D. M. DE BOER

Programmeren, stap voor stap

Het toetsenbord van de KIM

Om de KIM te kunnen programmeren moeten we de mogelijkheid hebben om op **bepaalde** geheugenplaatsen **bepaalde** getallen te zetten. Elk getal heeft voor de microcomputer een gegeven betekenis. Door een juiste getallencombinatie in opeenvolgende geheugenplaatsen te zetten, kunnen we de microprocessor een bepaalde handeling laten verrichten. Om deze getallen in het geheugen te zetten bevindt zich op de KIM een toetsenbordje met cijfers (0 . . . F) en controletoetsen (AD, DA, PC, +, GO, ST en RS). Wanneer we een keer op AD drukken (adres) hebben we de mogelijkheid een adres (= een geheugenplaats) te selecteren. Dit adres is zichtbaar op de 4 linker displays. Op de rechter 2 displays staat de inhoud van dit adres. Wanneer we de inhoud van het adres willen wijzigen moeten we eerst op 'DA' (data) drukken. Vanaf dit moment worden ingedrukte cijfers op de geselecteerde geheugenplaats gezet. Op het display is steeds de inhoud van het geheugen te zien. Het is dus niet nodig om op '+' of een andere toets te drukken om de informatie 'in te clocken'. Elk ingedrukt cijfer komt automatisch in de geselecteerde geheugenplaats te staan. Bij het intypen van een programma zou het wat omslachtig zijn om steeds op AD te drukken, het adres in te voeren, op DA te drukken, de data in te voeren, weer op AD te drukken enz. Daarom is de '+'-toets aanwezig. Met deze toets kan het adres met 1 worden opgehoogd, zodat een programma makkelijk ingevoerd kan worden.

De 'ST'-toets, een gevaarlijke toets

De 'ST'-toets (STOP-toets) dient om een programma dat gestart is weer te stoppen. Op het moment dat de ST-toets wordt ingedrukt maakt de microprocessor de opdracht waar hij mee bezig was af, om vervolgens te stoppen met het programma. Op het display verschijnt nu het adres van de eerstvolgende uit te voeren instructie. Door op 'GO' te drukken wordt het programma weer vervolgd, alsof er niets aan de hand was. Maar waarom is deze toets dan gevaarlijk? Wel, met een druk op 'ST' wordt een interrupt gegenereerd, en de microprocessor zal naar het interruptprogramma springen. De plaats waar dit programma staat kunnen we zelf definiëren, zodat ook naar zelf geschreven programma's kan worden gesprongen. Wanneer dit adres niet gedefinieerd is, zal de microprocessor na een druk op 'ST' op een willekeurige plaats in het geheugen starten. Dat **kan** tot gevolg hebben dat het hele geheugen gewist wordt. Daarom: maak er een gewoonte van om direct na het inschakelen van de KIM de interrupt vector in te voeren (eerst op 'RS' drukken om het display te laten oplichten). Voor de stopfunctie (en de single step) voeren we in:

Adres	inhoud
17FA	00
17FB	1C

Wanneer dit gedaan is kunt u net zo vaak op 'ST' drukken als u wilt.

De toetsen 'GO', 'RS' en 'PC'

Er blijven nu nog 3 toetsen over die niet ter sprake zijn gekomen. De makkelijkste van de 3 is 'GO'. Met deze toets kunt u een zelf geschreven programma starten. Met 'GO' wordt de microprocessor gestart vanaf het adres dat op dat moment op het display staat. U moet dus altijd eerst het startadres invoeren. (Dus het adres waar de eerste instructie staat.) Met de 'RS' toets (reset) wordt onvoorwaardelijk naar het monitorprogramma gesprongen. (Het monitorprogramma zorgt dat het display gestuurd wordt, en decodeert de signalen van het toetsenbordje.) De functie van 'RS' lijkt een beetje op de functie van 'ST'. Er is echter een belangrijk verschil. Met 'ST' kan een programma onderbroken worden, op het display is te zien op welk adres de microprocessor was gebleven. Met 'RS' wordt een programma ook onderbroken, op het display verschijnt nu echter het laatste zichtbaar gemaakte adres (meestal het startadres). We kunnen het programma niet vervolgen, omdat we niet weten waar de microprocessor was gebleven. Bovendien zijn bepaalde gegevens uit de inwendige registers van de microprocessor niet bewaard gebleven. We kunnen nu het programma alleen weer vanaf het begin starten. De 'RS'-toets heeft echter enkele wezenlijke voordelen.

We kunnen b.v. het monitorprogramma starten als de interruptvector nog niet gedefinieerd is (b.v. als de KIM net is aangezet). Bij programmafouten kan het voorkomen dat een niet bestaande instructie gelezen wordt. Vaak wil de 'ST'-toets dan niet werken, omdat de microprocessor eerst de instructie, waar hij mee bezig is, wil afmaken. Met sommige niet bestaande instructies (b.v. 42) komt de microprocessor nooit klaar, en dus werkt 'ST' niet. Ook hier brengt de 'RS'-toets uitkomst. Bij de meer gevorderde programmeurs kan het voorkomen dat een andere interruptvector wordt ingevoerd, waardoor 'ST' een andere functie kan krijgen. Ook in dit geval zijn we aangewezen op 'RS'. Bovendien wordt bij een druk op 'RS' de stackpointer goed gezet en de inhoud van de richtingsregisters van de in/uitgangen wordt '00'. Om deze punten behoeven we ons nog niet druk te maken, voorlopig gebruiken we deze functies nog niet.

De 'PC'-toets werkt in combinatie met de 'ST' of de single step. (We gaan ervan uit dat de juiste interruptvector is ingevoerd, '00' op 17FA en '1C' op 17FB.) We hebben al gezegd dat bij een druk op 'ST' het adres op het display komt te staan van de instructie waar de microprocessor is gebleven. We kunnen nu andere adressen invoeren om bepaalde registers of geheugens te controleren. Wanneer we het oorspronkelijke adres terug willen hebben, drukken we gewoon op 'PC' (program counter), en het juiste adres verschijnt weer op het display. Soms is het makkelijk om snel een bepaald adres terug te kunnen roepen. In dat geval kunnen we ook gebruik maken van 'PC'. Wanneer we b.v. bij het drukken op 'PC' adres 0123 tevoorschijn willen toveren, moeten we dit als volgt invoeren:

adres	inhoud
00EF	23
00F0	01

Zolang we de 'ST' en de single step (SST) niet gebruiken, zal steeds dit adres verschijnen bij een druk op 'PC'.

Het SST-schuifje

Wanneer het SST-schuifje op 'on' staat, werken we met single step. Bij elke druk op 'GO' wordt nu maar één instructie uitgevoerd. Wel moet de interruptvector zijn ingevoerd (zie het stukje bij de 'ST'-toets). Na elke instructie kunnen

we controleren hoe de inhoud van de verschillende registers en geheugenplaatsen is veranderd. Met een druk op 'PC' verschijnt het adres van de volgende instructie weer op het display, en met een druk op 'GO' wordt ook deze instructie weer uitgevoerd. De single step dient om programma's te controleren, en indien het programma niet werkt, de fout te zoeken.

De eerste instructies

Als u het grote aantal instructies met de daarbij behorende adresseermogelijkheden bekijkt, zal het zeker in het begin zeer onoverzichtelijk lijken. Vaak begrijpt men niet goed welke adresseermogelijkheid gebruikt moet worden. Daarom beginnen we met een beperkt aantal instructies met alleen de 'absolute addressing'. De instructies zijn voorlopig alleen:

Instructie	Code
LDA	AD
STA	8D
JMP	4C

Met de eerste instructie heeft u de mogelijkheid om een bepaald getal van een willekeurige plaats uit het geheugen te halen en in register A te zetten. (Register A bevindt zich in de microprocessor.) De tweede instructie doet het tegenovergestelde, een getal dat in register A staat kan op een willekeurige plaats in het geheugen gezet worden. De derde instructie (JMP) is een onvoorwaardelijke sprongopdracht. Dit betekent dat de microprocessor de volgende instructie niet van de volgende geheugenplaats haalt, maar van het adres dat achter de JMP ('jump')-instructie staat. De microprocessor springt dus naar een ander deel van het geheugen.

Met deze 3 instructies kunnen we het eerste programmaatje maken. We willen een programma dat een getal, dat op adres 0000 staat, op adres 0001 zet. Het programma ziet er als volgt uit:

```
0200 AD 00 00     LDA, abs $0000
0203 8D 01 00     STA, abs $0001
0206 4C 22 1C     JMP, abs RST
```

Op adres 0200 zien we de instructie 'AD' welke staat voor 'LDA'. Vervolgens kunnen we ons afvragen: 'Welk getal moet in register A komen te staan?' Direct na het getal 'AD' (de op-code) volgt over twee geheugens (bytes) verdeeld, de operand. In dit geval is dit '0000'. De microprocessor zal dus een getal dat op adres '0000' staat in register 'A' zetten. Bij de tweede instructie gebeurt het omgekeerde, de inhoud van register A wordt nu weggezet in het geheugen. Welke geheugenplaats dit is, wordt weer aangegeven door de 2 volgende geheugenplaatsen. (Op de **eerste** plaats de laatste cijfers v.h. adres, en op de **tweede** plaats de eerste cijfers van het adres.) Met deze twee instructies hebben we dus al bereikt wat we wilden, het getal op adres 0000 is verplaatst naar adres 0001. De microprocessor weet dit echter niet en gaat stug door met de volgende instructie. Hier staat 'JMP'. De microprocessor zal nu naar adres 1C22 springen (dit adres neeft de naam RST gekregen). Hier begint het monitorprogramma, zodat het display oplicht, en we het toetsenbord weer kunnen gebruiken. Wanneer u het programma ingetypt hebt, en u start het programma, dan zal het net lijken of er niets gebeurt. Het programma is nl. zo kort (het wordt in 11 µs uitgevoerd) dat dit niet waarneembaar is. Als u echter de geheugenplaatsen 0000 en 0001 controleert, dan zal u zien dat de opdracht perfect is uitgevoerd.

Nog een eenvoudig voorbeeld

We gaan nog een eenvoudig voorbeeld bespreken; nu moeten de getallen op de adressen 0000 en 0001 verwisseld worden. Als we beginnen op dezelfde manier als in het vorige voorbeeld (LDA 0000, STA 0001) gaat de inhoud van geheugenplaats 0001 verloren. We moeten dus een tussenstap maken en eerst het getal op adres 0001 tijdelijk ergens anders plaatsen. Dit kan als volgt:

```
0200 AD 01 00     LDA, abs $0001
0203 8D 02 00     STA, abs $0002
0206 AD 00 00     LDA, abs $0000
0209 8D 01 00     STA, abs $0001
020C AD 02 00     LDA, abs $0002
020F 8D 00 00     STA, abs $0000
0212 4C 22 1C     JMP, abs RST
```

Voor dit korte stukje programma hebben we maar liefst 21 geheugenplaatsen nodig. Dit aantal kan drastisch beperkt worden door de 'zero-page addressing' toe te passen.

Zero page addressing

Het is u misschien opgevallen dat de geheugenplaatsen welke het programma gebruikt allemaal met '00' beginnen (0000, 0001 en 0002). We zeggen ook wel: al deze adressen liggen in pagina 0. Het programma dat we geschreven hebben loopt van adres 0200 . . . 0214. Deze adressen liggen dus in pagina 2. Voor adressen in pagina 0 bestaat bij de KIM een aparte adresseermogelijkheid, nl. de 'zero page addressing'. Deze adressering heeft hetzelfde effect als de absolute adressering, maar kan alleen toegepast worden bij adressen in pagina 0. We zullen nu hetzelfde programma schrijven, nu echter met gebruikmaking van de 'zero page addressing'.

```
0200 A5 01        LDA, Zpage $01
0202 85 02        STA, Zpage $02
0204 A5 00        LDA, Zpage $00
0206 85 01        STA, Zpage $01
0208 A5 02        LDA, Zpage $02
020A 85 00        STA, Zpage $00
020C 4C 22 1C     JMP, abs RST
```

We zien, dat we nu nog maar 15 geheugenplaatsen nodig hebben i.p.v. de 21 bij de absolute adressering in het vorige voorbeeld. Ook de codes voor de instructie zijn anders (A5 i.p.v. AD, 85 i.p.v. 8D), dit is logisch, de microprocessor moet immers weten welke adresseermethode is toegepast. Bij dit voorbeeld ziet u meteen dat het programma korter kan worden als de door het programma gebruikte geheugenplaatsen op pagina 0 liggen. (Deze korte methode had niet gekund als we b.v. de adressen 0300, 0301 en 0302 hadden gebruikt.) Het programma kan nog korter, want in de microprocessor zit niet alleen een register A, maar ook een register X. We gebruiken nu geheugenplaats 0002 om tijdelijk een getal in op te bergen. Wanneer we hiervoor register X gebruiken kunnen we weer een paar plaatsen winnen.

Register X

Register X is net als register A een geheugen in de microprocessor. Register X kan echter enige specifieke opdrachten uitvoeren, in combinatie met de 'indexed addressing'. Daarom zeggen we ook wel: Indexregister X. We komen hier nog op terug. Voorlopig 2 nieuwe instructies met de reeds besproken adresseermogelijkheden.

instructie	abs.	Zpage
LDX	AE	A6
STX	8E	86

Wanneer we nu weer hetzelfde voorbeeld maken (verwissel de getallen op de adressen 0000 en 0001) krijgen we:

```
0200 A6 01        LDX, Zpage $01
0202 A5 00        LDA, Zpage $00
0204 85 01        STA, Zpage $01
0206 86 00        STX, Zpage $00
0208 4C 22 1C     JMP, abs RST
```

En we zien dat nu nog maar 11 geheugenplaatsen nodig zijn, i.p.v. de 21 welke we in het begin nodig hadden. Ook de benodigde tijd is teruggebracht, wanneer we uitsluitend met register A en de 'absolute addressing' werken kost dit programma 27 µs. In het laatste voorbeeld is dit slechts 15 µs. Ogenschijnlijk niet ter zake doende verschillen, u moet echter bedenken dat bij langere programma's deze verschillen wel degelijk een rol gaan spelen. (Door de juiste adressering toe te passen kunnen er meer instructies, en dus langere en

betere programma's in dezelfde geheugenruimte).

Namen voor geheugenplaatsen

Vaak wordt aan een geheugenplaats een naam toegekend, omdat dan een programma vaak gemakkelijker te lezen is. In ons voorbeeld moesten 2 getallen verwisseld worden, nl. de getallen op adres 0000 en adres 0001. Als we deze geheugenplaatsen nu resp. 'GETAL1' en 'GETAL2' noemen, ziet het programma er overzichtelijker uit:

```
0200 A6 01   START  LDX, Zpage GETAL2
0202 A5 00          LDA, Zpage GETAL1
0204 85 01          STA, Zpage GETAL2
0206 86 00          STX, Zpage GETAL1
0208 4c22 IC        JMP, abs   RST
```

We zien nu in één oogopslag dat eerst getal 2 in register X komt te staan, vervolgens getal 1 in register A, om gelijk op de oude plaats van getal 2 gezet te worden. Tot slot wordt getal 2 dat nu in register X staat op de oude plaats van getal 1 gezet. (De JMP-instructie dient alleen om aan te geven dat het programma klaar is.) Vooral bij grotere programma's met veel geheugenplaatsen zeggen namen ons veel meer dan adressen. Ook geheugenplaatsen **in** het programma hebben soms een naam. Deze naam wordt dan vóór de instructie geplaatst. In bovenstaand voorbeeld hebben we adres 0200 de naam 'START' gegeven. Zo staat in het monitorprogramma op adres 1C 22 op dezelfde plaats 'RST'

Voorbeeld 3

Het kan in programma's ook voorkomen dat een bepaald geheugen en/of register gevuld moet worden met een reeds bekende waarde. (In het vorige voorbeeld konden de getallen die verwisseld werden buiten het programma om bepaald worden.) Daarom willen we nu een programma maken dat op adres 0000 en adres 0001 resp. de waarde '11' en '05' zet.

Met de tot nu toe besproken instructies zou het programma er als volgt uit kunnen zien.

```
0200 AD 0D 02 START  LDA, abs    ELF
0203 85 00           STA, Zpage  GETAL1
0205 AD 0E 02        LDA, abs    VIJF
0208 85 01           STA, Zpage  GETAL2
020A 4C 22 1C        JMP, abs    RST
020D 11         ELF
020E 05         VIJF
```

Allereerst wordt in register A het getal gezet van adres 020D. De programmeur heeft hier 11 gezet, omdat dit getal **door het programma** op adres 0000 gezet moet worden. Als u het programma heeft ingetypt kunt u het op adres 0200 starten. Ogenschijnlijk gebeurt er niets, maar controleert u nu maar eens de geheugenplaatsen 0000 en 0001. Voor deze eenvoudige opdracht hebben we maar liefst 15 geheugenplaatsen nodig. Het kan nog korter, met de derde, veel gebruikte adressering.

De Immediate addressing

Na de absolute en de zero page addressing komen we nu aan de 'Immediate addressing'. Met deze adresseermogelijkheid kunnen we een register met een vaste waarde vullen. Het programma van het vorige voorbeeld wordt nu:

```
0200 A9 11   START  LDA, imm.   $11
0202 85 00          STA, Zpage  GETAL1
0204 A9 05          LDA, imm.   $05
0206 85 01          STA, Zpage  GETAL2
0208 4C 22 1C       JMP, abs    RST
```

Het aantal geheugenplaatsen is teruggebracht tot 11, terwijl het programma wat overzichtelijker is geworden. Ook de code voor 'LDA' is weer anders, zodat de microprocessor ook weet dat hier 'immediate addressing' is gebruikt. Nog even voor de duidelijkheid: In register A komt nu niet het getal uit geheugenplaats $11 te staan, maar het getal $11 zelf. ($ betekent dat dit getal hexadecimaal, dus in het 16-tallig stelsel, geschreven is)

Het statusregister

Een belangrijk register in de microprocessor is het statusregister. Met dit register zijn we in staat de microprocessor voorwaardelijke sprongen te laten maken. Net als register A en X is het statusregister 8 bits breed. Van deze 8 bits worden er echter maar 7 gebruikt. Het statusregister dient niet zoals registers X en A om getallen in te bewaren, elk bitje van het statusregister heeft z'n eigen functie. In afb. 1 zijn alle inwendige registers van de 6502 nog eens getekend. De registers Y, PC en S zijn hiervan nog niet besproken. Bij het statusregister is de functie van elk bitje aangegeven.

Maar hoe werkt het statusregister nu? Voorlopig beperken we ons tot bit 1 van het statusregister, het Z-bit. Het Z-bit wordt automatisch door de microprocessor geset of gereset bij gebruik van bepaalde instructies. Op het instructiekaartje, waar de instructies met de bijbehorende code op zijn vermeld, is in een aparte kolom aangegeven welke instructies de bits in het statusregister activeren. Voor de beperkte instructieset die wij voor de overzichtelijkheid hanteren krijgen we nu het volgende lijstje:

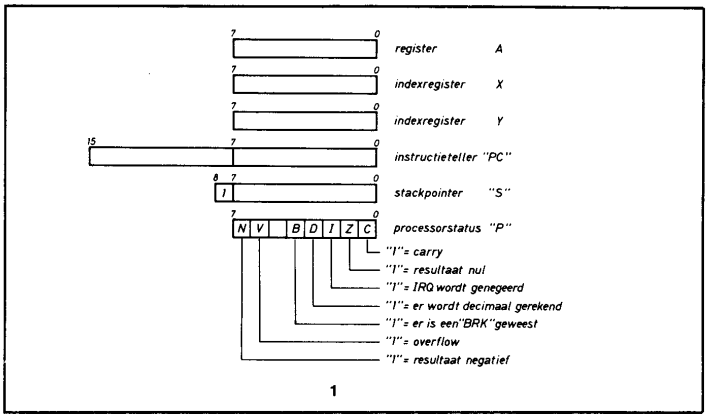
instr.	abs	Z-page	imm	Z-bit
LDA	AD	A5	A9	V
STA	8D	85	–	–
LDX	AE	A6	A2	V
STX	8E	86	–	–
JMP	4C	–	–	–

Een streepje in de laatste kolom betekent dat het Z-bit zijn oude waarde houdt, een V betekent dat het bit overeenkomstig de bijbehorende conditie '0' of '1' wordt. In ons geval wil dat zeggen dat het Z-bit '1' wordt zodra er '00' in register X of A wordt gezet, en '0' zodra er een ander getal in X of A wordt gezet.

Voorwaardelijke spronginstructies

De al eerder besproken 'JMP' instructie werd altijd uitgevoerd. De KIM kent ook nog voorwaardelijke spronginstructies, die afhankelijk van de toestand van een bepaald bitje in het statusregister al dan niet worden uitgevoerd. Voorlopig weer alleen de spronginstructies die betrekking hebben op het Z-bit:

1 De inwendige registers van de 6502. Registers Y, PC en S zijn hiervan nog niet besproken.



1

instructie	code	voorwaarde
BNE	DO	Z=0
BEQ	FO	Z=1

Wanneer niet aan de voorwaarde is voldaan, wordt de instructie niet uitgevoerd, en gaat de microprocessor gewoon verder met de volgende instructie. De voorwaardelijke spronginstructies bestaan alleen met de 'relative addressing'. We geven dus niet zoals bij 'JMP' een absoluut adres waar de microprocessor naar toe moet gaan, maar een aantal plaatsen dat heen of terug moet worden gesprongen. We zullen een en ander toelichten aan de hand van een voorbeeld.

Voorbeeld

Als voorbeeld nemen we een programma dat de geheugens tussen de adressen $0001 en $00E0 allemaal met '$EA' vult. Het programma komt er als volgt uit te zien:

```
0200 A9 EA        LDA, imm    $EA
0202 85 E0  NEXT  STA, Zpage  $E0
0204 CE 03 02     DEC, abs    $0203
0207 D0 F9        BNE, rel    NEXT
0209 A9 E0        LDA, imm    $E0
020B 8D 03 02     STA, abs    $0203
020E 4C 22 1C     JMP, abs    RST
```

Op adres $0204 zien we een instructie die we nog niet eerder hebben gebruikt, nl. 'DEC' (decrement). Deze instructie zorgt dat de inhoud van een bepaalde geheugenplaats met 1 verminderd wordt. In dit voorbeeld staat deze instructie in de 'absolute addressing' geschreven. Het programma loopt nu als volgt: De waarde EA komt via register A op adres $00E0. Vervolgens wordt geheugenplaats $0203 met 1 verminderd. Zolang dit niet tot gevolg heeft dat geheugen $0203 de waarde '00' krijgt, wordt het Z-bit uit het statusregister steeds op '0' gezet. Hierdoor zal de voorwaardelijke spronginstructie BNE (voorwaarde: Z=0) steeds uitgevoerd worden. In het programma zien we dat we naar geheugenplaats 'NEXT' moeten springen. (Hoe we aan F9 komen laten we nog even in het midden). Op deze plaats (adres $0202) wordt weer de inhoud van register A ('$EA') in het geheugen gezet, maar nu op locatie $00DF. (De inhoud van locatie $0203 was immers met 1 verminderd.) Hierna wordt de inhoud van adres $0203 opnieuw verminderd.

Het hele proces herhaalt zich, totdat op adres $0203 de waarde '$01' staat. Inmiddels staat dus op alle geheugenplaatsen tussen $01 en $E0 de waarde '$EA'. Wanneer nu de inhoud van $0203 opnieuw met 1 wordt verminderd, is het resultaat van deze bewerking '00'. Dit heeft tot gevolg dat het 'Z'-bit uit het statusregister nu '1' wordt.

Hierdoor zal de instructie 'BNE' genegeerd worden, en komen we aan het laatste deel van het programma, vanaf adres $0209. Hier zetten we via register A weer '$E0' op adres $0203, zodat het programma weer is hersteld.

Als we dit niet doen, zouden we elke keer dat we het programma uitvoeren, eerst adres $0203 moeten controleren. Met dit programma hebben we bereikt wat we wilden, het is echter geen elegante oplossing. Een programma dat op deze manier werkt kan nooit in een ROM opgeslagen worden. Dit, omdat het programma zichzelf voortdurend verandert (adres $0203). Er is bovendien een veel mooiere en kortere methode om hetzelfde te bereiken, nl. met de 'indexed addressing'.

De indexed addressing

Bij de indexed addressing wordt bij de operand eerst de inhoud van het indexregister geteld, de som van beide getallen wordt het adres waar wat mee gedaan gaat worden. De codes voor de indexed addressing zijn als volgt:

instr.	abs.X	Zp, X
LDA	BD	B5
STA	9D	95
LDX	–	–
STX	–	–
DEC	DE	D6
JMP	–	–

Zoals we zien zijn er twee soorten indexed addressing, nl. weer de 'absolute indexed addressing' welke in de hele geheugenruimte gebruikt kan worden, en de 'Zero page indexed addressing' welke alleen gebruikt kan worden op pagina 0, en hierdoor maar 2 i.p.v. 3 geheugenplaatsen nodig heeft. We zullen nu het programma uit het voorbeeld nog eens schrijven, maar nu met behulp van de 'indexed addressing'. Omdat de gebruikte geheugenplaatsen in pagina 0 liggen gebruiken we de 'Zero page indexed addressing'.

```
0200 A2 E0        LDX,imm    $EO
0202 A9 EA        LDA,imm    $EA
0204 95 00  NEXT  STA,Zp,X   $0000
0206 CA           DEX,impl
0207 D0 FB        BNE,rel    NEXT
0209 4C 22 1C     JMP,abs    RST
```

Ook hier zien we op adres $0206 een nieuwe instructie, nl. 'DEX'. Deze instructie vermindert de inhoud van het indexregister X met 1. Het verschil met DEC' is dus dat de instructie geen betrekking heeft op het geheugen, maar op één inwendig register v.d. microprocessor. Daarom kunnen we deze instructie alleen in de 'implied addressing' schrijven, het adres is in de opcode 'ingesloten'.

De werking v.h. programma is als volgt: In register X komt de waarde $E0 te staan, vervolgens wordt de waarde $EA via register A naar het geheugen geschreven (adres $0204). Het adres waar $EA terecht komt wordt door de microprocessor berekend door de inhoud van register X op te tellen bij de operand van de instructie (deze staat op adres $0205). Het resultaat van deze berekening wordt: $00+$E0=$E0, dus op adres $E0 komt de waarde '$EA' te staan. Vervolgens wordt de inhoud van register X met 1 verminderd. Zolang het resultaat hiervan geen '00' is, wordt het Z-bit uit het statusregister steeds op '0' gezet, en wordt de sprongopdracht steeds uitgevoerd. Dus de inhoud van register X doorloopt alle waarden tussen $E0 en $01, en ook de geheugens $E0 . . . $01 worden met '$EA' gevuld. Op het moment dat de inhoud van (index) register X van 01 naar 00 gaat, wordt het Z-bit uit het statusregister op 1 gezet. De BNE-instructie wordt nu overgeslagen en we springen d.m.v. JMP naar het monitorprogramma.

We zien dat hetzelfde programma door gebruik van de indexed addressing nu maar 12 i.p.v. 17 geheugenplaatsen inneemt. Bovendien treden er nu in het programma zelf geen wijzigingen op, zodat het programma in een ROM (Read Only Memory) gezet zou kunnen worden. Er is echter nog een reden waarom de laatste methode de voorkeur verdient. In het eerste voorbeeld konden we alleen de bovengrens ($E0) van het stuk geheugen waarin een vaste waarde werd geschreven makkelijk variëren, de ondergrens was niet zondermeer te verschuiven. Dit komt, omdat de voorwaardelijke sprong pas overgeslagen wordt als de waarde '00' bereikt werd, daardoor lag de ondergrens altijd bij adres $01.

Bij de indexed addressing hebben we de mogelijkheid om beide grenzen te variëren, zonder extra instructies. Stel, we willen nu alleen de adressen van $0020 . . . $0030 met een vaste waarde vullen. We kunnen hiervoor weer hetzelfde programma gebruiken, alleen de grenzen moeten gewijzigd worden.

```
0200 A2 11        LDX,imm    $11
0202 A9 08        LDA,imm    $00
0204 95 1F  NEXT  STA,Zp.X   $1F
0206 CA           DEX,impl
0207 DO FB        BNE,rel    NEXT
0209 4C 22 1C     JMP,abs    RST
```

De begininhoud van register X is nu $11 i.p.v. $E0, en de operand van STA is nu $1F i.p.v. $00. De inhoud van register X doorloopt nu de waarden $11 . . . $01, zodat de bovenste grens bij $1F+$11 = $30 komt te liggen, en de onderste grens bij $1F+$01=$20 (steeds de operand + inhoud indexreg. X). De vas-

te waarde waarmee de geheugens gevuld worden, hebben we in dit voorbeeld ook anders gemaakt, ($08 i.p.v. $EA) omdat we anders niet kunnen controleren of het programma de opdracht goed heeft uitgevoerd. Op de adressen $01 . . . $E0 was immers door het vorige programma al '$EA' geschreven. Als het goed is, moet na het starten van het programma (de tijd is nog steeds zo kort dat het net lijkt alsof er niets gebeurt) de geheugens $0020 . . . $0030 gevuld zijn met '08'.

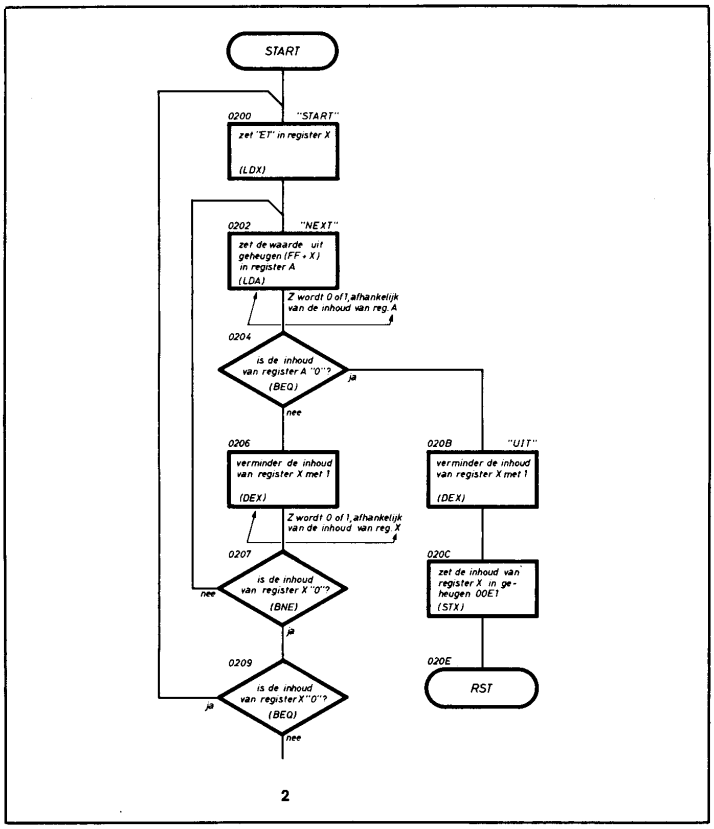
Nog een voorbeeld

In het volgende voorbeeld willen we een programma, dat vanaf adres $00E0 tot en met adres $0000 gaat zoeken naar een geheugenplaats waar '00' in staat, en vervolgens het adres van deze geheugenplaats wegzet in geheugenplaats $00E1. Wanneer we het programma starten, terwijl we van te voren b.v. adres $0037 de waarde '00' hebben gegeven, zal de inhoud van $00E1 dus '$37' zijn. Wanneer er géén '00' te vinden is, zal het programma blijven zoeken, en het display blijft donker.

```
0200 A2 E1    START  LDX,imm    $E1
0202 B5 FF    NEXT   LDA,Zp,X   $FF
0204 F0 05           BEQ,rel    UIT
0206 CA              DEX,impl.
0207 D0 F9           BNE,rel    NEXT
0209 F0 F5           BEQ,rel    START
020B CA       UIT    DEX,impl
020C 86 E1           STX,Zpage  $E1
020E 4C 22 1C        JMP,abs    RST
```

Zoals u ziet is dit programma al aanzienlijk ingewikkelder dan de tot nu toe besproken voorbeelden. Doordat het aantal voorwaardelijke sprongen is toegenomen is het overzicht een beetje verloren gegaan. Dat is dan ook de reden, dat een programmeur zich bedient van een flow-chart. In afb. 2 ziet u de flowchart van dit programma. Bij deze flowchart heeft u meteen een veel beter overzicht in het programmaverloop.

Allereerst wordt register X gevuld met de waarde $E1. Hierdoor zal register A gevuld worden met de inhoud van adres $E1+$FF=$E0 (eigenlijk is dit $01E0, maar de microprocessor negeert de carry, zodat het resultaat van de 'Zeropage indexed addressing' ook altijd op pagina 0 ligt). Wanneer het getal dat in register A is gekomen géén '00' was, zal het Z-bit 0 gemaakt worden, en de sprong naar 'uit' wordt niet gemaakt. De waarde van X wordt met 1 verminderd, zolang dit geen '00' is gaan we terug naar 'NEXT'. Wanneer register X de waarde $01 heeft bereikt, zal de instructie 'DEX' (adres $0206) er voor zorgen dat register X '00' wordt. Hierdoor wordt Z=1, en nu zal 'BNE' (adres $0207) worden genegeerd. Op adres



2

2 De flowchart van het zoekprogramma. Omdat een flowchart een veel beter overzicht van het programmaverloop geeft, wordt altijd éérst een flowchart gemaakt.

$0209 springen we nu weer naar 'START'. Doordat de instructie BNE en BEQ achter elkaar geschreven staan, zal er altijd gesprongen worden. (Als Z=0 via BNE naar 'NEXT', en als Z=1 via BEQ naar 'START'). Op 'START' begint de zoek cyclus opnieuw, er zat echter geen '00' in de te doorzoeken ruimte, zodat het programma blijft lopen, totdat op 'RS' of 'ST' gedrukt wordt. Wanneer er wel een '00' gevonden wordt, zal de microprocessor via de BEQ instructie op adres $0204 naar 'uit' springen. Hier moeten we het adres van de geheugenplaats waar '00' stond op adres $00E1 zetten. Stel dat indexregister X op dat moment de waarde $81 heeft, dan is de geheugenplaats waar '00' stond te berekenen door $81 op te tellen bij de operand van 'LDA, Zp, X'. We krijgen $81+$FF=$0180. Zoals eerder gezegd wordt de carry naar de $100-tallen genegeerd, waardoor de gevonden '00' van adres $80 gekomen moet zijn. We kunnen blijkbaar volstaan met het verminderen met 1 van indexregister X. Dit gebeurt dan op adres $020B. Tot slot wordt de waarde van indexregister X op geheugenplaats $E1 gezet, en het programma is klaar.

Het bepalen van de relatieve sprong

Bij alle voorwaardelijke sprongen in de vorige voorbeelden, werd een relatief adres gebruikt. Dat wil zeggen, dat alleen wordt aangegeven hoeveel plaatsen heen of terug moet worden gesprongen. Om de juiste waarde van deze sprong te bepalen bestaan verschillende methoden. Theoretisch wordt de waarde van de sprong verkregen door de waarde van de instructieteller af te trekken van het adres waar naar toe gesprongen moet worden. Als we in het laatste voorbeeld op adres 0204 kijken zien we daar de instructie

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15	00
10	+16	+17	+18	+19	+20	+21	+22	+23	+24	+25	+26	+27	+28	+29	+30	+31	10
20	+32	+33	+34	+35	+36	+37	+38	+39	+40	+41	+42	+43	+44	+45	+46	+47	20
30	+48	+49	+50	+51	+52	+53	+54	+55	+56	+57	+58	+59	+60	+61	+62	+63	30
40	+64	+65	+66	+67	+68	+69	+70	+71	+72	+73	+74	+75	+76	+77	+78	+79	40
50	+80	+81	+82	+83	+84	+85	+86	+87	+88	+89	+90	+91	+92	+93	+94	+95	50
60	+96	+97	+98	+99	+100	+101	+102	+103	+104	+105	+106	+107	+108	+109	+110	+111	60
70	+112	+113	+114	+115	+116	+117	+118	+119	+120	+121	+122	+123	+124	+125	+126	+127	70
80	−128	−127	−126	−125	−124	−123	−122	−121	−120	−119	−118	−117	−116	−115	−114	−113	80
90	−112	−111	−110	−109	−108	−107	−106	−105	−104	−103	−102	−101	−100	−99	−98	−97	90
A0	−96	−95	−94	−93	−92	−91	−90	−89	−88	−87	−86	−85	−84	−83	−82	−81	A0
B0	−80	−79	−78	−77	−76	−75	−74	−73	−72	−71	−70	−69	−68	−67	−66	−65	B0
C0	−64	−63	−62	−61	−60	−59	−58	−57	−56	−55	−54	−53	−52	−51	−50	−49	C0
D0	−48	−47	−46	−45	−44	−43	−42	−41	−40	−39	−38	−37	−36	−35	−34	−33	D0
E0	−32	−31	−30	−29	−28	−27	−26	−25	−24	−23	−22	−21	−20	−19	−18	−17	E0
F0	−16	−15	−14	−13	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1	F0
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Tabel 1 voor het omzetten van een positieve of negatieve sprongwaarde naar het hexa-decimale (two's-complement) equivalent.

'BEQ, rel UIT'. Er moet dus een relatieve sprong berekend worden naar $020B. Op het moment dat de microprocessor het nieuwe adres uit gaat rekenen is de instructieteller echter al opgehoogd tot de volgende instructie, en staat dus inmiddels op $0206. De waarde van de sprong volgt nu uit: $020B-$0206 = $5.

Deze methode is echter niet handig, omdat we de aftrekking hexa-decimaal moeten uitvoeren. Vooral als terug gesprongen moet worden, moet de negatieve hexa-decimale waarde worden omgezet naar het two's-complement, hetgeen zeker in het begin de nodige problemen zal opleveren. Daarom kunnen we het beste gewoon het aantal stappen tellen. Met behulp van tabel 1 kan het aantal stappen dan omgezet worden naar het hexa-decimale equivalent.

Voorbeeld voor het bepalen van de relatieve sprong

We zullen een en ander weer toelichten met een voorbeeldje. We nemen uit het laatste programmavoorbeeld op adres $0207 de instructie 'BNE, rel NEXT'. We houden er even rekening mee dat de instructieteller reeds is opgehoogd tot de volgende instructie (adres $0209, 'F0'). Op deze plaats zetten we een pen of potlood en gaan al tellend via de geheugeninhouden F9, D0, CA, 05, F0, FF, B5 naar adres $0202. Dit was het adres waar we heen moesten, en we hebben tot 7 geteld. Omdat we **terug** moeten, kijken we nu in tabel 1 bij −7. Dit getal vinden we op regel F0 onder kolom 9, zodat we voor de sprong (adres $0208) $F9 in kunnen vullen. Nog een voorbeeld: adres $0209, ook van het laatste voorbeeld, de voorwaardelijke sprong 'BEQ, rel START'. We zetten weer een pen of potlood op de **volgende** instructie (dus op $020B, 'CA') en tellen de plaatsen F5, F0, F9, D0, CA, 05, F0, FF, B5, E1, A2. De waarde 'A2' staat op het adres waar we heen moesten springen. We komen uit op 11. Omdat we weer **terug** springen kijken we in de tabel bij −11, en vinden dit getal op rij F0, in kolom 5. De juiste sprongwaarde is dus $F5. Ook sprongen in de richting van het programma kunnen op deze manier worden bepaald. Weer in het laatste voorbeeld (op adres $0204) staat de instructie 'BEQ, rel UIT'. Ook nu kijken we naar de **volgende** instructie ('CA' adres $0206). We tellen vervolgens naar 'UIT' (over de getallen: D0, F9, F0, F5, CA) en komen op 5. We springen nu **heen,** dus we kijken in de tabel bij +5. Dit getal vinden we in de rij 00 onder kolom 5. Het hexa-decimale equivalent is dus ook $05. Alle getallen onder de 10 geven een hexa-decimaal equivalent, dat gelijk is aan het decimale getal. Voor de veiligheid is het echter het beste om (voorlopig) altijd in de tabel te kijken.

Het controleren van de sprong

Wanneer u in een zelf geschreven programma voorwaardelijke sprongen hebt gebruikt is het zeker aan te bevelen om eerst te controleren of de sprongen juist berekend zijn. Een verkeerde sprong heeft vaak niet alleen tot gevolg dat het programma niet werkt, maar kan ook een vernietiging van het hele programma (of een gedeelte daarvan) veroorzaken. Daarom zeker in het begin: altijd éérst de relatieve sprongen controleren. Dit kan als volgt: Allereerst het SST-schakelaartje op 'ON', zodat het programma in single step kan worden doorlopen. Typ vervolgens adres $00F1 in, het adres waar het statusregister gecontroleerd en gewijzigd kan worden. Voor de spronginstructies met voorwaarde statusbit = 0 (zoals BNE) typen we '00' in, voor sprong-instructies met voorwaarde statusbit = 1 (zoals BEQ) typen we '$FF' in. Nu typen we het adres van de sprong-instructie in, en drukken op 'GO'. De sprong wordt nu altijd uitgevoerd, omdat we vooraf het statusregister goed hebben gezet. Op het display verschijnt nu het adres waar de microprocessor heen is gesprongen. Wanneer we te ver **terug** zijn gesprongen, moet de sprongwaarde **groter** worden. Wanneer te ver **heen** gesprongen is, moet de sprongwaarde **kleiner** worden. Wanneer alle sprongen op deze manier zijn gecontroleerd, en eventueel verbeterd, kunt u met een gerust hart het programma starten. Ten overvloede: Een foutieve sprong kan **nooit** de KIM beschadigen, in het ergste geval moet u het programma opnieuw intypen.

Het vergelijken van 2 getallen

We hebben nu gezien dat bepaalde instructies het Z-bit in het statusregister '1' maken als de inhoud van een register of geheugenplaats naar '00' gaat. Met behulp van de stand van het Z-bit kunnen we dan weer voorwaardelijke sprongen maken. Met een speciale instructie is het ook mogelijk het Z-bit '1' te laten worden wanneer 2 getallen gelijk zijn. De instructie heet 'CMP' (compare). De codes zijn als volgt:

instr.	imm.	abs.	Zpage	Zp,X	abs,X
CMP	C9	CD	C5	D5	DD

Wanneer we in een programma b.v. schrijven:

CMP,imm $63

dan zal het Z-bit uit het statusregister '1' worden indien de inhoud van register A

op dat moment ook $63 is. Met de CMP instructie kunnen we dus voorwaardelijke sprongen verzorgen voor elke willekeurige waarde van register A. We kunnen register A ook vergelijken met de inhoud v.e. bepaalde geheugenplaats:

CMP,Zpage $63

Nu zal het Z-bit alleen 1 worden als de inhoud van register A hetzelfde is als de inhoud van geheugenplaats $0063. Tot slot geven we nog een programmavoorbeeld waar 'CMP' gebruikt wordt. Het is weer het eerder genoemde zoekprogramma. Nu wordt echter niet gezocht naar de waarde '00', maar naar een getal dat u zelf op adres $00E2 kunt definiëren. Als er een plaats gevonden wordt waar dit getal staat, zal deze plaats op het display zichtbaar gemaakt worden. Wanneer er geen plaats gevonden wordt met de gedefinieerde inhoud, blijft het display het startadres van het programma weergeven.

```
0200 A2 E1      START  LDX,imm    $E1
0202 B5 FF      NEXT   LDA,Zp,X   $FF
0204 C5 E2             CMP,Zpage  $E2
0206 F0 05             BEQ,rel    UIT
0208 CA                DEX,impl
0209 D0 F7             BNE,rel    NEXT
020B F0 07             BEQ,rel    RST2
020D CA         UIT    DEX,impl
020E 86 FA             STX,Zpage  POINTL
0210 A9 00             LDA,imm    $00
0212 85 FB             STA,Zpage  POINTH
0214 4C 22 1C   RST2   JMP,abs    RST
```

We gaan nu niet verder in op de werking van dit programma, maar controleert u voor uzelf of u aan dezelfde waarden komt voor de relatieve sprongen (vet gedrukt). Ook ziet u hier het begin van het sturen van het display.

De in- en uitgangen van de KIM

Op de applicatieconnector van de KIM (de connector ter hoogte van het toetsenbordje) bevinden zich op de punten 2 . . . 16 de 15 aansluitingen welke de KIM heeft. Voor deze aansluitingen heeft de 6502 microprocessor geen aparte instructie. De aansluitingen van de KIM zien er voor de microprocessor gewoon uit als geheugenplaatsen. Omdat de data-bus 8 bits breed is, kunnen op één geheugenplaats maximaal 8 aansluitingen zitten. Voor de 15 aansluitingen zijn dus 2 geheugenplaatsen nodig.

Wanneer we het adres van zo'n geheugenplaats intypen, zien we op het display wat de toestand van de ingangen is (na een druk op 'RS' zijn alle aansluitingen als ingang geprogrammeerd). Wanneer de ingangen open zijn, zien we op adres $1700 de waarde $FF. Door deze hexa-decimale waarde om te zetten naar het binaire equivalent is het mogelijk om de toestand van elk bitje te bepalen. $FF komt overeen met het binaire getal '11111111', en we zien dus dat alle (open) ingangen gelezen worden als een logische '1'. Als we bepaalde ingangen van poort PA '0' maken kunnen we op het display zien wat er gebeurt, het beste bewijs dat de KIM nu inderdaad naar zijn aansluitingen kijkt. Als voorbeeld geven we nu een programma dat alleen naar aansluiting PA3 kijkt. Zolang deze aansluiting hoog is, zal het display donker blijven. Wanneer deze aansluiting laag wordt, zal het programma worden beëindigd waardoor het display oplicht. De computer mag echter niet reageren op een verandering van een van de andere ingangen. Wanneer de microprocessor de opdracht LDA $1700 krijgt, worden tegelijkertijd de ingangen PA0 . . . PA7 gelezen. Om het mogelijk te maken alléén te reageren op PA3, gaan we de andere bits 'afdekken'. We maken hierbij gebruik van de 'AND' instructie.
Deze instructie geeft als resultaat de logische AND tussen een getal in register A en een getal in het geheugen. Voor de duidelijkheid schrijven we de getallen nu even binair i.p.v. hexa-decimaal:

```
inhoud register A      : 00110101
inhoud geheugen        : 10101100
resultaat in register A : 00100100
```

We zien dat de normale AND tussen alle bitjes wordt uitgevoerd. Het belangrijkste hierbij is dat een '0' in een van beide getallen altijd een '0' in het resultaat geeft. Wanneer we nu de toestand van de ingangen in register A zetten en met $08 afdekken, krijgen we:

```
inhoud register A  : X X X X X X X X
inhoud geheugen    : 0 0 0 0 1 0 0 0
resultaat register A : 0 0 0 0 X 0 0 0
```

Een X wil hierbij zeggen dat het bit zowel '0' als '1' kan zijn. Op deze manier kunnen we dus elk bit of elke combinatie van bits isoleren. Het programma loopt nu als volgt:

```
0200 AD 00 17 START  LDA,abs   PAD
0203 29 08           AND,imm   $08
0205 D0 F9           BNE,rel   START
0207 4C 22 1C        JMP,abs   RST
```

Op adres $200 wordt register A gevuld met een waarde, die afhankelijk is van de toestand van de ingangen PA0 . . . PA7. Deze ingangen bevinden zich op adres $1700 (we kunnen dus geen zero-page addressing toepassen). Dit adres ($1700) heeft de naam PAD gekregen, hetgeen de afkorting is van Port A Data. Op adres 0203 worden alle bits die niet van belang zijn naar '0' gedwongen op de besproken wijze. Merk op dat het binaire getal 00001000 nu is geschreven in de kortere hexadecimale vorm $08. Zolang bit PA3 géén 0 is, zal het resultaat van deze AND instructie ook geen '00' zijn. Het Z-bit uit het statusregister zal dus steeds op '0' gezet worden, zodat op adres $0205 de sprong naar start wordt gemaakt. Zodra PA3 laag is geworden, zal het resultaat van de AND instructie '00' als resultaat geven:

```
inhoud register A  : X X X X 0 X X X
inhoud geheugen    : 0 0 0 0 1 0 0 0
resultaat register A : 0 0 0 0 0 0 0 0
```

Het gevolg is dat het Z-bit uit het statusregister op '1' gezet wordt. De sprong-instructie (BNE) wordt hierdoor niet uitgevoerd, en via 'JMP' springen we terug naar het monitorprogramma.

De aansluitingen PB0 . . . PB5 en PB7

Van poort B is aansluiting PB6 niet beschikbaar, deze aansluiting wordt bij de KIM gebruikt voor de adres-selectie van de 6530. Poort B vinden we op adres $1702. Dit adres heeft de naam PBD gekregen (Port B Data). Wanneer u dit adres intypt, ziet u niet zoals bij PAD de waarde $FF, maar de waarde $3F of $BF. Binair geschreven wordt dit:

10111111 of 00111111

We zien dat bit PB6, welke we niet kunnen gebruiken, als '0' wordt gelezen. Ook bit PB7 gedraagd zich anders. Dit komt doordat PB7 geen inwendige pull-up weerstand heeft, waardoor de pin als hij open hangt, een ongedefinieerde waarde heeft. Eenmaal als uitgang geschakeld gedraagt pin PB7 zich als een opencollector uitgang, en kan parallel gezet worden met andere opencollector uitgangen.

De aansluitingen als uitgang

Zoals eerder gezegd kan elke aansluiting zowel ingang als uitgang zijn. Om een aansluitpen als uitgang te definiëren moeten we een '1' in het data richtingsregister zetten. Het richtingsregister voor de aansluitingen PA0 . . . PA7 vinden we op adres $1701 en heeft de naam PADD gekregen (Port A Data Direction). Het richtingsregister voor de aansluitingen PB0 . . . PB5 en PB7 vinden we op adres $1703 en heeft de naam PBDD gekregen (Port B Data Direction). Voor het overzicht even het volgende tabelletje:

adres	naam	functie
$1700	PAD	aansluitingen PA
$1701	PADD	richting PA
$1702	PBD	aansluitingen PB
$1703	PBDD	richting PB

Wanneer we dus in PADD de waarde $03 zetten, zullen PA0 en PA1 als uitgang fungeren en PA2 . . . PA7 als ingang. (Bij $03 zijn de bits 0 en 1 hoog, de rest laag.) Bij een druk op 'RS' zullen alle uitgangen weer als ingang worden geschakeld. U kunt dit controleren door op adres $1701 een bepaald getal te zetten. Na een druk op 'RS' zal er altijd weer '00' staan.

Doormeetzoemer

We zullen nu een voorbeeld laten zien waarin één van de aansluitingen als uitgang geschakeld wordt. Op deze uitgang (PA0) sluiten we een luidspreker aan volgens afb. 3. Het programma zorgt dat de luidspreker een toon weergeeft zodra PA3 aan massa wordt gelegd.

```
0200 A9 01      START  LDA,imm   $01
0202 8D 01 17          STA,abs   PADD
0205 AD 00 17  LEES    LDA,abs   PAD
0208 29 08             AND,imm   $08
020A D0 F9             BNE,rel   LEES
020C E8        LOOP    INX,impl
020D D0 FD             BNE,rel   LOOP
020F EE 00 17          INC,impl  PAD
0212 4C 05 02          JMP,abs   LEES
```

We zien dat we als eerste stap in het programma de aansluiting PA0 programmeren als uitgang en de rest als ingang ($01 wordt op PADD gezet). Vanaf adres 0205 begint het eigenlijke programma. Het eerste stuk is het programma uit het vorige voorbeeld, dit deel bepaalt of ingang PA3 hoog dan wel laag is. Zolang PA3 hoog is (open ingang) zal de microprocessor steeds terug springen naar 'LEES' adres $0205.

3. De aansluitingen welke voor de doormeetzoemer gemaakt moeten worden.

Zodra PA3 laag is, wordt de sprong naar 'LEES' niet gemaakt (het resultaat van AND $08 is nu 0, dus Z wordt '1'). We komen nu in het programmadeel dat de toon opwekt. Het principe is eenvoudig, geheugenplaats PAD wordt met 1 opgehoogd, en via JMP springen we weer naar 'LEES'. Zolang PA3 laag is rollen we steeds door het eerste deel (adres $0205 . . . $020A) heen en wordt 'PAD' voortdurend (adres 020F) met '1' opgehoogd. Door dit ophogen klapt PA0 voortdurend van 0 naar 1, en daarna van 1 naar 0. De rest van de aansluitingen is als ingang geprogrammeerd, en reageert niet op deze increment instructie. (De ingangen kunnen alleen van buiten af hoog of laag gemaakt worden.) Op uitgang PA0 verschijnt nu dus een blokgolf. Doordat de microprocessor maar een paar instructies hoeft uit te voeren, zullen de increment's elkaar zeer snel opvolgen, en de frequentie van de blokgolf komt boven de gehoorgrens te liggen. Daarom is op de adressen $020C . . . $020E een loop opgenomen. De microprocessor zal in deze loop voortdurend de inhoud van register X ophogen, net zolang totdat de inhoud van dit register '00' is. Door deze extra vertraging zal de periodetijd van de blokgolf zo groot worden, dat een hoorbare toon ontstaat. Een verschil met vorige programma's is dat dit programma steeds opnieuw wordt uitgevoerd. Hierdoor zal het display doven. Alleen met een druk op 'RS' of 'ST' (indien u de NMI vector ingevoerd had, 00 op $17FA en $1C op $17FB) kunt u terugkeren naar het monitorprogramma.

De stack

Een heel ander onderwerp is de stack van de KIM. In een van de eerste artikelen in RB over de microprocessor is de stack al eens behandeld. De stack is een stapel registers waarin gegevens tijdelijk opgeslagen kunnen worden. Een voordeel is dat de geheugenplaats waarin een bepaalde waarde komt te staan niet geadresseerd hoeft te worden, een beperking is dat de gegevens uitsluitend in omgekeerde volgorde weer teruggeroepen kunnen worden. De instructies die bij de stack horen zijn voorlopig:

instructie	code
PHA	48
PLA	68

De instructie 'PHA' (PUSH register A) betekent: 'zet de inhoud van register A in de stack'. Deze instructie neemt maar één geheugenplaats in beslag, en is daarom aantrekkelijk. De omgekeerde instructie is 'PLA' (PULL register A). Deze instructie zet de waarde welke het laatst in de stack gezet is, in register A. De waarden welke in de stack zijn gezet, komen uiteindelijk gewoon op bepaalde adressen in het RAM terecht, hoe dit werkt behandelen we nog, maar eerst een voorbeeld waarin we de stack gebruiken.

Voorbeeld

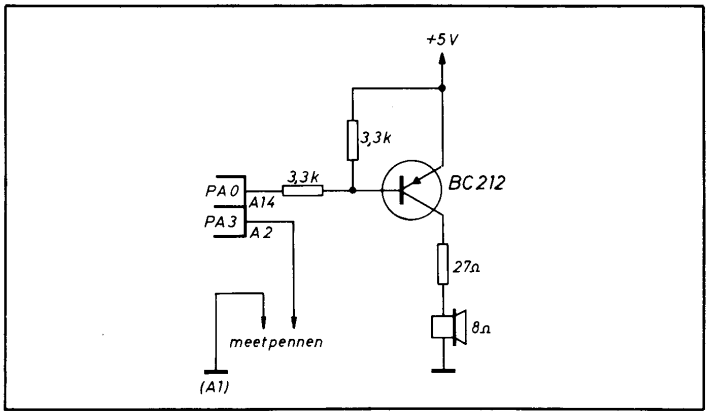
In het volgende voorbeeld zetten we eerst 5 verschillende getallen in de stack en vervolgens halen we deze getallen er weer uit. Het eerste getal uit de stack zetten we op adres $0000, het laatste getal komt op adres $0004. Op deze manier zien we duidelijk in welke volgorde de getallen weer uit de stack komen.

```
0200 A9 11     IN      LDA,imm   $11
0202 48                PHA,imp
0203 A9 22             LDA,imm   $22
0205 48                PHA,imp
0206 A9 33             LDA,imm   $33
0208 48                PHA,imp
0209 A9 44             LDA,imm   $44
020B 48                PHA,imp
020C A9 55             LDA,imm   $55
020E 48                PHA,imp
020F 68        UIT     PLA,imp
0210 85 00             STA,Zpage nul
0212 68                PLA,imp
0213 85 01             STA,Zpage een
0215 68                PLA,imp
0216 85 02             STA,Zpage twee
0218 68                PLA,imp
0219 85 03             STA,Zpage drie
021B 68                PLA,imp
021C 85 04             STA,Zpage vier
021E 4C 22 1C          JMP,abs
```

Na het uitvoeren van dit programma zien we, dat het getal dat het eerste in de stack gezet is, er pas het laatst uitkomt (op adres $0000 staat $55, en op adres $0004 staat $11).

Werking van de stack

De KIM gebruikt voor de stack een deel



van het RAM, en wel de adresruimte van $0100 . . . $01FF. In een speciaal register houdt de microprocessor bij welk adres nog niet gebruikt is. Dit register heeft de naam 'S' gekregen, de afkorting van Stack Pointer. Na een druk op 'RS' zal de inhoud van register 'S' altijd $1FF zijn, waardoor het eerste getal dat met PHA in de stack wordt gezet op adres $01FF zal belanden. Met 'PHA' wordt bovendien automatisch de stackpointer met 1 verminderd tot $1FE. In ons vorige voorbeeld werd dus $11 op adres $01FF gezet, $22 op adres $01FE, $33 op adres $01FD, $44 op adres $01FC en tot slot $55 op adres $01FB.

Als u na het uitvoeren van het laatste voorbeeld adres $01FB bekijkt, zult u ook $55 als inhoud zjen. De adressen $01FC . . . $01FF zijn inmiddels overschreven door het monitorprogramma, omdat ook het monitorprogramma gebruik maakt van de stack. Bij iedere 'PLA' gebeurt het omgekeerde, de stackpointer wordt verhoogd, en het getal dat op de door de stack pointer bepaalde plaats staat wordt in register A gezet. Op adres $00F2 kunnen we steeds kijken wat de toestand van de stackpointer is. Na een druk op 'RS' zal de inhoud van $00F2 altijd $FF zijn (adres $00F2 is **niet** de stackpointer zelf, het is een copie van de stackpointer gemaakt door het monitorprogramma). Door het vorige voorbeeld in single stap te doorlopen kunt u het hele proces volgen. Na elke PHA ($48) ziet u dat de stackpointer (te controleren op adres $00F2) met 1 verlaagd is, en u ziet dat de getallen $11 . . . $55 één voor één in de stack gezet worden. Na elke PLA ($68) ziet u het omgekeerde gebeuren. Omdat het monitorprogramma ook gebruikt maakt van de stack zullen getallen die eenmaal uit de stack zijn gehaald, meteen overschreven worden, zodat de indruk wordt gewekt dat de getallen die uit de stack worden gehaald ook echt uit het geheugen verdwijnen. In werkelijkheid (als we niet in single-step werken) blijft een getal na PLA gewoon in het geheugen staan. In register A komt een copie van dat getal uit de stack, terwijl de stackpointer de betreffende geheugenplaats aanwijst als eerste geheugenplaats waar weer een getal in gezet kan worden. De stackpointer regelt deze zaken echter feilloos, zodat we ons in deze materie niet zo hoeven te verdiepen.

De subroutines

In programma's gebeurt het vaak dat bepaalde programmastukjes veel voorkomen. Het is dan handig om zo'n programmastukje als subroutine uit te voeren. We zullen dit weer met een voorbeeld toelichten. Al eerder in dit artikel hebben we het gehad over de ingangen van de KIM. Als voorbeeld maken we nu een programma dat telt hoeveel maal een bepaalde ingang '0' is geweest, dus hoeveel maal deze ingang aan massa is gelegd. Bij het starten van het programma zal de ingang '1' zijn (open hangen). In de loop wachten we nu totdat deze ingang laag wordt. Zodra dit het geval is, moet een teller opgehoogd worden, zodat we kunnen zien hoeveel maal de ingang laag is geweest. Hierna moeten we weer wachten totdat de ingang hoog wordt, en dan pas kunnen we weer naar het begin van het programma springen. Om te voorkomen dat de teller door de contactdender teveel wordt opgehoogd maken we in het programma ook een vertraging. Het programma (zonder subroutines) is te zien in lijst 1. We zien dat aan deze lijst nog een extra kolom is toegevoegd, waarin een korte toelichting wordt gegeven op de programmastappen. We zien in dit programma een aantal nieuwe instructies. Allereerst 'TAX' (adres $0204), deze instructie zorgt er voor dat een getal dat in register A staat óók in register X komt te staan. We gebruiken deze instructie hier om register X de begininhoud '00' te geven. In principe hadden we ook LDX,imm $00 kunnen gebruiken, maar deze instructie neemt 2 geheugenplaatsen in beslag i.p.v. 1.

Verder maken we hier kennis met het tweede indexregister, nl. indexregister Y. Dit register heeft ongeveer dezelfde mogelijkheden als register X. Voor sommige taken heeft register X de aantrekkelijkste instructies, voor sommige taken heeft register Y de voorkeur. Hier wordt register Y gebruikt om 2 loops in elkaar te maken. Dit omdat 1 loop niet genoeg vertraging geeft om de contact dender op te vangen. Register X begint een loop altijd met '00', en eindigt een loop altijd met '00'. Register Y krijgt steeds een beginwaarde, met deze waarde is de vertraging groter of kleiner te maken. De instructies bij register Y:

instr.	imm.	abs.	Zpage	Zp,X	abs,X	impl.
LDY	A0	AC	A4	B4	BC	–
DEY	–	–	–	–	–	88

De instructie 'DEY' (decrement reg. Y) vermindert de inhoud van indexregister Y met één. Doordat deze instructie altijd betrekking heeft op register Y is geen nader adres nodig, en is de addressing mode implied.
Maar weer terug naar ons voorbeeld, wanneer we het programma starten dooft het display. We maken u ingang PA0 een aantal malen '0' door hem aan massa te leggen. We drukken op 'RS', en op adres $0000 zien we nu hoeveel maal ingang PA0 laag is geweest.

Hetzelfde voorbeeld met gebruikmaking van subroutines

Bij het programma volgens lijst 1 zien we dat het programmastukje tussen adres $0205 en $0212 bijna gelijk is aan het programmastukje tussen $0216 en $0223. Alleen de adressen $020A en $021B verschillen iets. De ene keer

moeten we nl. wachten tot de ingang '0' is geworden. De tweede keer moeten we juist wachten totdat de ingang '1' is geworden. Ondanks het kleine verschil kunnen we deze programmastukjes toch als subroutine uitvoeren, de beslissing of we terugspringen bij ingang = 1 of ingang = 0 moeten we gewoon uitstellen totdat we uit de subroutine terugkeren. Om dit te bereiken moeten we de inhoud van het statusregister even opbergen, want de toestand van het Z-bit uit dit register geeft aan of de ingang nu 0 of 1 was (zie het vorige deel). De KIM heeft hiervoor een handige instructie, nl:

instr.	code
PHP	08
PLP	28

We hebben dus de mogelijkheid om de inhoud van het statusregister in de stack te zetten. Door de vertragingloops wordt de toestand van het Z-bit veranderd, maar na 'PLP' staat het Z-bit weer volgens de ingangconditie. (Deze waarde kreeg hij na de AND instructie, Z=1 als PA0=0 en omgekeerd). In lijst 2 zien we nu het programma mét gebruik van subroutines. In ons geval is het programma vier geheugenplaatsen korter en het lijkt zonde van de moeite. U moet echter bedenken dat we als voorbeeld een kort programma hebben gebruikt. Hoe langer de subroutine, en hoe meer hij aangeroepen wordt, des te groter wordt de winst in geheugenruimte. Daarbij komt dat een programma vaak wat overzichtelijker wordt bij gebruik van subroutines. Dit omdat een bepaalde subroutine gauw wordt gezien als een soort 'instructie op maat'.

Lijst 1 Het telprogramma

Lijst 2 Het telprogramma, nu met gebruikmaking van subroutines.

Lijst 1

0200	A9 00	START	LDA,imm	$00	
0202	85 00		STA,Zpage	TELLER	register op 0
0204	AA		TAX,impl		
0205	AD 00 17	WACHT1	LDA,abs	PAD	wacht tot de
0208	29 01		AND,imm	$01	ingang 0 is
020A	D0 F9		BNE,rel	WACHT1	geworden
020C	A0 40		LDY,imm	$40	
020E	E8	LOOP1	INX,impl		wacht totdat er
020F	D0 FD		BNE,rel	LOOP1	geen dender
0211	88		DEY,impl		meer is
0212	D0 FA		BNE,rel	LOOP1	
0214	E6 00		INC,Zpage	TELLER	hoog de teller op
0216	AD 00 17	WACHT2	LDA,abs	PAD	wacht tot de
0219	29 01		AND,imm	$01	ingang 1 is
021B	F0 F9		BEQ,rel	WACHT2	geworden
021D	A0 40		LDY,imm	$40	
021F	E8	LOOP2	INX,impl		wacht totdat
0220	D0 FD		BNE,rel	LOOP2	er geen dender
0222	88		DEY,impl		meer is
0223	D0 FA		BNE,rel	LOOP2	
0225	4C 05 02		JMP,abs	WACHT1	ga terug

Lijst 2

0200	A9 00	START	LDA,imm	$00	
0202	85 00		STA,Zpage	TELLER	register op 0
0204	AA		TAX,impl		
0205	20 14 02	WACHT1	JSR,abs	SUB	wacht tot de ing. 0
0208	D0 FB		BNE,rel	WACHT1	is, en uitgedenderd
020A	E6 00		INC,Zpage	TELLER	hoog de teller op
020C	20 14 02	WACHT2	JSR,abs	SUB	wacht tot de ing. 1
020F	F0 FB		BEQ,rel	WACHT2	is en uitgedenderd,
0211	4C 05 02		JMP,abs	WACHT1	ga terug
0214	AD 00 17	SUB	LDA,abs	PAD	is de ingang
0217	29 01		AND,imm	$01	0 of 1?
0219	08		PHP,impl		zet status weg
021A	A0 40		LDY,imm	$40	
021C	E8	LOOP	INX,impl		wacht totdat er
021D	D0 FD		BNE,rel	LOOP	geen dender
021F	88		DEY,impl		meer is
0220	D0 FA		BNE,rel	LOOP	
0222	28		PLP,impl		haal status terug
0223	60		RTS,impl		terug naar het hoofdprogramma

Het optellen van twee 8-bits binaire getallen

We beginnen met de eenvoudigste rekenkundige bewerking, nl. het optellen van twee 8-bits getallen. De instructie die de 6502 hiervoor heeft is 'ADC', deze instructie zorgt dat een getal uit het geheugen wordt opgeteld bij de inhoud van de accumulator. De codes voor de verschillende tot nu toe behandelde adresseringsmethoden:

inst.	imm.	abs.	Zpage
ADC	69	6D	65

Zp,X	abs,X	abs,Y
75	7D	79

In het nu volgende voorbeeld gaan we er vanuit dat beide op te tellen getallen op adressen $00 en $10 staan:

```
0200 18       START CLC impl.
0201 D8             CLD impl.
0202 A5 00          LDA, Zpage GETAL 1
0204 65 10          ADC, page GETAL 2
0206 85 20          STA, Zpage GETAL 3
0208 4C 22 1C       JMP, abs. RST
```

Wanneer we nu bv. $12 op adres $100 zetten, en $28 op adres $10, zal er (na het uitvoeren van het programma) $3A op adres $20 komen te staan. Dit is inderdaad de hexadecimale som van beide getallen. Op de instructie 'CLC' (adres $0200) komen we nog terug. Met de instructies 'CLD' en 'SED' (resp. 'clear decimal mode' en 'set decimal mode') hebben we de zeer handige mogelijkheid een optelling decimaal of hexadecimaal uit te voeren. In bovenstaand programma hebben we 'CLD' gebruikt, en dus wordt de optelling hexadecimaal uitgevoerd. (Dit is voor de microprocessor de 'gewone' manier.) Wanneer we de instructie 'CLD' vervangen door 'SED' zal de optelling decimaal uitgevoerd worden:

```
0200 18       START CLC impl.
0201 F8             SED, impl.
0202 A5 00          LDA, Zpage GETAL 1
0204 65 10          ADC, page GETAL 2
0206 85 20          STA, Zpage GETAL 3
0208 4C 22 1C       JMP, abs. RST
```

Met 12 op adres $00, en 28 op adres $10 krijgen we nu 40 als resultaat op adres $20. De rest van het programma spreekt voor zich, op adres $0202 wordt GETAL 1 in de accumulator gezet. Op adres $0204 wordt de inhoud

van adres $10 (GETAL 2) bij het getal in de accumulator geteld. Het resultaat staat eveneens in de accumulator. Met 'STA' op adres $0206 wordt het antwoord naar adres $20 gebracht. Tot slot springen we weer naar het monitor programma, zodat het display oplicht, en we de toetsen weer kunnen gebruiken.

Het resultaat past niet op één geheugenplaats

Het bovenstaande programma biedt de mogelijkheid om 2 getallen op te tellen. Maar probeert u dit nu eens b.v. 86 en 95 ('86' op adres $00, en '95' op adres $10, en het programma starten op adres 0200). Het resultaat is 81, terwijl het 181 zou moeten zijn. De oorzaak van deze fout zal duidelijk zijn: het getal 81 wordt in het geheugen bewaard als 1000 0001, dit zijn 8 bits en voor de honderdtallen is geen ruimte meer. Het zou natuurlijk te gek zijn als de microprocessor geen grotere getallen zou kunnen verwerken. De honderdtallen gaan dan ook niet verloren, zij worden onthouden in het 'carry bit' van het status register. Dit bit wordt '1' als er bij een optelling een 'carry' ontstaat, en wordt '0' als er geen carry ontstaat ('carry' is engels voor 'één onthouden'). Eerder in het verhaal hebben we al vermeld dat de instructie 'ADC' de inhoud van een geheugen optelt bij de inhoud van de accumulator (= register A). Dit was echter niet volledig, ook de inhoud van het carry bit (0 of 1) wordt opgeteld bij register A. Hiermee is gelijk de eerste instructie van het programma verklaard, 'CLC' betekent nl. 'clear carry'. Omdat bij elke optelling ook het carry bit opgeteld wordt, zijn we verplicht dit carry bit eerst op 0 te zetten.

Wanneer we nu willen zorgen dat het programma ook de 100-tallen weergeeft, moeten we het antwoord uitbreiden tot 16-bits. De eenheden en de tientallen komen op adres $20 te staan, de honderdtallen op adres $21:

```
0200 18      START CLC, impl.
0201 F8            SED, impl.
0202 A5 00         LDA, Zpage GETAL 1
0204 65 10         ADC, Zpage GETAL 2
0206 85 20         STA, Zpage GET. E 3
0208 A9 00         LDA, imm. $00
020A 69 00         ADC, imm. $00
020C 85 21         STA, Zpage GET. H 3
020E 4C 22 1C      JMP, abs. RST
```

Als we nu '86' op adres $00 zetten, en 95 op adres $10, zal er na het uitvoeren van het programma '01' op adres $21 staan, en '81' op adres $20 het antwoord is dus 181. We kunnen natuurlijk ook hexadecimaal optellen door de instructie 'SED' op adres $0201 te vervangen door 'CLD', dus D8 i.p.v. F8. In dat geval krijgt u als antwoord:

```
$86 =    1000  0110
$95 =    1001  0101
         ---------- +
       10001  1011   = $11B
```

U ziet dat hexadecimale getallen het makkelijkst opgeteld kunnen worden door ze binair te schrijven. De computer geeft inderdaad op adres $20 het getal $1B, en op adres $21 het getal $01. Het $-teken voor de getallen geeft aan dat het hier hexadecimale getallen betreft, die in waarde **niet** gelijk zijn aan de decimale getallen.

Het programma is tot adres $0208 hetzelfde gebleven. Op adres $0208 zorgen we dat register A '00' wordt. Met de instructie 'ADC, imm. $00' wordt er bij de inhoud van register A de waarde '00' en de inhoud van het 'C'-bit geteld. Het uiteindelijke resultaat is dat register A de waarde '01' krijgt als het carry-bit '1' is, en de waarde '00' als het carry-bit '0' is. Op deze manier staan de honderdtallen nu in register A. Op adres 020C wordt deze inhoud op adres $21 weggezet. Ten overvloede nog even het verschil tussen 'ADC, Zpage $00' en 'ADC, imm. $00'. In het eerste geval wordt **de inhoud** van geheugenplaats '00' bij de inhoud van de accumulator (register A) geteld, in het tweede geval wordt **het getal** '00' bij de inhoud van register A geteld.

Het zichtbaar maken van de uitkomst

In bovenstaande voorbeelden rekent de computer steeds feilloos de antwoorden uit. Wanneer we dit antwoord willen zien, moeten we steeds het adres van het antwoord intypen. Mooier zou het zijn indien het antwoord meteen op het display verschijnt. Om dit te bereiken kunnen we gebruik maken van een subroutine uit het monitor programma, nl. 'SCANDS'. Deze subroutine verzorgt alle stuursignalen die nodig zijn om het display te sturen. Om van deze subroutine gebruik te maken moeten we de eenheden en tientallen op adres $F9 zetten, de honderdtallen en de duizendtallen op adres $FA, en tot slot de tienduizendtallen en de honderdduizendtallen op adres $FB. Wanneer de adressen F9... FB gevuld zijn met de juiste waarden, is de instructie 'JSR, abs. SCANDS' voldoende om het display voor enkele ms te laten werken. Door steeds opnieuw naar deze subroutine te springen blijft het display continu branden. Het programma wordt nu als volgt:

```
0200 18      START CLC impl.
0201 F8            SED, impl.
0202 A5 00         LDA, Zpage GETAL 1
0204 65 10         ADC, Zpage GETAL 2
0206 85 F9         STA, Zpage EENHED
0208 A9 00         LDA, imm. $00
020A 85 FB         STA, Zpage TIENDU
020C 69 00         ADC, imm. $00
020E 85 FA         STA, Zpage HONDER
0210 20 1F 1F DIS  JSR, abs. SCANDS
0213 4C 10 02      JMP, abs. DIS
```

We zien dat de antwoorden nu niet meer naar adres $20 gebracht worden, maar naar adres $F9. De honderdtallen komen op $FA i.p.v. $21. We zullen het programma nog even vanaf het begin doornemen.

Adres $0200. Hier wordt het carry-bit uit het statusregister op '0' gezet. Dit is nodig omdat bij de optelling verderop in het programma altijd óók het carry-bit opgeteld wordt. (Bij andere microprocessors is er vaak een aparte optelinstructie die het carry-bit niet meeneemt.)

Adres $0201. Hier wordt het decimal mode bit uit het statusregister op '1' gezet. Hierdoor zullen alle rekenkundige bewerkingen decimaal uitgevoerd worden. (Normaal: 2 + 8 = A nu: 2+ 8 = 10)

Adres $0202. Het eerste getal wordt in register A gezet, zodat straks het 2e getal erbij opgeteld kan worden.

Adres $0204. Het tweede getal wordt samen met de inhoud van het carry-bit opgeteld bij de inhoud van register A. Het resultaat blijft in register A. Het carry-bit was al eerder op '0' gezet, zodat nu de som van getal 1 en 2 in register A staat. Wanneer het resultaat meer dan 8 bits vraagt wordt ook het carry-bit op '1' gezet (dus als het resultaat ⩾ 100).

Adres $0206. Het antwoord (zonder de 100-tallen) wordt op adres $F9 gezet, zodat het later zichtbaar wordt op de twee rechter display's. Het carry-bit blijft ongemoeid.

Adres $0208. De inhoud van register A wordt '00' gemaakt.

Adres $020A. De inhoud van register A (00) wordt op adres $FB gezet. Hierdoor zullen de 2 linker display's straks de cijfers '00' weergeven.

Adres $020C. Bij de inhoud van register A (nog steeds '00') wordt de vaste waarde '00' met de waarde van het carry-bit geteld. Het resultaat zal '01' zijn als het carry-bit '1' was, en '00' als het carry-bit '0' was.

Adres $020E. Hier wordt de inhoud van register A op adres $FA gezet. Het resultaat zal zijn, dat straks op de middelste twee display's '00' of '01' verschijnt, afhankelijk van de toestand van het carry-bit.
Adres $0210. De inhoud van de adressen $FB, $FA en $F9 worden van links naar rechts op het display zichtbaar gemaakt gedurende enkele ms.
Adres $0213. Door de sprong naar adres $0210 wordt de subroutine 'SCANDS' eindeloos herhaald, zodat het antwoord continu op het display blijft staan. Alleen met 'RS of met 'ST' is het programma nog te onderbreken.

Het optellen van grotere getallen

Natuurlijk is het ook mogelijk getallen van meer dan 2 cijfers op te tellen. Door voldoende geheugenplaatsen aaneen te rijgen kunnen we getallen van 3, 4, 5 enz. getallen optellen. Een beperking is er nauwelijks. (De geheugenruimte is de enige beperkende factor.) Wanneer we alle geheugenplaatsen van pagina 0 zouden gebruiken, is het mogelijk om 2 getallen van 128 bytes op te tellen. Elke byte kan 2 cijfers bevatten (decimaal of hex) zodat met gemak getallen van 256 cijfers opgeteld kunnen worden. We hebben dan nog maar 1/2k geheugenruimte gebruikt. Natuurlijk is het zinloos om met zulke lange getallen te werken. In ons voorbeeld zullen we ons beperken tot getallen van 6 cijfers (past precies op het display). Eerst vragen we ons even af hoe u zelf zo'n getal zou optellen:

```
  598169
  139516
  ------ +
```

Bij deze optelling zouden we zelf als volgt handelen:

```
    9 + 6 = 15→ 5 opschrijven 1 onthouden
1 + 6 + 1 =  8→ 8 opschrijven
    1 + 5 =  6→ 6 opschrijven
    8 + 9 = 17→ 7 opschrijven 1 onthouden
1 + 9 + 3 = 13→ 3 opschrijven 1 onthouden
1 + 5 + 1 =  7→ 7 opschrijven
```

Aldus verkrijgen we het resultaat: 737685. Wellicht herkent u de werkwijze van de instructie 'ADC'. U heeft nl. net als bij deze instructie steeds de carry van de vorige optelling bij de nieuwe som opgeteld. De microprocessor werkt echter steeds met 2 cijfers tegelijkertijd:

```
0 + 69 + 16 = 85  carry = 0
0 + 81 + 95 = 76  carry = 1
1 + 59 + 13 = 73  carry = 0
```

U ziet, we moeten nu ook weer achteraan beginnen, en steeds de carry van de vorige optelling meenemen. We moeten er wel voor zorgen dat de carry de eerste keer '0' is. Het programma wordt:

```
0200 18       START CLC, impl.
0201 F8             SED, impl.
0202 A5 oo          LDA, Zpage GET 1, 1
0204 65 10          ADC, Zpage GET 2, 1
0206 85 F9          STA, Zpage EENHED
0208 A5 01          LDA, Zpage GET 1, 2
020A 65 11          ADC, Zpage GET 2, 2
020C 85 FA          STA, Zpage HONDER
020E A5 02          LDA, Zpage GET 1,3
0210 65 12          ADC, Zpage GET 2, 3
0212 85 FB          STA, Zpage TIENDU
0214 20 1F 1F DIS   JSR, abs. SCANDS
0217 4C 14 02       JMP, abs DIS
```

De op te tellen getallen zijn elk over 3 geheugenplaatsen verdeeld. Als we even bij de beide getallen uit het voorbeeld blijven (598169+ 139516) moeten de getallen als volgt in het geheugen gezet worden:

adres	inhoud	
00	69	} getal 1
01	81	
02	59	
10	16	} getal 2
11	95	
12	13	

We hebben dus 3 bytes aaneen geregen. De optelling bestaat steeds uit 3 gedeelten, nl. een deel van getal 1 ophalen, het overeenkomstige deel van getal 2 optellen en het antwoord wegzetten. Deze 3 stappen moeten herhaald worden totdat het hele getal is opgeteld. Het programma spreekt verder voor zich.
De invoer van de getallen kan natuurlijk veel mooier. We kunnen b.v. een hulpprogramma schrijven die de ingetoetste getallen zichtbaar maakt, net als een rekenmachine. Natuurlijk is het ook mogelijk een complete rekenmachine te programmeren. Dergelijke programma's worden echter vrij lang, en vallen daardoor buiten het bestek van deze artikelenreeks. Misschien besteden we hier t.z.t. een apart artikel aan.

Het aftrekken van 2 binaire getallen

Het aftrekken van 2 binaire getallen is in principe niet moeilijker dan optellen. Ook hier bestaat een aparte instructie voor; nl. 'SBC'. Met deze instructie trekken we de inhoud van een geheugenplaats af van de inhoud van de accumulator. De codes van 'SBC' voor de tot nu toe besproken adresseringsmethoden:

instr.	imm.	abs.	Zpage
SBC	E9	ED	E5
Zp,X	abs,X	abs,X	
F5	FD	F9	

In het nu volgende voorbeeld gaan we er weer vanuit, dat beide van elkaar af te trekken getallen op adressen $00 en $10 staan:

```
0200 38       START SEC, impl.
0201 D8             CLD, impl.
0202 A5 00          LDA, page GETAL 1
0204 E5 10          SBC, Zpage GETAL 2
0206 85 20          STA, Zpage GETAL 3
0208 4C 22 1C       JMP, abs RST
```

Omdat de inhoud van het geheugen wordt afgetrokken van de inhoud van de accumulator zal nu getal 2 afgetrokken worden van getal 1 (getal 1 hebben we immers in de accumulator gezet). We zetten nu '0A' op adres $00, en '08' op adres $10. Na het uitvoeren van het programma zal er '02' op adres $20 staan. Dit is inderdaad het verschil tussen $0A (= 10) en $08 (= 8).
Nog even over het programma: We zien nu i.p.v. 'CLC' de instructie 'SEC' (set carry). Bij het aftrekken moet het carry-bit nl. '1' zijn. Waarom dit zo is zal verderop in het verhaal blijken.

Negatieve getallen

Waar we tot nog toe niet over gesproken hebben zijn de negatieve getallen. Er zijn een aantal manieren om negatieve getallen binair weer te geven. De meest gebruikte is de notatie volgens het 'two's complement'. We zullen ons tot deze notatie beperken, omdat hij bij vrijwel alle microprocessors wordt toegepast. Om b.v. het getal $-67 volgens het two's complement weer te geven moeten we het hexadecimale getal eerst binair schrijven:

$$\$-67 \rightarrow 0110\ \ 0111$$

vervolgens moet elk bitje van het getal geïnverteerd worden:

$$0110\ \ 0111 \rightarrow 1001\ \ 1000$$

Als laatste stap moet er 1 bij het resultaat worden opgeteld:

$$1001\ \ 1000 + 1 = 1001\ \ 1001$$

hexadecimaal geschreven wordt dit $99.
Wanneer we met negatieve getallen werken betekent $99 **altijd** $-67. De notatie heeft veel voordelen, b.v. wanneer we getallen gaan optellen. Stel dat we het getal $69 willen optellen bij het getal $-67.

In de computer staat $-67 genoteerd als $99:

```
$69     01101001
$99     10011001
        --------  +
       100000010
```

Het antwoord is dus 0000 0010 hetgeen overeenkomt met $02. (Het carry-bit is '1' geworden.) We zien dus dat we dankzij deze notatie zowel negatieve als positieve getallen gewoon kunnen optellen. Ook als het antwoord negatief wordt komt het goede getal tevoorschijn:

```
          $65   01100101
$-67 = $99      10011001
                --------  +
                11111110
```

Dit antwoord zou dus het two's complement van $-02 moeten zijn, hetgeen inderdaad klopt.
We zitten nu nog met één probleem, hoe kun je een negatief getal herkennen? Met andere woorden is b.v. het getal $99 nu écht $99 of is het de two's complement notering van $-67!

Om een éénduidige aanduiding te verkrijgen is daarom afgesproken dat bit 7 (in onze voorbeelden het meest linkse bit) bepalend is voor het teken. Wanneer bit 7 = 0 is het getal positief, en wanneer bit 7 = 1, is het getal negatief. Het gevolg hiervan is, dat getallen groter dan $7F niet meer met 1 byte weergegeven kunnen worden. Het grootste negatieve getal is $80. Decimaal loopt het bereik van 1 byte nu van −128...+127:

decimaal:	hexa-decimaal:	binair:
127	7F	01111111
126	7E	01111110
.	.	.
.	.	.
.	.	.
2	02	00000010
1	01	00000001
0	00	00000000
−1	FF	11111111
−2	FE	11111110
.	.	.
.	.	.
.	.	.
−127	81	10000001
−128	80	10000000
		↑ teken bit

Het carry-bit

We kunnen het optellen van een positief en een negatief getal ook opvatten als het aftrekken van twee getallen. De optelling $69 + $99 zal dan ook hetzelfde resultaat moeten hebben als de aftrekking $69 − $67. Het getal $99 is immers de two's complement notatie van $-67. Bij de optelling $69 + $99 werd het carry-bit op '1' gezet, zodat ook bij de aftrekking $69 − $67 het carry-bit op '1' wordt gezet. Bij de optelling $65 + $99 werd het carry-bit op '0' gezet, en dus zal ook bij de aftrekking $65 − $67 het carry-bit op '0' gezet moeten worden. We zien dus dat het carry-bit '0' wordt als het resultaat negatief is, óf (wat op hetzelfde neerkomt) indien we geleend hebben van een volgend byte. Als voorbeeld gaan we weer twee getallen van 6 cijfers van elkaar aftrekken:

```
598169
139516
------  −
```

Wanneer we zelf de berekening uitvoeren:

```
    9 − 6 = 3
    6 − 1 = 5
    1 − 5 kan niet, één lenen:
    11 − 5 = C (hex!!)
(8 − 1) − 9 kan niet, één lenen:
    17 − 9 = E (hex!!)
(9 − 1) − 3 = 5
    5 − 1 = 4
```

het antwoord is dus 45EC53.
Zoals u ziet hebben we de aftrekking hexadecimaal uitgevoerd. De computer rekent nu als volgt:

```
69 − 19 − 0 = 53 carry-bit = 1 dus geen borrow
81 − 95 − 0 = EC carry-bit = 0 dus wél borrow
59 − 13 − 1 = 45 carry-bit = 1 dus géén borrow
```

'Borrow' is het Engels voor 'één lenen'. We zien dat als het carry-bit '1' is, er niet geleend hoeft te worden, en als het carry-bit '0' is, er wél geleend moet worden (de uitkomst van de middelste bewerking 'EC' is het two's complement van −14!!).

Terug naar de computer

Het computerprogramma van deze aftrekking ziet er als volgt uit:

```
0200 38     START SEC, impl.
0201 D8           CLD, impl.
0202 A5 00        LDA, Zpage GET 1, 1
0204 E5 10        SBC, Zpage GET 2,1
0206 85 F9        STA, Zpage EENHED
0208 A5 01        LDA, Zpage GET 1, 2
020A E5 11        SBC, Zpage GET 2,2
020C 85 FA        STA, Zpage HONDER
020E A5 02        LDA, Zpage GET 1,3
0210 E5 12        SBC, Zpage GET 2,3
0212 85 FB        STA, Zpage TIENDU
0214 20 1F 1 F DIS JSR, abs. SCANDS
0217 4C 14 02     JMP, abs. DIS
```

U ziet, dat het programma erg veel lijkt op het optelprogramma voor twee getallen van 6 cijfers. De borrow (= carry) wordt door de 'SBC' instructie automatisch meegenomen. Dit houdt in dat de eerste keer de carry op '1' gezet moet worden (C = '1' betekent: 'niet geleend'). Wanneer u voorgaande berekeningen door de computer wilt laten uitvoeren moet dit weer als volgt ingevoerd worden:

adres	inhoud	
00	69	
01	81	} getal 1
02	59	
10	16	
11	95	} getal 2
12	13	

Wanneer u beide getallen omwisselt, wordt het uiteindelijke resultaat: BA13AD, hetgeen het two's complement is van −45EC53:

```
BA13AD:        1011 1010 0001 0011 1010 1101
inverteren:    0100 0101 1110 1100 0101 0010
1 optellen:    0100 0101 1110 1100 0101 0011
terug naar
hex :             4    5    E    C    5    3
                  ↑
                teken bit
```

Hoewel het 24-bits getal BA13AD in het geheugen 3 bytes in beslag neemt, is alleen het allereerste bit van deze 3 bytes achterelkaar bepalend voor het teken.

Decimaal aftrekken

We kunnen, door de 'decimal mode flag' op '1' te zetten, dezelfde bewerking ook decimaal uitvoeren. Als antwoord krijgen we dan:

598169 − 139516 = 458653

(u hoeft alleen op adres $0201 'F8' te zetten i.p.v. 'D8'). Wanneer het antwoord van een getal negatief is, gaat het echter niet meer zo mooi. De two's complement notatie gaat niet meer op. Wanneer we de getallen omgekeerd aftrekken krijgen we:

139516 − 598169 = 541347

We kunnen wel weer achter de juiste waarde komen door het 'ten's complement' te nemen:

getal	541347
inverteren (aanvullen tot 9)	458652
1 optellen	458653
antwoord	−458653

U ziet, dat de gevolgde methode vrijwel identiek is, maar voor de computer wordt het een stuk moeilijker. Ten eerste is het nu niet meer zo eenvoudig om te bepalen of een getal negatief is. Ten tweede is het ten's complement ook niet eenvoudig te vinden. (Het two's complement is voor de microprocessor één instructie.) Daarom wordt, wanneer we decimaal rekenen, vaak het teken apart onthouden. Een speciale routine zorgt er dan voor dat altijd het kleinste getal van het grootste wordt afgetrokken, terwijl een andere routine er voor zorgt dat het antwoord weer het goede teken krijgt. Decimaal rekenen heeft dus het voordeel dat we niet van hex naar decimaal en omgekeerd hoeven te rekenen, maar het nadeel dat negatieve getallen wat moeilijker te verwerken zijn. Hierdoor zullen berekeningen in het hexadecimale stelsel ook wat sneller verlopen.

Het optellen van 2 negatieve getallen

Hiervoor gaven we al een klein programmaatje voor het optellen van twee 8-bits getallen:

```
0200 18      START  CLC, impl
0201 D8             CLD, impl
0202 A5 00          LDA, page   GETAL 1
0204 65 10          ADC, Zpage  GETAL 2
0206 85 20          STA, Zpage  GETAL 3
0208 4C 22 IC       JMP, abs    RST
```

Dit programma telt een getal dat op geheugenplaats $00 staat op bij een getal dat op plaats $10 staat. Het antwoord komt op adres $20 te staan. Wanneer we $0F op adres $00 zetten, en $''07'' op adres $10, zal er na het uitvoeren van het programma $16 op adres $20 staan. ($0F +$07 = $16) **Een en ander is reeds uitvoerig toegelicht. We hebben toen ook gezien dat we negatieve getallen gewoon kunnen optellen mits ze geschreven staan in het two's complement. Wanneer we dus $-0F en $-07 willen optellen moeten we eerst deze getallen omzetten naar het two's complement:**

```
binair geschreven:   0F → 00001111
inverteren:          → 11110000
één optellen:        → 11110001
terug naar hex:      → F1
```

Op dezelfde manier vinden we voor $-07 het two's complement $F9. We zetten nu $F1 (-0F) op adres $00, en $F9 (-07) op adres $10. Als antwoord krijgen we $EA op adres $20. Dit moet dus de two's complementnotatie van het antwoord zijn. Het terug gaan naar de voor ons beter leesbare notering met – teken verloopt op dezelfde manier als de omzetting naar het two's complement:

```
binair schrijven:    EA → 11101010
inverteren:          → 00010101
één optellen:        → 00010110
terug naar hex:      → 16
```

Het antwoord is dus $-16, hetgeen inderdaad klopt. U zult zich misschien afvragen waarom we zo ingewikkeld doen. We kunnen immers ook gewoon $0F en $07 optellen en even onthouden dat het hier om negatieve getallen gaat. Het voordeel van two's complementnotatie is dat we zonder uitzondering door elkaar positieve én negatieve getallen kunnen optellen en aftrekken.

Nog een voorbeeld

We gaan nog even door met het bovenstaande voorbeeld. Stel dat we de getallen $-70 en $-13 willen optellen. Het resultaat moet dus $-83 worden. Eerst het two's complement berekenen:

$-70 → $90
$-13 → $ED

Wanneer we deze getallen op adressen $00 en $10 zetten, zullen we na het uitvoeren van het programma het getal $7D op adres $20 vinden. Nog even terugrekenen:

```
binair schrijven:    7D → 01111101
inverteren:          → 10000010
één optellen:        → 10000011
terug naar hex:      → 83
```

En we zien dat het antwoord inderdaad klopt, tenminste, dat lijkt zo!! Als $7D inderdaad opgevat wordt als een negatief getal krijgen we inderdaad $-83 als antwoord. Maar in het vorige deel hebben we gezien dat bit 7 (het linker bitje) '1' moet zijn als een getal negatief is, en '0' als een getal positief is. Bij het getal $7D heeft bit 7 de waarde '0', zodat $7D door de microprocessor wordt opgevat als een *positief* getal.

De overflow flag

Het zal duidelijk zijn dat fouten, zoals hier boven niet onopgemerkt mogen blijven. Want wat is er eigenlijk aan de hand? Met één byte kunnen we 256 verschillende bitcombinaties maken (2^8). Wanneer we niet met negatieve getallen werken kunnen we dus alle getallen tussen 0 en 255 weergeven. Wanneer we met positieve én negatieve getallen werken ligt dit bereik tussen −128 en +127, of hexadecimaal geschreven tussen $-80 en $7F. Met $-83 zijn we deze grens gepasseerd, waardoor het antwoord als positief beschouwd werd. Wanneer deze situatie zich voordoet wordt in de microprocessor de overflowflag geset. Hierdoor krijgen we de mogelijkheid om een foutmelding te geven, of om een correctie toe te passen.

Voorbeeld:

We zullen nu het eerder gegeven programma uitbreiden met een foutmelding als er een overflow ontstaat.

```
0200 18      START  CLC, impl
0201 D8             CLD, impl
0202 A5 00          LDA, page   GETAL 1
0204 65 10          ADC, Zpage  GETAL 2
0206 70 FE   FOUT   BVS, rel    FOUT
0208 85 20          STA, Zpage  GETAL 3
020A 4C 22 1C  .    JMP, abs    RST
```

Wanneer we de getallen $60 en $10 willen optellen (dus 2 positieve getallen) zal er na het uitvoeren van het programma $70 op adres $20 staan. Niets aan de hand dus. Wanneer we $60 en $20 willen optellen zal het display doven. ($60 op adres $00, $20 op adres $10, programma starten op adres $0200). Op deze manier maakt het programma ons duidelijk dat er een overflow is ontstaan. Met 'RS' is het programma weer te stoppen. Alvorens op de werking van het programma in te gaan, maken we eerst even duidelijk waarom er een overflow ontstond, we telden nu immers twee positieve getallen op!

Het verwachte antwoord zou moeten zijn; $20 + $60 = $80. Wanneer we alleen met positieve getallen werken is dit antwoord goed, en hoeven we helemaal niet op de overflowflag te letten. Wanneer we echter in één programma onderscheid maken tussen positieve én negatieve getallen, wordt het getal opgevat als een *negatief* getal. Dit omdat bit 7 de waarde '1' heeft ($80 = 1000 0000). Het komt er dus op neer dat we weer over de beschikbare capaciteit ($-80...$+7F) gekomen zijn, zodat de overflow werd geset. In het programma zien we dat op adres $0206 en $0207 de instructie 'BVS, rel FOUT' is tussen gevoegd. BVS is de afkorting van 'Branch if oVerflow Set', of in Nederlands: 'maak een sprong als het overflowbit 1 is'. De sprongwaarde is zo ingevuld dat steeds weer opnieuw naar deze instructie teruggesprongen wordt.

Hierdoor zal het programma nooit naar

de monitor terugspringen, en het display blijft donker. Natuurlijk kunnen we ook naar een ander adres springen, waar bijv. een wat elegantere foutmelding wordt verzorgd.

Het verschil tussen 'overflow' en 'carry'

In het vorige deel zijn we uitvoerig ingegaan op de carryflag. U kunt zich misschien herinneren dat de carryflag ook geset werd als een resultaat niet meer in één byte paste. Er zijn echter duidelijke verschillen:
De *carry*flag wordt geset als het antwoord van een optelling groter wordt dan 255. Het resultaat past dan niet meer in een byte.
De carry krijgt hier de functie van 'één onthouden' bij een optelling. Ook bij aftrekken wordt het carrybit gebruikt als 'borrow' of 'één lenen'. Door de 'carry' zijn we in staat om bytes aan een te rijgen tot 16, 24, 32 enz. bits getallen. De *overflow*flag geeft aan dat het tekenbit verminkt is doordat het antwoord groter dan $7F, of kleiner dan $-80 werd. Wanneer we uitsluitend met positieve getallen werken hoeven we niet op de overflow te letten. Wanneer we met positieve en negatieve getallen werken, langer dan 1 byte, hoeven we alleen bij het meest significante byte (dus het byte waarin zich het tekenbit bevindt) op de overflow te letten.

Vermenigvuldigingen

We zijn nu toegekomen aan het vermenigvuldigen van twee getallen. Verreweg de eenvoudigste manier om te vermenigvuldigen is herhaald optellen. Wanneer we bijv. 5 × 6 willen uitrekenen herleiden we dit tot 5 + 5 + 5 + 5 + 5 + 5 (zo heeft u zelf ook vermenigvuldigen geleerd). Het programma voor deze manier van vermenigvuldigen is heel eenvoudig:

```
0200 F8       START  SED, impl
0201 18              CLC, impl
0202 A6 00           LDX, Zpage   GETAL 1
0204 A9 00           LDA, imm     $00
0206 65 10    TELOP  ADC, Zpage   GETAL 2
0208 CA              DEX, impl
0209 DO FB           BNE, rel     TELOP
020B 85 20           STA, Zpage   GETAL 3
020D 4C 22 1C        JMP, abs     RST
```

Zoals bij alle optelprogramma's beginnen we weer met het aangeven of decimaal dan wel hexadecimaal willen rekenen. Vervolgens wordt de carry op '0' gezet, en de eigenlijke vermenigvuldiging gaat plaats vinden. Eén van beide getallen wordt in indexregister X gezet (adres $0202). De inhoud van de accumulator wordt $00 gemaakt, en in een loop (adressen $0206...$0209) wordt nu het tweede getal net zoveel bij register A geteld als door het indexregister (eerste getal) wordt aangegeven. Het antwoord wordt nu op adres $20 weggezet.
Hiermede is het programma klaar. Deze manier van vermenigvuldigen behoeft wel wat kanttekeningen. Het bovenstaande programma kan maximaal een 8 bits antwoord onthouden, hetgeen inhoudt dat het antwoord nooit groter kan worden dan 99 (als we decimaal werken) of FF (als we hexadecimaal werken). Deze capaciteit kan natuurlijk uitgebreid worden tot 16, 24, 32 enz. bits. We hebben dan wel een erg langzame vermenigvuldiging. Eén keer door de loop kost 3 + 2 + 3 = 8 µs. Bij het vermenigvuldigen van bijv. 9 × 9 duurt de loop dus 9 × 8 = 72 µs. De loop moet immers negen maal worden doorlopen.
Stelt u zich nu eens voor dat een getal van vier of vijf cijfers vermenigvuldigd zou moeten worden. Ten eerste moeten in de loop meer instructies worden opgenomen, omdat er nu méér bytes opgeteld moeten worden. Ook kunnen we register X niet meer gebruiken omdat we hierin hooguit 8 bits kwijt kunnen. Dus één keer door de loop duurt al langer, maar bovendien moeten we de loop nu zo'n slordige 10.000 keer doorlopen. Een beetje vermenigvuldiging komt zo al gauw in de tientallen ms! Wanneer u een rekenmachine programmeert is 10 ms voor een vermenigvuldiging geen bezwaar. Bij reeks ontwikkelingen, het berekenen van grafieken enz. is 10 ms veel te langzaam. We moeten dus naar een andere methode zoeken.

Een andere methode om te vermenigvuldigen

Wanneer u zelf een vermenigvuldiging uit moet rekenen, gaat u als volgt te werk:

```
  345
  123 ×
 ----
 1035
 6900
34500
-----
42435
```

Wat doet u nu precies? Eerst neemt u het bovenste getal 3 ×, en het antwoord schrijft u in een kolom die u straks gaat optellen. Vervolgens schrijft u een 0 op, en berekent u 2 × 345. Doordat u de 0 al opgeschreven had berekende u eigenlijk 2 × 3450. Hetzelfde geldt voor het laatste getal, u begint nu met het opschrijven van twee nullen.
De computer laten we nu precies hetzelfde doen. Zoals u weet werkt de computer in het tweetallig stelsel, zodat we nu binair moeten vermenigvuldigen. Als voorbeeld nemen we nu twee eenvoudige getallen. Grotere getallen worden op exact dezelfde wijze behandeld.

Stel:

```
1110    (14)
1001    ( 9)
---- ×
```

Voor we beginnen moeten we eerst de tafels van het tweetallig stelsel kennen. Een prettige bijkomstigheid is dat de tafels erg kort zijn. De tafel van 0:

0 × 0 = 0
1 × 0 = 0

De tafel van 1:

0 × 1 = 0
1 × 1 = 1

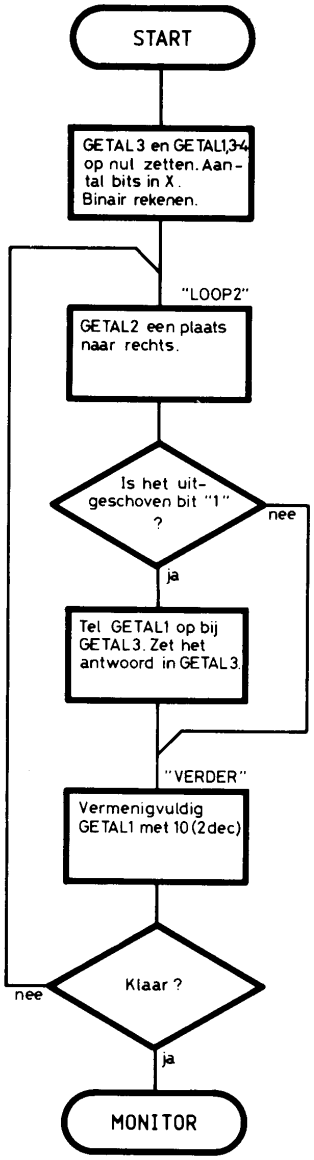
En meer cijfers zijn er niet. Al met al komt het er op neer dat we bij een '0' het getal niet optellen en bij een '1' het getal wel optellen. Wij zouden de vermenigvuldiging als volgt maken:

```
      1110 (= $E = 14)
      1001 (= $9 =  9)
-     ---- ×
      1110
     00000
    000000
   1110000
   -------- +
   1111110     (= $7E = 126)
```

De computer werkt op dezelfde manier, alleen het optellen van 0000 slaat hij over:

```
      1110
      1001
      ---- ×
      1110
   1110000
   -------- +
   1111110
```

We moeten nu alleen nog bepalen *hoe* we deze methode omzetten in 'computertaal'. Om te beginnen moeten we de verschillende getallen een naam geven. Zoals we al eerder gedaan hebben noemen we de twee te vermenigvuldigen getallen 'GETAL 1' en 'GETAL 2', en het antwoord 'GETAL 3'. Het aantal bits dat elk getal heeft laten we nog even in het midden. De tussenantwoorden, die we zelf steeds onder elkaar schreven, laten we de computer



Het flowdiagram voor de 16-bits vermenigvuldiging.

In deze tabel kunt u precies volgen welke acties de microprocessor tijdens elke loop onderneemt.

Tabel 2

Loop nr	reg X	GETAL2 wordt	carry bit	GETAL1 was	GETAL3 was	GETAL3 wordt	GETAL1 wordt
1	4	0100	1	1110	00000000	00001110	11100
2	3	0010	0	11100	00001110	00001110	111000
3	2	0001	0	111000	00001110	00001110	1110000
4	1	0000	1	1110000	00001110	01111110	11100000

gelijk optellen. Het optellen moet toch gebeuren en als we het gelijk doen wordt geheugenruimte uitgespaard. De eerste actie moet dus zijn het op nul zetten van 'GETAL 3', zodat we later de tussen antwoorden hierbij kunnen optellen.
We volgen nu letterlijk de procedure die wij ook gevolgd hebben:
'Indien bit 0 van getal 2 één is, getal 1 bij getal 3 tellen'. Bit 0 van getal 2 is inderdaad 1, dus getal 3 is nu '1110'.
De volgende opdracht zal luiden:
'Indien bit 1 van getal 2 één is, 10 × getal 1 bij getal 3 tellen'. Omdat bit 1 van getal 1 '0' is, zal er niets bij getal 3 geteld worden. Op deze manier moeten we doorgaan totdat het hele getal vermenigvuldigd is. Wellicht ontdekt u een grote gelijkenis tussen beide laatste opdrachten. Het enige verschil is dat we de eerste keer naar bit 0 van getal 2 kijken, de tweede keer naar bit 1, en de n^e keer naar bit n-1. Bovendien moeten de eerste keer 1 × getal 1 optellen, de tweede keer 10 × getal 1 (10 is een binair getal), de derde keer 100 × getal 1, en de n^e keer 10^{n-1} (of decimaal 2^{n-1}) × getal 1.

De flowchart

Om een duidelijk inzicht te krijgen van de volgorde van de verschillende instructies is het nodig een flowchart te **maken. Afb. 4 geeft de flowchart van ons probleem. We moeten GETAL 3 eerst '0' maken.**
We tellen straks nl. steeds GETAL 1 bij GETAL 3. De totale som geeft het antwoord. Om te voorkomen dat straks de oude inhoud van getal 3 bij het antwoord is geteld moeten we beginnen met het '0' maken van GETAL 1. Ook twee hulpregisters moeten '0' gemaakt worden, hier komen we nog op terug. De loop moet voor ieder bitje van het te vermenigvuldigen getal een keer worden doorlopen. We moeten dus het aantal bitjes in reg. X zetten. Na elke loop wordt register X met één verlaagd. Zolang het resultaat geen '0' is gaan we terug.
Omdat deze methode alleen binair werkt, moet de 'decimal mode flag' worden gecleard. Na deze voorbereidende werkzaamheden (ook wel 'inializeren' genoemd) gaat het vermenigvuldigen beginnen. Allereerst schuiven we getal 2 één plaats naar rechts. Hierdoor komt bit 0 in het carry bit van het statusregister. Als dit bitje '0' is springen we over de telopdracht heen. Als het bitje 1 is wordt 'GETAL 1' bij het antwoord geteld. Als voorbereiding op de volgende loop wordt nu getal 1 met 10 (binair) vermenigvuldigd. Het vermenigvuldigen met 2 in het 2-tallig stelsel is net zo eenvoudig als het vermenigvuldigen van 10 in het 10-tallig stelsel. Het getal moet gewoon 1 plaats naar links worden geschoven:
decimaal 123 × 10 = 1230
binair 1001 × 10 = 10010
(9 × 2 = 18)
Hoe het programma nu loopt ziet u in tabel 2 in deze tabel is voor elke loop aangegeven hoe groot elk getal is.

Het programma

Wanneer we het programma gaan schrijven moeten we het aantal bits vaststellen. Als voorbeeld nemen we voor getal 1 en 2 twee bytes, zodat 2 16-bits getallen kunnen worden vermenigvuldigd. Het antwoord (GETAL 3) neemt dan maximaal 4 bytes in beslag. Omdat GETAL 1 steeds een plaats naar links wordt geschoven moet GETAL 1 net zoveel ruimte hebben als het antwoord, getal 3 (zie tabel 1). De extra ruimte die getal 1 nodig heeft, moet ook vóór het vermenigvuldigen '0' gemaakt worden (dit zijn de eerder genoemde hulpregisters). Voor de getallen nemen we de volgende geheugenplaatsen:

getal 1,1 00
getal 1,2 01
getal 1,3 02
getal 1,4 03

getal 2,1 10
getal 2,2 11
getal 3,1 20
getal 3,2 21
getal 3,3 22
getal 3,4 23

Hierbij nemen we aan dat het minst significante digit (LSD) zich in getal X,1 bevindt. Dat wil dus zeggen dat bijv. bij $3CF5 $F5 in getal 1,1 komt te staan, en $3C in getal 1,2. We gaan nu blokje voor blokje het flowdiagram omzetten in programmastapjes.

Het inializeren

Om te beginnen moeten we getal 2,1 2,2 2,3 en 2,4 '00' worden gemaakt. In het programma hebben we dit met behulp van een loop gedaan, omdat dit iets minder ruimte vraagt dan:

```
LDA,  imm    $00
STA,  Zpage  GETAL 3,1
```

```
STA,  Zpage  GETAL 3,2
STA,  Zpage  GETAL 3,3
STA,  Zpage  GETAL 3,4
```

Door de sprong die steeds gemaakt moet worden (adres $0207) duurt het wel iets langer dan de 'recht toe recht aan' methode. Op de totale vermenigvuldigingstijd maakt dit echter niets meer uit. Op adres $0209 en $020B wordt getal 1,2 en 1,4 nog '00' gemaakt. Na 'CLD' en 'LDX' zijn we klaar met het voorbereidende werk. (Met deze instructies zorgen we dat er binair wordt opgeteld, en dat het hele proces voor 16 bitjes (10 hexadecimaal) wordt herhaald.

Getal 2 één plaats naar rechts

Wanneer GETAL 2 maar uit één byte zou bestaan zou deze opdracht slechts één instructie zijn:

```
LSR, Zpage GETAL2
```

Deze instructie zijn we nog niet eerder tegen gekomen. De letters staan voor 'Logic Shift Right'. Deze instructie zorgt dat alle bitjes van de geadresseerde geheugenplaats één plaats naar rechts schuiven. Bit 0 schuift als het ware uit de geheugenplaats, en wordt in de carry bewaard. Bit 7 schuift naar positie 6 en op positie 7 komt een 0. Dus:

voor schuiven:
B7 B6 B5 B4 B3 B2 B1 B0
na schuiven:
0 B7 B6 B5 B4 B3 B2 B1

B0 bevindt zich in het carrybit van het statusregister.
Wanneer een getal langer dan één byte is, moet het uitgeschoven bit in het volgende byte worden geschoven. **Dus GETAL 2 ziet er vóór het schuiven zo uit, zie afb. 5.**
Bit 8 is niet verloren gegaan, maar is zoals eerder gezegd in het carrybit van het statusregister terecht gekomen. We moeten nu GETAL 2,1 een plaats naar rechts schuiven. Op positie 7 moet nu echter geen '0' komen, maar de inhoud van het carrybit (B8). Bovendien moet ook het uitgeschoven bit 0 bewaard blijven. Deze actie wordt uitgevoerd door een tweede nieuwe instructie, nl. 'ROR' hetgeen staat voor Rotate Right. Bij deze instructie schuift de inhoud van het carrybit in positie 7, en schuift B0 weer in het carrybit. De bits 'roteren' dus. Na de instructie **'ROR Zpage GETAL 2,1' ziet de situatie er als volgt uit (zie afb. 6).**

U ziet, het hele getal is één positie naar rechts geschoven, en B0 bevindt zich in het carrybit. Helaas zit de ROR-instructie niet in alle 6502's. In de oude types (vóór juni 1976) ontbreekt de ROR.
Indien u een ouder type heeft kunt u

```
LSR,  Zpage  GETAL 2,2
ROR,  Zpage  GETAL 2,1
```

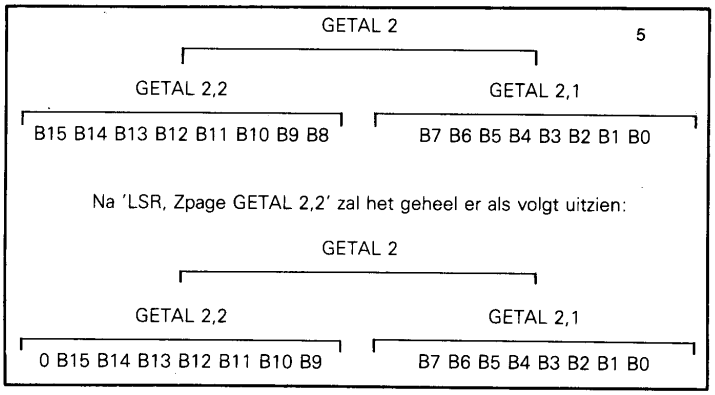
vervangen door:

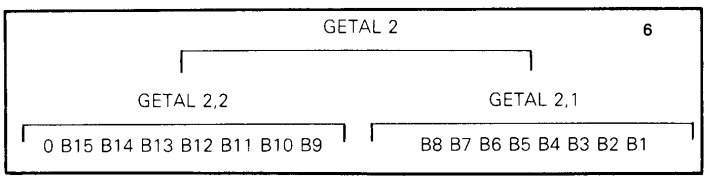
```
LDA,  imm    $80
LSR,  Zpage  getal 2,2
BCS,  rel    $01
ASL,  accu
LSR,  Zpage  GETAL 2,1
ORA,  Zpage  GETAL 2,1
STA,  Zpage  GETAL 2,1
```

LIJST 3

```
0200 A2 04      START  LDX,  imm   $04
0202 A9 00             LDA,  imm   $00
0204 95 1F      LOOP1  STA,  Zp, X GETAL 3,1-1
0206 CA                DEX,  impl
0207 D0 FB             BNE,  rel   LOOP1        Diverse geheugenplaatsen '0' maken.
0209 85 02             STA,  Zpage GETAL 1,3
020B 85 03             STA,  Zpage GETAL 1,4
020D D8                CLD,  impl               Binair rekenen
020E A2 10             LDX,  imm   $10          16 bitjes
0210 46 11      LOOP2  LSR,  Zpage GETAL 2,2    GETAL 2 één plaats naar
0212 66 10             ROR,  Zpage GETAL 2,1    rechts
0214 EA EA             NOP,  impl
0216 EA                NOP,  impl               9×NOP, zodat eventueel
0217 EA EA             NOP,  impl               de ROR-instructie ge-
0219 EA EA             NOP,  impl               simuleerd kan worden
021B EA EA             NOP,  impl               (zie tekst)
021D 90 19             BCC,  rel   VERDER       Indien C=1 :optellen
021F 18                CLC   impl
0220 A5 00             LDA,  Zpage GETAL 1,1
0222 65 20             ADC,  Zpage GETAL 3,1
0224 85 20             STA,  Zpage GETAL 3,1
0226 A5 01             LDA,  Zpage GETAL 1,2
0228 65 21             ADC,  Zpage GETAL 3,2
022A 85 21             STA,  Zpage GETAL 3,2    32-bits optelling
022C A5 02             LDA,  Zpage GETAL 1,3
022E 65 22             ADC,  Zpage GETAL 3,3
0230 85 22             STA,  Zpage GETAL 3,3
0232 A5 03             LDA,  Zpage GETAL 1,4
0234 65 23             ADC,  Zpage GETAL 3,4
0236 85 23             STA,  Zpage GETAL 3,4
0238 06 00      VERDER ASL,  Zpage GETAL 1,1
023A 26 01             ROL,  Zpage GETAL 1,2    Schuif getal 1 één
023C 26 02             ROL,  Zpage GETAL 1,3    plaats naar links
023E 26 03             ROL,  Zpage GETAL 1,4
0240 CA                DEX,  impl               Is het laatste bitje
0241 D0 CD             BNE,  rel   LOOP2        geweest?
0243 4C 22 1C          JMP,  abs   RST          terug naar monitor
```

Het uiteindelijke programma van de 16-bits vermenigvuldiging. Het resultaat telt 32-bits.





Door de NOP's in lijst 3 (adressen 0214 . . . 021C) is er in het programma voldoende ruimte om deze instructies te plaatsen. Op de werking van dit stukje programma gaan we niet in, omdat het met het vermenigvuldigen niets heeft te maken.

Is het uitgeschoven bitje '1'?

In het volgende blokje wordt er een beslissing genomen. Indien het uitgeschoven bitje van getal 2 een '1' was wordt er wel opgeteld, indien dit bitje '0' was wordt er niet opgeteld. Omdat het uitgeschoven bitje zich in het carrybit bevindt kunnen we met:

BCC, rel (adres $021D)

bereiken wat we willen. Deze instructie verzorgt een sprong (Branch if Carry Clear) als het carrybit '0' is. De spronggrootte is zo gekozen dat we precies over de optelling heen springen. Als het carrybit '1' is, wordt de sprong niet uitgevoerd, en wordt er dus wel opgeteld.

De 32-bits optelling

Op de adressen $021F...$0236 staat de optelling GETAL 1 + GETAL 3. Het antwoord wordt weer in GETAL 3 geschreven. Het principe van deze 'meerbytes' optellingen is in het vorige deel van 'stap voor stap' uitvoerig behandeld. Het is mogelijk deze lange rij instructies in een loop te zetten, en met de Zpage, X te werken. Het resultaat zal zeker een besparing van geheugenruimte geven. Zoals al eerder gezegd kost een loop wel meer tijd, omdat steeds register X veranderd moet worden, en omdat steeds terug gesprongen moet worden. Dit gedeelte van het programma staat echter ook in een loop en wordt 16 × uitgevoerd. Hierdoor zal de extra tijd, nodig voor een optelloop, ook 16 maal zolang worden. Waar besparing van geheugenruimte belangrijker is dan snelheid is het zeker nuttig om de optelloop te maken. Wel moet dan de teller die het aantal bits bijhoudt (adres $020E en $0240) met indexregister Y gerealiseerd worden. De NOP-instructies zijn hier opgenomen als service voor hen die een 'ROR-loze' microprocessor hebben. De NOP-instructies veroorzaken geen acties van de microprocessor, maar zijn wel tijdverslinders. Wanneer deze vermenigvuldigingsroutine serieus gebruikt gaat worden verdient het aanbeveling de NOPjes uit dit programma te schrappen. Wel moet dan de sprongrootte op adres $0241 aangepast worden.

Vermenigvuldig getal 1 met 2

Zoals al eerder opgemerkt komt deze vermenigvuldiging neer op het één plaats naar links schuiven van GETAL 1. Het naar links schuiven gebeurt op dezelfde wijze als het naar rechts schuiven, zij het dat we nu aan de andere kant beginnen:

```
ASL, Zpage GETAL 1,1
ROL, Zpage GETAL 1,2
ROL, Zpage GETAL 1,3
ROL, Zpage GETAL 1,4
```

We zien weer 2 nieuwe instructies nl.:

ASL (Arithmetic Shift Left)
ROL (Rotate Left)

De werking is hetzelfde als van LSR en ROR, alleen de richting is anders. Het te verschuiven getal is hier 4 bytes lang, zodat er twee ROL-instructies extra nodig zijn.

Klaar?

In dit blokje wordt indexregister X met één verminderd. Zolang het resultaat géén '0' is, gaan we terug naar 'LOOP2'. Zodra het resultaat wél '0' is, gaan we terug naar het monitorprogramma. De gebruikte instructies spreken voor zich.