teken							
		B	9	A	B	C	D	E	F
Linker hexadecimaal teken	0	PHP (1)	ORA IMM (2)	ASL A (1)			ORA ABS (3)	ASL ABS (3)	0
	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 instructielengte 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 (0...F) bevat de opzoektabel de instructielengte van de bij de kolom horende instructies. Bevat dus 00, 01, 02 of 03.

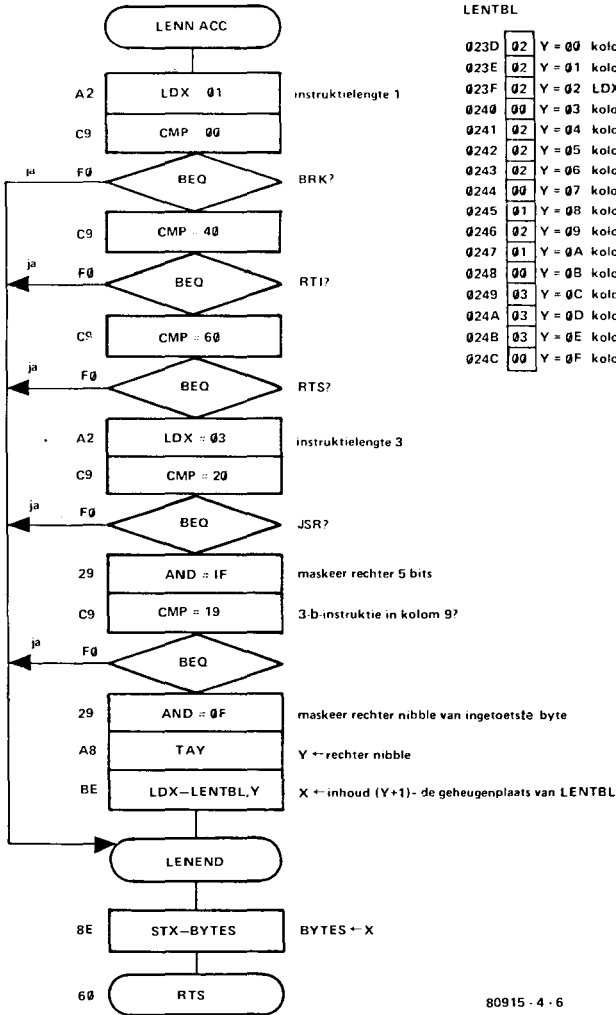
We horen in gedachten de lezer al protesteren: Schitterend, het klopt allemaal voor de 9 kolommen 1, 3, 5, 6, 7, 8, B, D en F, die elk uit 16 "0-", 1-, 2- of 3-byte-instructies bestaan (achter de mnemonics staat in figuur 5 de instructielengte tussen haakjes), maar de overige 7 kolommen bevatten instructies 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-instructies), 4 (los van de open plaatsen uitsluitend 2-byte-instructies), A (los van de open plaatsen uitsluitend 1-byte-instructies), C (los van de open plaatsen uitsluitend 3-byte-instructies) en kolom E, die op de lege plaats van 9E na uit 3-byte-instructies bestaat. De lege plaatsen kunnen overigens wel *allemaal* apart worden bekeken; zie verderop in dit verhaal.

Blijven over de kolommen 0 en 9. Kolom 0 bevat voornamelijk 2-byte-instructies, met 1-byte-uitzonderingen BRK, RTI en RTS en de 3-byte-uitzondering JSR. Kolom 9 bevat zowel 2- als 3-byte-instructies.

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 instructielengten (dus voor Y=00 wordt



Figuur 6. De subroutine LENACC zorgt ervoor dat, nadat twee numerieke toetsen 0... F zijn ingedrukt en in de accu geplaatst, de bij de accu-inhoud horende opcode de passende instructielengte (01, 02 of 03) 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 instructielengte 00 toegekend.

X=02, evenals voor Y=09) en ervoor zorgen dat de minderheidsuitzonderingen van de kolommen 0 en 9 apart worden bekeken. Dat laatste komt erop neer dat de 1-byte-instructie BRK, RTI en RTS van kolom 0 en de 3-byte-instructies van kolom 0 (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 01. Waarom 01? Omdat we eerst de drie 1-byte-instructies van kolom 0 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 instructies, dan gaan we door naar LENEND:X=01 komt in BYTES te staan.

Nu maken we X gelijk aan 03 en filteren we de 3-byte-instructie JSR van kolom 0 uit met de instructie CMP# 20 plus een aansluitende BEQ. Mocht het soms om de opcode van JSR gaan dan gaat er regelrecht worden gesprongen naar LENEND en komt er 03 in BYTES te staan.

Vervolgens worden met een slimme test de bokken (3-byte-instructies) van de schapen (2-byte-instructies) van kolom 9 gescheiden. Daartoe wordt de accu, dus het ingetoetste byte geAND met 1F (het zogenaamde *maskeren*). Dat is een hele mondvul. Daarom nu eerst het hoe, dan het waarom. Even de AND-functie voor de geest halen:

$$1 \cap X = X \text{ en } 0 \cap X = 0.$$

Dat gebeurt bit voor bit met de accu-inhoud. Het resultaat komt vervolgens in de accu:

accu voor ANDen:XXXXXXXX

ANDen met 1F :00011111

accu na ANDen :000XXXXX,

waarbij X van alles kan zijn (don't care), als het maar 0 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-instructies elkaar afwisselen. Het linker nibble van een instructie van kolom 9 is even voor een 2-byte-instructie en oneven voor een 3-byte-instructie. Voor 3-byte-instructies 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 0 geworden. Met andere woorden: door vergelijking (CMP IMM) met 19 stellen we vast of het een 3-byte-instructie is of niet. Is dat het geval (testen met een aansluitende BEQ) dan komt de inhoud van X (03) in BYTES te staan (sprong naar LENEND).

Waarmee alle uitzonderingsgevallen zijn uitgefilterd. Rest nog het ANDen (maskeren) met 0F, 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-instructies van de kolommen 3, 7, B en F als zodanig (X=00) te identificeren; de overige 41 zijn onder tafel geveegd. Hoewel het niet zal worden gedaan in

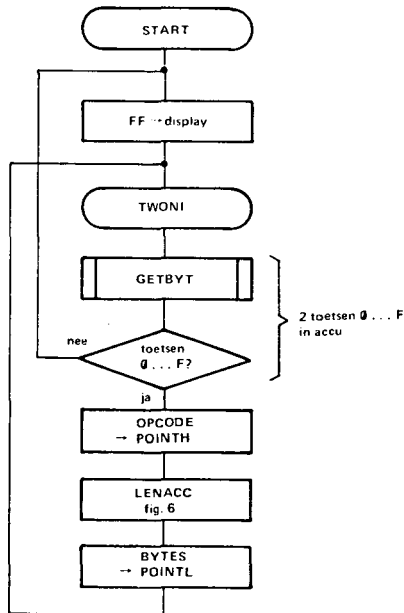
de uitwerking van figuur 6 zullen we nu aangeven hoe alle 105 open plaatsen van de opcodetabel van figuur 5 kunnen worden opgespoord. Dat gaat zo:

De subroutine LENACC begint nu met het laden van X met 00. Met 26 instructies CMP IMM . . . BEQ wordt gekeken of het ingetoetste byte hoort bij een lege plaats van kolom 0, 4, 9, A, C of E. Zoja, dan voert de betreffende BEQ naar LENEND. Zonee, dan wordt X=02 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=02 van 02 in 00. 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 complete programma van de "instructiemeter". Wat gebeurt er allemaal? We drukken eerst twee numerieke toetsen in (0 . . . 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 instructielengte weer met 01, 02 of 03. Of 00, 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



80915 - 4 - 7

Figuur 7. Het globale stroomdiagram van het instructielengte-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.

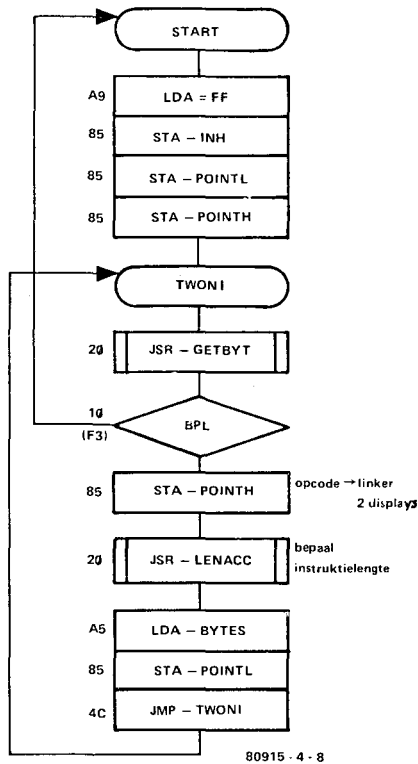
Nu hebben we geen boodschap aan kommandotoetsen. Slechts in de numerieke toetsen  $\emptyset \dots F$  zijn we geïnteresseerd. De routine GETBYT wordt dan ook gevolgd door een test waarin gekeken wordt of de twee ingedrukte toetsen numeriek zijn. Zolang dat niet het geval is is er sprake van een wachtklus: 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 instruktienlengte 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 instruktienlengte 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 instructies. Eén ding moet nog worden toegelicht: de instructie BPL na GETBYT. In de routine GETBYT wordt met een instructie  $CMP\# 1\emptyset$  onderscheid gemaakt tussen kommandotoetsen (toetswaarde  $\geq 1\emptyset$ ) en numerieke toetsen, met toetswaarden  $\emptyset\emptyset \dots \emptyset F$ . Voor toetswaarden kleiner dan  $1\emptyset$  is het gevolg van de vergelijking dat de N-vlag wordt geset, dus  $N=1$ . En de instructie BPL betekent voorwaardelijk springen, afhankelijk van de toestand van de N-vlag. Er wordt namelijk gesprongen bij  $N=\emptyset$ , dus als de toetswaarde  $1\emptyset$  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  $\emptyset 2\emptyset\emptyset$ . De hoofdroutine (figuur 8) loopt van adres  $\emptyset 2\emptyset\emptyset$  tot en met  $\emptyset 218$ , de subroutine LENACC (figuur 6) van  $\emptyset 219$  tot en met  $\emptyset 23C$  en de 16 daarop volgende geheugenplaatsen  $\emptyset 23D$  tot en met  $\emptyset 24C$  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  $\emptyset\emptyset F6$ ) zich op pagina nul bevindt. Dat komt omdat de monitor ook een instruktienlengte-routine bevat (OPLN; startadres  $1E5C$ ) die BYTES (adres  $\emptyset\emptyset F6$ , een van de RAM-opbergplaatsen  $\emptyset\emptyset E1 \dots \emptyset\emptyset FF$  op pagina nul) als bestemming heeft voor de instruktienlengte.

*Nu een opgave voor u:* Er is al vermeld dat lege plaatsen van de opcode-tabel van figuur 5 niet allemaal worden herkend als nul-byte-instructie. Er is ook aangegeven hoe dat euvel kan worden verholpen. Het is een goede test voor uw programmeerkennis om deze aanwijzingen concreet te vertalen in instructies en in een aangepast toetsprogramma. De junior-computer vertelt u na het intoetsen van het aangepaste toetsprogramma of u het goed heeft gedaan.



Figuur 8. Het gedetailleerde stroomdiagram van het instruktie lengte-programma.

toetsen		adres	data	comments
RTS	AD		xxxx xx	
0	2	0 0	0200 xx	startadres van instruktie meter
DA	A	9	0200 A9	LDA IMM <span style="border: 1px solid black; padding: 2px;">START</span>
+	F	F	0201 FF	FF → accu
+	8	5	0202 85	STAZ
+	F	9	0203 F9	accu → INH (adres 00F9)
+	8	5	0204 85	STAZ
+	F	A	0205 FA	accu → POINTL (adres 00FA)
+	8	5	0206 85	STAZ
+	F	B	0207 FB	accu → POINTH (adres 00FB)
+	2	0	0208 20	JSR- <span style="border: 1px solid black; padding: 2px;">TWO NI</span>
+	6	F	0209 6F	ADL } van GETBYT (monitor; ADH } adres 1D6F)
+	1	D	020A 1D	
+	1	0	020B 10	BPL; 2 toetsen 0... F ingetoetst?

+	F	3	020C	F3	zo nee, dan F3 plaatsen terug (= START)
+	8	5	020D	85	STA Z (byte in accu)
+	F	B	020E	FB	accu (= op code) → POINTH (adres 00FB)
+	2	0	020F	20	JSR-
+	1	9	0210	19	ADL } van LENACC ADH }
+	0	2	0211	02	
+	A	5	0212	A5	LDA Z
+	F	6	0213	F6	BYTES (adres 00F6) → accu
+	8	5	0214	85	STA Z
+	F	A	0215	FA	accu → POINTL (adres 00FA)
+	4	C	0216	4C	JMP ABS
+	0	8	0217	08	ADL } van sprongadres ADH } (TWOINI)
+	0	2	0218	02	
+	A	2	0219	A2	LDX IMM; subroutine <span style="border: 1px solid black; padding: 2px;">LENACC</span>
+	0	1	021A	01	01 → X; instruktie lengte 1
+	C	9	021B	C9	CMP IMM
+	0	0	021C	00	met 00
+	F	0	021D	F0	BEQ BRK?
+	1	A	021E	1A	1A plaatsen verder is LENEND
+	C	9	021F	C9	CMP IMM
+	4	0	0220	40	met 40
+	F	0	0221	F0	BEQ RTI?
+	1	4	0222	14	14 plaatsen verder is LENEND
+	C	9	0223	C9	CMP IMM
+	6	0	0224	60	met 60
+	F	0	0225	F0	BEQ RTS?
+	1	2	0226	12	12 plaatsen verder is LENEND
+	A	2	0227	A2	LDX IMM
+	0	3	0228	03	03 → X; instruktie lengte 3
+	C	9	0229	C9	CMP IMM
+	2	0	022A	20	met 20
+	F	0	022B	F0	BEQ JSR?
+	0	C	022C	0C	0C plaatsen verder is LENEND
+	2	9	022D	29	AND IMM
+	1	F	022E	1F	met 1F, maskeer rechter 5 bits
+	C	9	022F	C9	CMP IMM
+	1	9	0230	19	met 19
+	F	0	0231	F0	BEQ; 3-6-instructie in kolom 9?
+	0	6	0232	06	06 plaatsen verder is LENEND
+	2	9	0233	29	AND IMM

+		0	F	0234	0F	met 0F; maskeer rechter nibble
+		A	8	0235	A8	TAY; rechter nibble → Y
+		B	E	0236	BE	LDX ABS,Y; (Y + 1) de plaats van LENTBL
+		3	D	0237	3D	ADL } van LENTBL
+		0	2	0238	02	ADH } (look up table)
+		8	E	0239	8E	STX ABS <span style="border: 1px solid black; padding: 2px;">LENEND</span>
+		F	6	023A	F6	ADL BYTES
+		0	0	023B	00	ADH BYTES
+		6	0	023C	60	RTS terugkeer naar hoofdprogramma
+		0	2	023D	02	kolom 0; Y = 00 <span style="border: 1px solid black; padding: 2px;">LENTBL</span>
+		0	2	023E	02	kolom 1; Y = 01
+		0	2	023F	02	kolom 2; Y = 02
+		0	0	0240	00	kolom 3; Y = 03
+		0	2	0241	02	kolom 4; Y = 04
+		0	2	0242	02	kolom 5; Y = 05
+		0	2	0243	02	kolom 6; Y = 06
+		0	0	0244	00	kolom 7; Y = 07
+		0	1	0245	01	kolom 8; Y = 08
+		0	2	0246	02	kolom 9; Y = 09
+		0	1	0247	01	kolom A; Y = 0A
+		0	0	0248	00	kolom B; Y = 0B
+		0	3	0249	03	kolom C; Y = 0C
+		0	3	024A	03	kolom D; Y = 0D
+		0	3	024B	03	kolom E; Y = 0E
+		0	0	024C	00	kolom F; Y = 0F
AD						
0	2	0	0			startadres
GO						start programma
		A	9	A902	FF	
		0	3	0300	FF	
		D	2	D202	FF	
		9	E	9E03	FF	
		D	5	D502	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 00-FF. Ook de niet gebruikte hexadecimale cijferkombinaties zijn opgenomen.

00	BRK	08	JSR	40	RTI	60	RTS
01	ORA (IND,X)	21	AND (IND,X)	41	EOB (IND,X)	61	ADC (IND,X)
02	-	22	-	42	-	62	-
03	-	23	-	43	-	63	-
04	-	24	BIT Z	44	-	64	-
05	ORA Z	25	AND Z	45	EOB Z	65	ADC Z
06	ASL Z	26	ROL Z	46	LSR Z	66	ROR Z
07	-	27	-	47	-	67	-
08	PHP	28	PLP	48	PHA	68	PLA
09	ORA IMM	29	AND IMM	49	EOB IMM	69	ADC IMM
0A	ASL A	2A	ROL A	4A	LSR A	6A	ROR A
0B	-	2B	-	4B	-	6B	-
0C	-	2C	BIT ABS	4C	JMP ABS	6C	JMP IND
0D	ORA ABS	2D	AND ABS	4D	EOB ABS	6D	ADC ABS
0E	ASL ABS	2E	ROL ABS	4E	LSR ABS	6E	ROR ABS
0F	-	2F	-	4F	-	6F	-
10	BPL	30	BMI	50	BVC	70	BVS
11	ORA (IND),Y	31	AND (IND),Y	51	EOB (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	EOB 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	EOB 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	EOB 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	-

$\times C - \times E = ABS$   
 $\times 4 - \times 6 = Z$

80	-	A0	LDY IMM	C0	CPY IMM	E0	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	-
90	BCC	B0	BCS	D0	BNE	F0	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.** Direct na de opcode volgt de mnemonics van de instructie. Deze bestaat uit drie hoofdletters. Indien er meerdere adresseringsmogelijkheden zijn voor een bepaalde instructie volgt nog een aanduiding van de relevante adresseringsmethode. 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 indirecte adressering (indexed indirect addressing)
- (IND),Y Indirecte Y-geïndexeerde adressering (indirect indexed addressing)
- Z,X X-geïndexeerde adresseren op pagina nul (zero page indexed,X addressing)
- Z,Y Y-geïndexeerde 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 Indirecte adressering (indirect addressing)

## 2 Instructie-overzicht

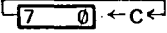
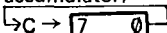
De 56 instructies van de 6502-microprocessor in alfabetische volgorde. Voor een groot aantal instructies zijn meerdere adresseringsmethoden mogelijk. Dit levert in totaal 151 verschillende opcodes op.

mnemonics + omschrijving	adresseringsmetode(n)	hex opcode	aantal kloppulsen (N)	aantal bytes	beïnvloede vlag(gen)
<b>ADC</b> Add memory to accumulator with carry $A + M + C \rightarrow A$ (1) (4)	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
<b>AND</b> "AND" memory with accumulator $A \wedge M \rightarrow A$ (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 -
<b>ASL</b> Shift left one bit (accu or memory) $C \leftarrow \boxed{7} \leftarrow 0$	ABS Z A Z,X ABS,X	0E 06 0A 16 1E	6 5 2 6 7	3 2 1 2 3	N ----- ZC
<b>BCC</b> Branch on carry clear (2) Branch on $C = 0$	REL	90	2	2	-----
<b>BCS</b> Branch on carry set (2) Branch on $C = 1$	REL	B0	2	2	-----
<b>BEQ</b> Branch on result zero (2) Branch on $Z = 1$	REL	F0	2	2	-----
<b>BIT</b> Test bits in memory: $A \wedge M$ $M_7 \rightarrow N; M_6 \rightarrow V$	ABS Z	2C 24	4 3	3 2	$M_7M_6$ ---- Z -
<b>BMI</b> Branch on result minus (2) Branch on $N = 1$	REL	30	2	2	-----

mnemonics + omschrijving	adresseringsmetode(n)	hex opcode	aantal klokpulsen (N)	aantal bytes	beïnvloede vlag(gen)
<b>BNE</b> Branch on result not zero (2) Branch on Z = 0	REL	D0	2	2	-----
<b>BPL</b> Branch on result plus (2) Branch on N = 0	REL	10	2	2	-----
<b>BRK</b> Force break forced interrupt	IMP	00	7	1	---1-1--- B  I
<b>BVC</b> Branch on overflow clear (2) Branch on V = 0	REL	50	2	2	-----
<b>BVS</b> Branch on overflow set (2) Branch on V = 1	REL	70	2	2	-----
<b>CLC</b> Clear carry flag 0 → C	IMP	18	2	1	-----0 C
<b>CLD</b> Clear decimal mode; 0 → D	IMP	D8	2	1	----0--- D
<b>CLI</b> Clear interrupt flag; 0 → I	IMP	58	2	1	-----0-- I
<b>CLV</b> Clear overflow flag; 0 → V	IMP	B8	2	1	0----- V
<b>CMP</b> 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
<b>CPX</b> Compare memory and index X X-M	IMM ABS Z	E0 EC E4	2 4 3	2 3 2	N-----ZC
<b>CPY</b> Compare memory and index Y M-Y	IMM ABS Z	C0 CC C4	2 4 3	2 3 2	N-----ZC

mneemonics + omschrijving	adresserings- methode(n)	hex opcode	aantal klokpulsen (N)	aantal bytes	beïnvloede vlag(gen)
<b>DEC</b> 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 -
<b>DEX</b> Decrement index X by one X-1 → X	IMP	CA	2	1	N - - - - - Z -
<b>DEY</b> Decrement index Y by one Y-1 → Y	IMP	88	2	1	N - - - - - Z -
<b>EOR</b> "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 -
<b>INC</b> 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 -
<b>INX</b> Increment index X by one X + 1 → X	IMP	E8	2	1	N - - - - - Z -
<b>INY</b> Increment index Y by one Y + 1 → Y	IMP	C8	2	1	N - - - - - Z -
<b>JMP</b> Jump to new location (PC + 1) → PCL (PC + 2) → PCH	ABS IND	4C 6C	3 5	3 3	- - - - -
<b>JSR</b> Jump to new location saving return address PC + 2 ↓ (PC + 1) → PCL (PC + 2) → PCH	ABS	20	6	3	- - - - -

mnemonics + omschrijving	adresserings- methode(n)	hex opcode	aantal klokpulsen (N)	aantal bytes	beïnvloede vlag(gen)
<b>LDA</b> 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 -
<b>LDX</b> 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 -
<b>LDY</b> 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 -
<b>LSR</b> Shift right one bit (memory or accumulator) 0 → <span style="border: 1px solid black; padding: 0 2px;">7 0</span> → 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
<b>NOP</b> No operation	IMP	EA	2	1	- - - - -
<b>ORA</b> "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 -
<b>PHA</b> Push accumulator on stack A↓	IMP	48	3	1	- - - - -
<b>PHP</b> Push processor status on stack; P↓	IMP	08	3	1	- - - - -
<b>PLA</b> 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)
<b>PLP</b> Pull processor status from stack; P↑	IMP	28	4	1	(hersteld)
<b>ROL</b> 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
<b>ROR</b> 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
<b>RTI</b> Return from interrupt PC↑; P↑	IMP	40	6	1	(hersteld)
<b>RTS</b> Return from subroutine PC↑; PC + 1 → PC	IMP	60	6	1	-----
<b>SBC</b> Subtract memory from accumulator with borrow (3) A-M-C̄ → 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
<b>SEC</b> Set carry flag 1 → C	IMP	38	2	1	-----1
<b>SED</b> Set decimal mode	IMP	F8	2	1	----1--- D
<b>SEI</b> Set interrupt disable; 1 → I	IMP	78	2	1	----1-- I
<b>STA</b> 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-methode(n)	hex opcode	aantal klokpulsen (N)	aantal bytes	beïnvloede vlag(gen)
<b>STX</b> Store index X in memory; X → M	ABS Z Z,Y	8E 86 96	4 3 4	3 2 2	-----
<b>STY</b> Store index Y in memory; Y → M	ABS Z Z,X	8C 84 94	4 3 4	3 2 2	-----
<b>TAX</b> Transfer accumulator to index X A → X	IMP	AA	2	1	N-----Z-
<b>TAY</b> Transfer accumulator to index Y A → Y	IMP	A8	2	1	N-----Z-
<b>TSX</b> Transfer stack pointer to index X S → X	IMP	BA	2	1	N-----Z-
<b>TXA</b> Transfer index X to accumulator X → A	IMP	8A	2	1	N-----Z-
<b>TXS</b> Transfer index X to stack pointer X → S	IMP	9A	2	1	N-----Z-
<b>TYA</b> 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 ( $\bar{C}$ )
- (4) Bij decimaal rekenen is de toestand van Z niet van toepassing; de accumulator moet op een andere manier worden getest op het resultaat nul

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 indirecte 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 het monitorprogramma

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D000	85	F3	68	85	F1	68	85	EF	85	FA	68	85	F0	85	FB	84
D010	F4	86	F5	BA	86	D2	A2	01	86	FF	4C	33	FC	A9	1E	8D
D020	83	FA	A9	04	85	F1	A9	03	85	FE	85	F6	A2	FF	9A	86
D030	F2	D8	78	20	88	ED	D0	FB	20	88	ED	F0	FB	20	88	ED
D040	F0	F6	20	F9	ED	C9	13	D0	13	A6	F2	9A	A5	FB	48	A5
D050	FA	48	A5	F1	48	A6	F5	A4	F4	A5	F3	40	C9	10	D0	06
D060	A9	03	85	FF	D0	14	C9	11	D0	06	A9	00	85	FF	F0	0A
D070	C9	12	D0	09	E6	FA	D0	02	E6	FB	4C	33	FC	C9	14	D0
IC80	0B	A5	EF	85	FA	A5	F0	85	FB	4C	7A	1C	C9	15	10	EA
IC90	85	E1	A4	FF	D0	0D	B1	FA	0A	0A	0A	0A	05	E1	91	FA
ICA0	4C	7A	1C	A2	04	06	FA	26	FB	CA	D0	F9	A5	FA	05	E1
ICB0	85	FA	4C	7A	1C	20	D3	1E	A4	E3	A6	E2	E8	D0	01	C8
ICC0	86	E8	84	E9	A9	77	A0	00	91	E6	20	4D	1D	C9	14	D0
ICD0	2A	20	6F	1D	10	F7	85	FB	20	6F	1D	10	F0	85	FA	20
ICE0	D3	1E	A0	00	B1	E6	C5	FB	D0	07	C8	B1	E6	C5	FA	F0
ICF0	D9	20	5C	1E	20	F8	1E	30	E9	10	3E	C9	10	D0	0A	20
ID00	20	1E	10	C9	20	47	1E	F0	C1	C9	13	D0	14	20	20	1E
ID10	10	BB	20	5C	1E	20	F8	1E	A5	FD	85	F6	20	47	1E	F0
ID20	A9	C9	12	D0	07	20	F8	1E	30	A0	10	0D	C9	11	D0	09
ID30	20	83	1E	20	EA	1E	4C	CA	1C	A9	EE	85	FB	85	FA	85
ID40	F9	A9	03	85	F6	20	8E	1D	D0	FB	4C	CA	1C	A2	02	A0
ID50	00	B1	E6	95	F9	C8	CA	10	F8	20	5C	1E	20	8E	1D	D0
ID60	FB	20	8E	1D	F0	FB	20	8E	1D	F0	F6	20	F9	1D	60	20
ID70	5C	1D	C9	10	10	11	0A	0A	0A	0A	85	FE	20	5C	1D	C9
ID80	10	10	04	05	FE	A2	FF	60	A0	00	B1	FA	85	F9	A9	7F
SCANPS	8D	81	1A	A2	08	A4	F6	A5	FB	20	CC	1D	88	F0	0D	A5
IDA0	FA	20	CC	1D	88	F0	05	A5	F9	20	CC	1D	A9	00	8D	81
IDB0	1A	A0	03	A2	00	A9	FF	8E	B2	1A	E8	E8	2D	80	1A	88
AK	D0	F5	A0	06	8C	82	1A	09	80	49	FF	60	48	84	FC	4A
IDD0	4A	4A	4A	20	DF	1D	68	29	0F	20	DF	1D	A4	FC	60	A8
IDE0	B9	0F	1F	8D	80	1A	8E	82	1A	A0	7F	88	10	FD	8C	80
IDF0	1A	A0	06	8C	82	1A	E8	E8	60	A2	21	A0	01	20	B5	1D
GETKEY	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

0 1 2 3 4 5 6 7 8 9 A B C D E F

GEWY	1E60:	A0	01/C9	00/F0	1A/C9	40/F0	16/C9	60/F0	12/A0	03/				
	1E70:	C9	20/F0	0C/29	1F/C9	19/F0	06/29	0F/AA	BC	1F 1F/				
	1E80:	84	F6/60	A5	E6/85	EA/A5	E7/85	EB/A4	F6/B1	EA/A0				
	1E90:	00/91	EA/E6	EA/D0	02/E6	EB/A5	EA/C5	E8/D0	EC/A5					
	1EA0:	EB/C5	E9/D0	E6/60	A5	E8/85	EA/A5	E9/85	EB/A0	00/				
	1EB0:	B1	EA/A4	F6/91	EA/A5	EA/C5	E6/D0	06/A5	EB/C5	E7/				
	1EC0:	F0	10/38	A5	EA/E9	01/85	EA/A5	EB/E9	00/85	EB/4C				
	1ED0:	AE	1E/60	A5	E2/85	E6/A5	E3/85	E7/60	18/A5	E8/65				
	1EE0:	F6/85	E8/A5	E9/69	00/85	E9/60	38/A5	E8/E5	F6/85					
	1EF0:	E8/A5	E9/E9	00/85	E9/60	18/A5	E6/65	F6/85	E6/A5					
	1F00:	E7/69	00/85	E7/38	A5	E6/E5	E8/A5	E7/E5	E9/60	40				
	1F10:	79	24/30	19	12/02	78	00/10	08	03/46	21	06	0E	02	
	1F20:	02/02	01	02/02	02	01/01	02	01/01	03	03	03	03	6C	
	1F30:	7A	1A/6C	7E	1A/B1	E6/A0	FF/C4	EE/F0	0D/D1	EC/D0				
	1F40:	0A/88	B1	EC/AA	88/B1	EC/A0	01/60	88/88	88/D0	E9/				
	1F50:	60/38	A5	E4/E9	FF/85	EC/A5	E5/E9	00/85	ED/A9	FF/				
	1F60:	85	EE/20	D3	1E/20	5C	1E/A0	00/B1	E6/C9	FF/D0	1D/			
	1F70:	C8	B1/E6	A4	EE/91	EC/88	A5	E7/91	EC/88	A5	E6/91			
	1F80:	EC/88	84	EE/20	83	1E/20	EA	1E/AC	65	1E/20	F8	1E/		
	1F90:	30	D3/20	D3	1E/20	5C	1E/A0	00/B1	E6/C9	4C/F0	16/			
	1FA0:	C9	20/F0	12/29	1F/C9	10/F0	1A/20	F8	1E/30	E6/A9				
	1FB0:	03/85	F6/AC	33	1C/C8	20	35	1F/F0	EE/91	E6/8A	C8/			
	1FC0:	91	E6/D0	E6/C8	20	35	1F/F0	E0/38	E5	E6/38	E9	02/		
	1FD0:	91	E6/4C	AA	1F/D8	A9	00/85	FB/85	FA/85	F9/20	6F	EA	7	
	1FE0:	1D/10	F2/85	FB	20	6F	1D/10	EB/85	FA/18	A5	FA	E5		
	1FF0:	FB/85	F9/C6	F9	4C	DE	1F/FF	FF	2F	FF	1D	FC	32	FF

NR2-vector      RESET vector

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 0) moet komen. Boven elke kolom ziet u het kolomnummer (0 ... F) staan.

- 0: 0, 8, 10, 18, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108, 110, 112, 114, 116, 118, 120, 122, 124, 126, 128, 130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154, 156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 176, 178, 180, 182, 184, 186, 188, 190, 192, 194, 196, 198, 200, 202, 204, 206, 208, 210, 212, 214, 216, 218, 220, 222, 224, 226, 228, 230, 232, 234, 236, 238, 240, 242, 244, 246, 248, 250, 252, 254, 256, 258, 260, 262, 264, 266, 268, 270, 272, 274, 276, 278, 280, 282, 284, 286, 288, 290, 292, 294, 296, 298, 300, 302, 304, 306, 308, 310, 312, 314, 316, 318, 320, 322, 324, 326, 328, 330, 332, 334, 336, 338, 340, 342, 344, 346, 348, 350, 352, 354, 356, 358, 360, 362, 364, 366, 368, 370, 372, 374, 376, 378, 380, 382, 384, 386, 388, 390, 392, 394, 396, 398, 400, 402, 404, 406, 408, 410, 412, 414, 416, 418, 420, 422, 424, 426, 428, 430, 432, 434, 436, 438, 440, 442, 444, 446, 448, 450, 452, 454, 456, 458, 460, 462, 464, 466, 468, 470, 472, 474, 476, 478, 480, 482, 484, 486, 488, 490, 492, 494, 496, 498, 500, 502, 504, 506, 508, 510, 512, 514, 516, 518, 520, 522, 524, 526, 528, 530, 532, 534, 536, 538, 540, 542, 544, 546, 548, 550, 552, 554, 556, 558, 560, 562, 564, 566, 568, 570, 572, 574, 576, 578, 580, 582, 584, 586, 588, 590, 592, 594, 596, 598, 600, 602, 604, 606, 608, 610, 612, 614, 616, 618, 620, 622, 624, 626, 628, 630, 632, 634, 636, 638, 640, 642, 644, 646, 648, 650, 652, 654, 656, 658, 660, 662, 664, 666, 668, 670, 672, 674, 676, 678, 680, 682, 684, 686, 688, 690, 692, 694, 696, 698, 700, 702, 704, 706, 708, 710, 712, 714, 716, 718, 720, 722, 724, 726, 728, 730, 732, 734, 736, 738, 740, 742, 744, 746, 748, 750, 752, 754, 756, 758, 760, 762, 764, 766, 768, 770, 772, 774, 776, 778, 780, 782, 784, 786, 788, 790, 792, 794, 796, 798, 800, 802, 804, 806, 808, 810, 812, 814, 816, 818, 820, 822, 824, 826, 828, 830, 832, 834, 836, 838, 840, 842, 844, 846, 848, 850, 852, 854, 856, 858, 860, 862, 864, 866, 868, 870, 872, 874, 876, 878, 880, 882, 884, 886, 888, 890, 892, 894, 896, 898, 900, 902, 904, 906, 908, 910, 912, 914, 916, 918, 920, 922, 924, 926, 928, 930, 932, 934, 936, 938, 940, 942, 944, 946, 948, 950, 952, 954, 956, 958, 960, 962, 964, 966, 968, 970, 972, 974, 976, 978, 980, 982, 984, 986, 988, 990, 992, 994, 996, 998, 1000, 1002, 1004, 1006, 1008, 1010, 1012, 1014, 1016, 1018, 1020, 1022, 1024, 1026, 1028, 1030, 1032, 1034, 1036, 1038, 1040, 1042, 1044, 1046, 1048, 1050, 1052, 1054, 1056, 1058, 1060, 1062, 1064, 1066, 1068, 1070, 1072, 1074, 1076, 1078, 1080, 1082, 1084, 1086, 1088, 1090, 1092, 1094, 1096, 1098, 1100, 1102, 1104, 1106, 1108, 1110, 1112, 1114, 1116, 1118, 1120, 1122, 1124, 1126, 1128, 1130, 1132, 1134, 1136, 1138, 1140, 1142, 1144, 1146, 1148, 1150, 1152, 1154, 1156, 1158, 1160, 1162, 1164, 1166, 1168, 1170, 1172, 1174, 1176, 1178, 1180, 1182, 1184, 1186, 1188, 1190, 1192, 1194, 1196, 1198, 1200, 1202, 1204, 1206, 1208, 1210, 1212, 1214, 1216, 1218, 1220, 1222, 1224, 1226, 1228, 1230, 1232, 1234, 1236, 1238, 1240, 1242, 1244, 1246, 1248, 1250, 1252, 1254, 1256, 1258, 1260, 1262, 1264, 1266, 1268, 1270, 1272, 1274, 1276, 1278, 1280, 1282, 1284, 1286, 1288, 1290, 1292, 1294, 1296, 1298, 1300, 1302, 1304, 1306, 1308, 1310, 1312, 1314, 1316, 1318, 1320, 1322, 1324, 1326, 1328, 1330, 1332, 1334, 1336, 1338, 1340, 1342, 1344, 1346, 1348, 1350, 1352, 1354, 1356, 1358, 1360, 1362, 1364, 1366, 1368, 1370, 1372, 1374, 1376, 1378, 1380, 1382, 1384, 1386, 1388, 1390, 1392, 1394, 1396, 1398, 1400, 1402, 1404, 1406, 1408, 1410, 1412, 1414, 1416, 1418, 1420, 1422, 1424, 1426, 1428, 1430, 1432, 1434, 1436, 1438, 1440, 1442, 1444, 1446, 1448, 1450, 1452, 1454, 1456, 1458, 1460, 1462, 1464, 1466, 1468, 1470, 1472, 1474, 1476, 1478, 1480, 1482, 1484, 1486, 1488, 1490, 1492, 1494, 1496, 1498, 1500, 1502, 1504, 1506, 1508, 1510, 1512, 1514, 1516, 1518, 1520, 1522, 1524, 1526, 1528, 1530, 1532, 1534, 1536, 1538, 1540, 1542, 1544, 1546, 1548, 1550, 1552, 1554, 1556, 1558, 1560, 1562, 1564, 1566, 1568, 1570, 1572, 1574, 1576, 1578, 1580, 1582, 1584, 1586, 1588, 1590, 1592, 1594, 1596, 1598, 1600, 1602, 1604, 1606, 1608, 1610, 1612, 1614, 1616, 1618, 1620, 1622, 1624, 1626, 1628, 1630, 1632, 1634, 1636, 1638, 1640, 1642, 1644, 1646, 1648, 1650, 1652, 1654, 1656, 1658, 1660, 1662, 1664, 1666, 1668, 1670, 1672, 1674, 1676, 1678, 1680, 1682, 1684, 1686, 1688, 1690, 1692, 1694, 1696, 1698, 1700, 1702, 1704, 1706, 1708, 1710, 1712, 1714, 1716, 1718, 1720, 1722, 1724, 1726, 1728, 1730, 1732, 1734, 1736, 1738, 1740, 1742, 1744, 1746, 1748, 1750, 1752, 1754, 1756, 1758, 1760, 1762, 1764, 1766, 1768, 1770, 1772, 1774, 1776, 1778, 1780, 1782, 1784, 1786, 1788, 1790, 1792, 1794, 1796, 1798, 1800, 1802, 1804, 1806, 1808, 1810, 1812, 1814, 1816, 1818, 1820, 1822, 1824, 1826, 1828, 1830, 1832, 1834, 1836, 1838, 1840, 1842, 1844, 1846, 1848, 1850, 1852, 1854, 1856, 1858, 1860, 1862, 1864, 1866, 1868, 1870, 1872, 1874, 1876, 1878, 1880, 1882, 1884, 1886, 1888, 1890, 1892, 1894, 1896, 1898, 1900, 1902, 1904, 1906, 1908, 1910, 1912, 1914, 1916, 1918, 1920, 1922, 1924, 1926, 1928, 1930, 1932, 1934, 1936, 1938, 1940, 1942, 1944, 1946, 1948, 1950, 1952, 1954, 1956, 1958, 1960, 1962, 1964, 1966, 1968, 1970, 1972, 1974, 1976, 1978, 1980, 1982, 1984, 1986, 1988, 1990, 1992, 1994, 1996, 1998, 2000, 2002, 2004, 2006, 2008, 2010, 2012, 2014, 2016, 2018, 2020, 2022, 2024, 2026, 2028, 2030, 2032, 2034, 2036, 2038, 2040, 2042, 2044, 2046, 2048, 2050, 2052, 2054, 2056, 2058, 2060, 2062, 2064, 2066, 2068, 2070, 2072, 2074, 2076, 2078, 2080, 2082, 2084, 2086, 2088, 2090, 2092, 2094, 2096, 2098, 2100, 2102, 2104, 2106, 2108, 2110, 2112, 2114, 2116, 2118, 2120, 2122, 2124, 2126, 2128, 2130, 2132, 2134, 2136, 2138, 2140, 2142, 2144, 2146, 2148, 2150, 2152, 2154, 2156, 2158, 2160, 2162, 2164, 2166, 2168, 2170, 2172, 2174, 2176, 2178, 2180, 2182, 2184, 2186, 2188, 2190, 2192, 2194, 2196, 2198, 2200, 2202, 2204, 2206, 2208, 2210, 2212, 2214, 2216, 2218, 2220, 2222, 2224, 2226, 2228, 2230, 2232, 2234, 2236, 2238, 2240, 2242, 2244, 2246, 2248, 2250, 2252, 2254, 2256, 2258, 2260, 2262, 2264, 2266, 2268, 2270, 2272, 2274, 2276, 2278, 2280, 2282, 2284, 2286, 2288, 2290, 2292, 2294, 2296, 2298, 2300, 2302, 2304, 2306, 2308, 2310, 2312, 2314, 2316, 2318, 2320, 2322, 2324, 2326, 2328, 2330, 2332, 2334, 2336, 2338, 2340, 2342, 2344, 2346, 2348, 2350, 2352, 2354, 2356, 2358, 2360, 2362, 2364, 2366, 2368, 2370, 2372, 2374, 2376, 2378, 2380, 2382, 2384, 2386, 2388, 2390, 2392, 2394, 2396, 2398, 2400, 2402, 2404, 2406, 2408, 2410, 2412, 2414, 2416, 2418, 2420, 2422, 2424, 2426, 2428, 2430, 2432, 2434, 2436, 2438, 2440, 2442, 2444, 2446, 2448, 2450, 2452, 2454, 2456, 2458, 2460, 2462, 2464, 2466, 2468, 2470, 2472, 2474, 2476, 2478, 2480, 2482, 2484, 2486, 2488, 2490, 2492, 2494, 2496, 2498, 2500, 2502, 2504, 2506, 2508, 2510, 2512, 2514, 2516, 2518, 2520, 2522, 2524, 2526, 2528, 2530, 2532, 2534, 2536, 2538, 2540, 2542, 2544, 2546, 2548, 2550, 2552, 2554, 2556, 2558, 2560, 2562, 2564, 2566, 2568, 2570, 2572, 2574, 2576, 2578, 2580, 2582, 2584, 2586, 2588, 2590, 2592, 2594, 2596, 2598, 2600, 2602, 2604, 2606, 2608, 2610, 2612, 2614, 2616, 2618, 2620, 2622, 2624, 2626, 2628, 2630, 2632, 2634, 2636, 2638, 2640, 2642, 2644, 2646, 2648, 2650, 2652, 2654, 2656, 2658, 2660, 2662, 2664, 2666, 2668, 2670, 2672, 2674, 2676, 2678, 2680, 2682, 2684, 2686, 2688, 2690, 2692, 2694, 2696, 2698, 2700, 2702, 2704, 2706, 2708, 2710, 2712, 2714, 2716, 2718, 2720, 2722, 2724, 2726, 2728, 2730, 2732, 2734, 2736, 2738, 2740, 2742, 2744, 2746, 2748, 2750, 2752, 2754, 2756, 2758, 2760, 2762, 2764, 2766, 2768, 2770, 2772, 2774, 2776, 2778, 2780, 2782, 2784, 2786, 2788, 2790, 2792, 2794, 2796, 2798, 2800, 2802, 2804, 2806, 2808, 2810, 2812, 2814, 2816, 2818, 2820, 2822, 2824, 2826, 2828, 2830, 2832, 2834, 2836, 2838, 2840, 2842, 2844, 2846, 2848, 2850, 2852, 2854, 2856, 2858, 2860, 2862, 2864, 2866, 2868, 2870, 2872, 2874, 2876, 2878, 2880, 2882, 2884, 2886, 2888, 2890, 2892, 2894, 2896, 2898, 2900, 2902, 2904, 2906, 2908, 2910, 2912, 2914, 2916, 2918, 2920, 2922, 2924, 2926, 2928, 2930, 2932, 2934, 2936, 2938, 2940, 2942, 2944, 2946, 2948, 2950, 2952, 2954, 2956, 2958, 2960, 2962, 2964, 2966, 2968, 2970, 2972, 2974, 2976, 2978, 2980, 2982, 2984, 2986, 2988, 2990, 2992, 2994, 2996, 2998, 3000, 3002, 3004, 3006, 3008, 3010, 3012, 3014, 3016, 3018, 3020, 3022, 3024, 3026, 3028, 3030, 3032, 3034, 3036, 3038, 3040, 3042, 3044, 3046, 3048, 3050, 3052, 3054, 3056, 3058, 3060, 3062, 3064, 3066, 3068, 3070, 3072, 3074, 3076, 3078, 3080, 3082, 3084, 3086, 3088, 3090, 3092, 3094, 3096, 3098, 3100, 3102, 3104, 3106, 3108, 3110, 3112, 3114, 3116, 3118, 3120, 3122, 3124, 3126, 3128, 3130, 3132, 3134, 3136, 3138, 3140, 3142, 3144, 3146, 3148, 3150, 3152, 3154, 3156, 3158, 3160, 3162, 3164, 3166, 3168, 3170, 3172, 3174, 3176, 3178, 3180, 3182, 3184, 3186, 3188, 3190, 3192, 3194, 3196, 3198, 3200, 3202, 3204, 3206, 3208, 3210, 3212,









0 0 0 0 0 0 0  
0 0 0 0 0 0 0  
0 0 0 0 0 0 0  
0 0 0 0 0 0 0  
0 0 0 0 0 0 0

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

ISBN 90 70160 15 3