PHILIPS

MICROPROCESSOR ZELFSTUDIE-CURSUS

INLEIDING TOT MICROCOMPUTERS

1

INLEIDING TOT MICROCOMPUTERS

INHOUD

Microprocessoren hebben in korte tijd een groot toepassingsgebied gevonden. De reden hiervan is dat wij met behulp van dit element veel, soms uiterst gecompliceerde, problemen op vrij eenvoudige en goedkope wijze kunnen oplossen. Het toepassingsgebied van microprocessoren zal in de komende tijd dan ook sterk worden uitgebreid.

Om enig inzicht in de werking van microprocessoren te verkrijgen, is het noodzakelijk, problemen zorgvuldig te leren analyseren. Voorts dient men getallen anders voor te gaan stellen dan met de ons bekende decimale symbolen 0 t/m 9.

In veel problemen komen logische beslissingen voor en men moet zich vertrouwd maken met een methodiek om deze logische beslissingen te beschrijven. Voorts zijn er een groot aantal elektronische elementen, die ook in microprocessoren gebruikt worden, waarmee men logische beslissingen kan realiseren.

In de volgende hoofdstukken wordt een overzicht gegeven van het rekenen met behulp van logische elementen en worden de getalvoorstellingen behandeld die men nodig heeft om met vrucht van microprocessoren gebruik te kunnen maken.

HET REKENEN MET MICROPROCESSOREN

In eerste benadering kan het rekenen met microprocessoren het best verklaard worden met behulp van de thans alom bekende zakrekenmachines. Fig. 1 stelt een dergelijke zakrekenmachine voor. Bovenaan deze machine treffen we een "scherm" aan, waarin hier met behulp van symbolen het getal 532 is weergegeven. Dit scherm zullen we voortaan aanduiden d.m.v. de afkorting DP (afgeleid van het Engelse woord DISPLAY). Voorts treffen we een veld aan met toetsen. Tien van deze toetsen dragen de numerieke symbolen 0 t/m 9. Eén toets is aanwezig met de punt. Deze punt is in het algemeen in gebruik en komt overeen met onze komma. Verder zijn er een vijftal functietoetsen, namelijk x, :, +, – en =. De toetsen C en CE dienen voor het uitvoeren van correcties indien men abusievelijk een foute toets heeft aangeslagen.

Met behulp van de numerieke toetsen kan men waarden aan de processor aanbieden, en ze doen dan ook dienst als invoermedium, de z.g. INPUT. Een getal wordt aan een dergelijke processor verstrekt door de cijfers van dit getal achtereenvolgens aan te slaan, te beginnen met het meest significante cijfer; dit betekent dat voor het getal 532 eerst de 5, dan de 3 en vervolgens de 2 moet worden ingedrukt. Maakt men een fout bij een dergelijke invoer, dan kan men deze herstellen door op de knop CE te drukken, waarna men het getal opnieuw dient aan te slaan. Men kan nu de vraag stellen, hoe een processor weet wanneer men het laatste cijfer heeft ingevoerd. Nu is het bij de meeste processoren gebruikelijk dat men, na het laatste cijfer te hebben aangeslagen, een operatietoets indrukt, d.w.z. de x, :, + of –. Behalve de bewerking die men tussen twee getallen wenst te verrichten, geeft men hiermee dus het einde van de cijferreeks aan. Vervolgens voert men het volgende getal m.b.v. het numerieke toetsenveld in.

Het einde van het laatste getal kan men niet meer aangeven d.m.v. een operatietoets. In de plaats daarvan voert men een pseudo-operatie uit, door het = teken in te drukken. Deze toets geeft aan dat de cijferreeks van het laatste getal is beëindigd, en dat men verlangt dat de eerder aangegeven operatie wordt uitgevoerd.

De ingevoerde getallen worden veelal aangeduid als de z.g. operanden. Laten we nu eens een berekening uitvoeren. We nemen hiervoor de formule 7 x 5 + 4 x 3.

Stel dat we niets weten van de regels over de volgorde van vermenigvuldigen, optellen, enz. We kunnen nu de volgende handelingen verrichten:

toets 7 indrukken
toets x indrukken
toets 5 indrukken
toets = indrukken
toets + indrukken
toets 4 indrukken
toets = indrukken
toets x indrukken
toets 3 indrukken
toets = indrukken

Het resultaat zal nu zijn dat op de DP het getal 117 zichtbaar wordt. Volgens de methode waarop wij geleerd hebben te vermenigvuldigen en op te tellen, is dit antwoord onjuist.

Kennelijk kan de rekenmachine geen verschil maken tussen de volgorde van het optellen en het vermenigvuldigen. Deze volgorde zullen wij zelf op een of andere wijze in de rekenmachine dienen te brengen. Laten we opnieuw de berekening uitvoeren:

toets 7 indrukken
toets x indrukken
toets 5 indrukken
toets = indrukken

Nu verschijnt het getal 35 op de DP. De geschetste rekenmachine is zeer eenvoudig en we hebben geen mogelijkheid om dit getal op één of andere wijze te onthouden. Om deze moeilijkheid te ontgaan, nemen we het getal over op een stukje papier en schrijven daar dus 35 op. Om nu de volgende berekening uit te voeren, moeten we eerst zorgen dat de rekenmachine weer "schoon" is door op C te drukken. Nu verrichten we de volgende berekening:

toets 4 indrukken
toets x indrukken
toets 3 indrukken
toets = indrukken

Nu verschijnt het getal 12 op de DP. Dit getal kunnen wij in de rekenmachine laten staan. We slaan vervolgens de + aan en daarna 35. Om het resultaat te krijgen, drukken we de = toets in. Nu verschijnt het getal 47 op de DP. Wat we hebben gedaan, is eerst een analyse maken van de berekening, namelijk eerst vermenigvuldigen, het resultaat bewaren, vervolgens weer vermenigvuldigen en het oude resultaat erbij optellen. We kunnen dit in een z.g. stroomdiagram weergeven, zie fig. 2. Een dergelijk stroomdiagram geeft alleen in grote trekken weer wat er dient te gebeuren. In het aangegeven stroomdiagram wordt gezegd: vorm 7 x 5, schrijf het resultaat op, d.w.z. bewaar het "ergens", maak de rekenmachine schoon, vermenigvuldig 4 met 3 en tel er het bewaarde resultaat bij op.

Bij de voorgaande berekeningen kunnen we de volgende serie handelingen waarnemen:

a. Het inbrengen van een getal, het z.g. laden.
b. Het inbrengen van de operatie die verricht moet worden.
c. Het laten uitvoeren van de berekening.
d. Het overnemen op een "kladblok" van het tussenresultaat van de vermenigvuldiging.
e. Het "schoonmaken" van het rekenorgaan in de rekenmachine.
f. Het verkrijgen van het resultaat op de DP, de z.g. uitvoer (de OUTPUT).

Een tweetal bijzondere functies springen hierbij in het oog, namelijk de functies genoemd onder d en e, het tijdelijk opslaan van het tussenresultaat, d.w.z. het "onhouden" en het "schoonmaken" van de rekenmachine. Dit "onthouden" moet bij deze eenvoudige rekenmachine op een "kladblok" gebeuren. Een microprocessor met zijn omringende apparatuur biedt de faciliteit om dit tussenresultaat in een elektronisch geheugen te onthouden, of – zoals men dit noemt – te registreren. Het "schoonmaken" dient te geschieden omdat deze eenvoudige zakrekenmachine slechts over één element beschikt, dat iets kan onthouden. Moet dit element voor andere doeleinden gebruikt worden, dan dienen we er eerst voor te zorgen dat de oude informatie "gered" is.

Uit het bovenstaande blijkt duidelijk dat deze kleine zakrekenmachines erop gebaseerd zijn, dat degene die ze bedient, de nodige kennis heeft om de berekening op de juiste wijze uit te voeren. Het rekenen zelf, d.w.z. het optellen of het vermenigvuldigen, doet de eenvoudige rekenmachine voor ons. Dit rekenen is voor de meeste mensen de meest tijdrovende bezigheid en men maakt daarbij dan ook veel fouten. Hoe zal nu een dergelijke rekenmachine dienen te werken? Beschouwen we de bovenstaande berekening nader, dan blijken alle handelingen te geschieden tussen een tweetal getallen, de operanden. We zullen de ene operand A, en de andere operand B noemen. Eerst wordt m.b.v. het numerieke toetsenveld de operand A in de rekenmachine ingevoerd. Deze operand dient dus "onthouden" te worden door de machine. Daartoe beschikt de machine over een zeer klein "geheugen", dat we een register noemen. Vervolgens delen we de machine mee dat we wensen te vermenigvuldigen. Ook dit dient de machine te onthouden in een daartoe bestemd register. Tenslotte verstrekken we aan de machine de tweede operand B. Ook deze operand wordt bewaard in een register. We zullen de beide registers voor de operanden REG A en REG B noemen. Door de = toets in te drukken, wordt de berekening uitgevoerd. Dit gebeurt in een elektronische schakeling, die de ALU wordt genoemd; dit is een afkorting van Arithmetic and Logical Unit, het z.g. rekenkundig logisch element, dat dikwijls ook wel het "Rekenorgaan" wordt genoemd. Dit is in fig. 3 voorgesteld. Men ziet het register A, het register B, het register dat in de operatiecode gebruikt wordt, en tenslotte het scherm DP, waarop het resultaat van de bewerking zichtbaar wordt gemaakt.

Als een microprocessor wordt gebruikt, dan dient men vooraf de nodige getallen en operatiecodes in een z.g. geheugen te registreren. Ook tussenresultaten, en hetgeen uiteindelijk zichtbaar gemaakt moet worden, worden in dit geheugen geregistreerd, en wel op plaatsen die wij zelf kunnen bepalen.

Welke handelingen zijn er nu allemaal nodig om de informatie die op een of andere wijze in het geheugen geregistreerd is, naar de ALU en de bijbehorende registers te brengen? In de meest eenvoudige vorm kan dit als volgt gebeuren:

Breng de eerste operand naar register A
Breng de tweede operand naar register B
Presenteer beide registerinhouden aan de ALU
Voer de vereiste handeling of "operatie" uit
Berg het resultaat op in het geheugen.

Deze handelingen – de zogenaamde instructies – zijn nodig om de twee getallen 7 en 5 met elkaar te vermenigvuldigen,

en het resultaat in het geheugen op te bergen. Deze uitvoerige wijze van beschrijven is echter niet bruikbaar voor een microcomputer, niet alleen vanwege de microcomputer zelf, maar ook voor degene die deze instructies schrijft, de programmeur. Meestal wordt een korte schrijfwijze gebruikt met speciale afkortingen, die men mnemonics noemt. Op deze wijze zouden de bovenstaande vier instructies als volgt geschreven kunnen worden:

laad operand A
laad operand B
vermenigvuldig
berg resultaat op.

Deze kleine verzameling van instructies noemt men een programma. Een uitvoerige wijze van beschrijven, waarin meer over de fundamentele achtergronden van dit proces wordt uiteengezet, treft men aan in de handleiding over het programmeren van de microprocessor 2650. In deze handleiding is uitgegaan van de basishandelingen die nodig zijn voor het verrichten van operaties. De operanden die bij deze operaties betrokken zijn, worden tevens aangegeven. In dit hoofdstuk wordt slechts een inleiding gegeven over de basisbegrippen, zonder dat het programmeren hierbij in extenso verklaard wordt.

In het bovenstaande is in een korte omschrijving weergegeven wat er in een rekenmachine dient te geschieden. We zullen dit nader gaan beschouwen. In het bovenstaande korte programma kan men een tweetal groepen van instructies onderscheiden, namelijk:

a. De transportinstructies: LAAD en BERG OP
b. De instructie VERMENIGVULDIG

De eerste groep zijn transportinstructies. De tweede groep zijn instructies die betrekking hebben op de bewerkingen; deze moeten geschieden tussen de beide operanden in register A en register B.

Er is nog een derde groep instructies die in de rekenmachine van uitermate groot belang zijn. Deze groep instructies hebben betrekking op beslissingen. De aard van deze beslissingen dienen wij zelf te bepalen. In een volgend hoofdstuk wordt hierop teruggekomen.

TALSTELSELS

Om gegevens te kunnen uitwisselen tussen de mens en de rekenmachine en ook tussen rekenmachines onderling, zijn symbolen nodig, waarvan de betekenis vooraf bepaald is. Is de betekenis van deze symbolen niet ondubbelzinnig afgesproken, dan ontstaat er verwarring. De afspraak over de betekenis wordt bij rekenmachines in de meeste gevallen ingebouwd, zodat het noodzakelijk zal zijn dat de mens deze betekenis leert. De afspraken die gemaakt worden over de betekenis van de elementen, noemt men een *code*.

De mens is al lang gewend aan het gebruik van codes. Zo vormen de verkeerslichten, de verkeersborden, pijlen op de weg, enz. codes, die gebruikt worden in het verkeer (zie fig. 4). De wetgever gebruikt deze codes om de weggebruiker te wijzen aan welke voorschriften hij dient te voldoen. Een andere bekende code wordt gevormd door de morsetekens, die worden gebruikt voor het overseinen van letters en cijfers.

Welke code voor een bepaalde toepassing wordt gekozen, hangt af van verschillende factoren, b.v. de geschiktheid voor het rekenkundig bewerken van deze symbolen, de mogelijkheid om fouten in codes te ontdekken en/of te corrigeren, het gemak waarmee men bepaalde codes vast kan leggen, zoals b.v. bij ponskaarten enz.

Moderne rekenmachines gebruiken als basiselement in alle code-afspraken slechts twee symbolen, n.l. de 0 of de 1. De reden van deze eenvoudige afspraak is dat met de huidige elektronische middelen deze 0 en 1 zo eenvoudig te realiseren zijn.

Data-representatie

Bekende voorbeelden van data-representatie zijn codes die in geheimschriften gebruikt worden. Daar maakt men de afspraken zodanig dat een buitenstaander deze afspraken niet eenvoudig kan reconstrueren. De afspraken kunnen betrekking hebben op vele duizenden tekens. Een bepaalde letter van een alfabet kan dan, afhankelijk van de plaats waar hij voorkomt, een ander symbool vertegenwoordigen dan dezelfde letter op een andere plaats. Ten behoeve van rekenmachines wordt dit niet gedaan en worden eenvoudige afspraken gemaakt. Deze afspraak, de "code", transformeert de afbeelding van een "verzameling van tekens" A op een andere tekenverzameling B volgens fig. 5. Zo is de tekenverzameling A voor de mensen eenvoudig herkenbaar en kan bestaan uit de hoofd- en kleine letters van het alfabet, leestekens, enz. De andere afbeelding, n.l. B, heeft betrek-

king op symbolen die b.v. door een rekenmachine of door een communicatiesysteem eenvoudig geïnterpreteerd kunnen worden. Het voor ons eenvoudige teken A is voor de rekenmachine een bijzonder gecompliceerd teken en kan heel moeilijk herkend worden. Men werkt wel aan herkenningsinstallaties voor dit soort schrift, maar de ontwikkeling ervan is nog niet heel ver gevorderd. Bovendien is de apparatuur die ervoor nodig is, vrij omvangrijk en tot op heden nogal kostbaar. Het is voor ons echter mogelijk om deze A in een vorm te brengen waarmee de rekenmachine weinig moeite heeft. Een voorbeeld daarvan is de zogenaamde EBCDIC-code. In deze code wordt de hoofdletter A voorgesteld door de reeks 1 1 0 0 0 0 0 1. De volgorde waarin deze nullen en enen staan, is van groot belang en bepaalt het gekozen symbool, in ons geval dus de A. De verzameling B, waarmee de rekenmachine werkt, bestaat uit groepen van nullen en enen, die als zodanig door de afspraak een bepaalde betekenis hebben verkregen. De nullen en enen noemt men *bits* (binary information ticket).

De alfanumerieke gegevens, d.w.z. de hoofdletters + cijfers, kunnen door 36 verschillende combinaties van "0" en "1" worden voorgesteld. Daarvoor zijn dan groepen van ten minste 6 nullen en/of enen voor vereist.

Zeer veel gebruikte codes zijn:

American Standard Code for Information Interchange (ASCII)
Extended Binary Coded Decimal Interchance Code (EBCDIC)

De afspraken betreffende deze beide codes zullen later ter sprake komen.

Gewogen codes

De codes zoals die tot dusver behandeld werden, zijn eenvoudige afspraken. In het algemeen zijn ze ongeschikt voor de directe numerieke verwerking. Voor een aantal codes, speciaal die waarbij gebruik gemaakt wordt van letters, is het verband tussen de numerieke bewerking en de codeafspraken van geen belang. Anders ligt het als de code gebruikt moet worden voor het bepalen van bepaalde resultaten. In dat geval gaat men liever gebruik maken van wat men *gewogen codes* noemt.

Een bekend voorbeeld van gewogen codes is het decimale talstelsel. Het decimale talstelsel maakt gebruik van tien symbolen, n.l. 0 t/m 9. We zien hier direct dat er even veel symbolen zijn als de naam van het talstelsel aangeeft, n.l. 10. Elk symbool heeft zijn eigen gewicht, en in de reeks 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 verschillen de gewichten van de opeenvolgende tekens telkens met een waarde 1. Om getallen te kunnen maken die uitgaan boven het gewicht van deze 10 symbolen, heeft men een bepaalde systematiek verzonnen, waarbij de volgorde van de symbolen eveneens een bepaalde betekenis heeft. Zo heeft in het getal 252 de eerste 2 een andere betekenis dan de laatste 2. De eerste 2 geeft aan het gewicht in "honderdtallen" en de laatste 2 geeft het gewicht in "eenheden". Op deze wijze kan men het getal 252 als volgt schrijven:

$$200 + 50 + 2$$
$$\text{of} \quad 2.100 + 5.10 + 2.1$$
$$\text{of} \quad 2.10^2 + 5.10^1 + 2.10^0$$

Deze laatste voorstelling heeft de vorm van een reeks, en wel een reeks waarin de machten van 10 zijn gegeven, aangevende de gewichten die de symbolen op een bepaalde plaats hebben. Bij het schrijven van decimale getallen laten we deze specifieke gewichten eenvoudig weg, en moet de mens deze op grond van zijn leerproces in zijn kinderjaren zelf invullen. De mens is er verder aan gewend om ook de nullen voor en achter een bepaald getal eenvoudig weg te laten. In rekenmachines zal dit niet altijd mogelijk zijn, daar ze het ontbreken van een bepaald element eenvoudig niet accepteren.

Bekijken we nu getallen die een "komma" bevatten, b.v. 641,8, dan kan men dit getal schrijven als:

$$6.10^2 + 4.10^1 + 1.10^0 + 8.10^{-1}.$$

In het algemeen kan elk reëel getal voorgesteld worden door een reeks in de vorm:

$$N = a_n.g^n + a_{n-1}g^{n-1} + \ldots + a_1.g^1 + a_0.g^0 + a_{-1}.g^{-1}$$
$$+ a_{-2}.g^{-2} + \ldots + a_{-m}.g^{-m}$$
$$= \sum_{i=-m}^{i=n} a_i.g^i$$

In deze reeksontwikkeling komen verschillende symbolen voor, n.l. het symbool g en de symbolen a. Het symbool g wordt het *grondtal* van het talstelsel genoemd. In het decimale systeem is dat het getal 10. Deze g geeft eveneens aan hoeveel verschillende symbolen er in het talstelsel gebruikt mogen worden. De symbolen a worden in ons decimale systeem de *cijfers* genoemd en variëren van 0 t/m 9, uiteraard bij g = 10. Opmerkelijk is de overgang tussen eenheden en fracties bij $a_0.g^0$ en $a_{-1}.g^{-1}$.

In ons decimale systeem plaatsen wij, omdat we de gewichten van de plaatsen niet aangeven, tussen deze eenheden en fracties een komma. In de angelsaksische landen wordt hiervoor nog een punt gebruikt. Ontbreken de termen a_{-1}, a_{-2}, enz., m.a.w. als voor de ondergrens van de som geldt i = 0,, dan hebben we te maken met gehele getallen (*integers*). Is de bovengrens n = −1 en de ondergrens ≠ 0 dan hebben we te maken met echte breuken (*fractions*).

Voor gewogen codes gelden enkele bijzondere eigenschappen:

1. Voor elke a_i in een reeks geldt dat deze $a_i \leqslant g-1$.
   Als voor elke a_i in de reeksontwikkeling geldt:
   $a_i = g-1$, dan gelden de onderstaande eigenschappen:

$$\sum_{i=0}^{i=n-1} a_i.g^i = g^n - 1 \quad \text{(b.v. } 999 = 1000-1,\ n=3\text{)}$$

$$\sum_{i=-k}^{i=n-1} a_i.g^i = g^n - g^{-k} \quad \text{(als k nadert tot } +\infty\text{, wordt dit } g^n\text{)}$$

Voorbeeld: 999,99 = 1000 − 0,01
999,999 . . . . = 1000

2. Aangezien $a_i \leqslant g-1$, is $2.a_i \leqslant 2.(g-1)$
   of $a_i.g^i\ a_i.g^i \leqslant 2.(g-1).g^i$, waaruit volgt:

$$a_i.g^i + a_i.g^i + 1.g^i \leqslant 2.(g-1).g^i + g^i$$
$$= (2g-1)g^i < 2.g^{i+1}$$

Hieruit volgt dat men bij het optellen van twee getallen nimmer meer dan een 1 hoeft te onthouden. Deze wordt voortgebracht uit de optelling van de voorgaande cijfercombinaties. (Men wordt erop gewezen dat dit uitsluitend geldt als men 2 getallen bijeentelt.)

$$g.\sum_{i=0}^{i=n-1} a_i.g^i = \sum_{i=0}^{i=n-1} a_i.g^{i+1}$$

$$= \sum_{i=1}^{i=n} a_{i-1}.g^i + 0.g^0$$

Dit houdt in dat het vermenigvuldigen van een getal met het grondtal neerkomt op het verschuiven van het getal over één plaats naar links en het plaatsen van een 0 op een eenhedenplaats.

Dit is wel bekend in het decimale talstelsel en geldt uiteraard alleen voor getallen zonder komma's. Bij aanwezigheid van een komma, moet deze komma een plaats naar rechts verschoven worden. Ga dit zelf na aan de hand van de hierboven vermelde reeksontwikkeling.

Voorbeelden van talstelsels

1. *Tweetallig stelsel:* het grondtal is 2
   de symbolen zijn 0 en 1.

$0 = 0.2^0 = 0$
$1 = 1.2^0 = 1$
$2 = 1.2^1 + 0.2^0 = 10$
$3 = 1.2^1 + 1.2^0 = 11$
$4 = 1.2^2 + 0.2^1 + 0.2^0 = 100$
. . . . .
$9 = 1.2^3 + 0.2^2 + 0.2^1 + 1.2^0 = 1001$
. . . . . .
$15 = 1.2^3 + 1.2^2 + 1.2^1 + 1.2^0 = 1111$

Deze zogenaamde *binaire* code is van uitermate groot belang en wordt veelvuldig in digitale rekenmachines toegepast.

Het is dus geboden dat men een getal zonder meer in deze binaire code kan voorstellen.

2. *Hexaldecimale talstelsel:* het grondtal is 16
   de symbolen zijn 0 t/m 9, A, B, C, D, E, F.

De symbolen A t/m F zijn gekozen i.p.v. de symbolen 10 t/m 15. Het voordeel van de eerstgenoemde symbolen is dat men ze niet kan verwisselen met de symbolen 10 t/m 15. Dit talstelsel wordt veel gebruikt voor het op eenvoudige wijze voorstellen van groepen van 4 bits van de binaire code. Deze groepen van 4 bits hebben in totaal 16 mogelijkheden en het invoeren daarvan in de rekenmachines leidt vaak tot fouten. Hetzelfde geldt voor het zichtbaar maken van gegevens van een rekenmachine aan de omgeving. Om dit proces van communicatie te vereenvoudigen, wordt vaak van de hexadecimale code gebruik gemaakt. Op zichzelf behoudt deze hexadecimale code uiteraard de volledige rekenkundige eigenschappen, zoals deze bij de talstelsels in het algemeen behandeld zijn.

Tellen in het binaire talstelsel

Wenst men in een talstelsel twee getallen te sommeren, dan doet men dit in wezen door de getallen onder elkaar te schrijven, en wel zodanig dat de symbolen met gelijke gewichten, d.w.z. dezelfde macht van g, onder elkaar komen te staan. Men telt nu de symbolen samen en hieruit wordt een nieuw symbool gevonden, b.v. in het decimale talstelsel $7 + 2 = 9$. Dit is een z.g. rekenregel.

Voor het binaire talstelsel gelden de volgende rekenregels:

$0.2^i + 0.2^i = 0.2^i \quad (0 + 0 = 0)$
$0.2^i + 1.2^i = 1.2^i \quad (0 + 1 = 1)$
$1.2^i + 0.2^i = 1.2^i \quad (1 + 0 = 1)$
$1.2^i + 1.2^i = 2.2^i = 1.2^{i+1} + 0.2^i$ (1 + 1 = 0 en 1 "onthouden").

Voor de binaire aftrekking geldt:

$0 - 0 = 0$
$0 - 1 = 1$ (1 "geleend")
$1 - 0 = 1$
$1 - 1 = 0$

Deze laatste rekenregels kan men op een eenvoudige wijze bepalen analoog aan die van de binaire optelling.
Voor de binaire vermenigvuldiging geldt:

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

Men dient zorgvuldig in het oog te houden welke macht van g ontstaat. Hierop wordt bij het vermenigvuldigen nog nader teruggekomen.

Voorbeeld van een optelling:

```
  1011
  1001 +
  ----
 10100
```

Woordlengte

Wanneer wij getallen opschrijven, zijn we meestal in staat om te werken met getallen van zeer wisselende lengten. Rekenmachines daarentegen werken meestal met getallen van een vast aantal symbolen. Men noemt dit de woordlengte. De woordlengten die worden toegepast, lopen nogal uiteen. Veel gebruikte lengten zijn: 8, 12, 16, 18, 24, 32, 48 en 64 bits. Veel moderne microcomputers maken gebruik van woorden met een lengte van 8 bits. Voor speciale toepassingen zijn er ook wel microprocessoren met een woordlengte van 4 bits. Deze worden speciaal gebruikt voor het rekenen met decimale getallen, zoals men b.v. kan constateren bij de bekende moderne zakrekenmachientjes.

Heeft een machine maar een beperkte woordlengte, dan zal het toch vaak mogelijk zijn ook met grotere woordlengten te werken. Men deelt dan het grotere woord in een aantal woorden op ter grootte van het standaardwoord van de rekenmachine. Met behulp van programmatuur kan men nu deze deelgetallen bij elkaar optellen, resp. vermenigvuldigen. In machines met een woordlengte van 8 bits (die men vaak een byte noemt), heet dit een "multiple byte operation". Er zijn ook rekenmachines waarbij de lengte van een woord variabel in grootte kan zijn. Dit is dan wel afgerond op b.v. eenheden van 8 bits. Om aan te geven dat het getal beëindigd is, gebruikt men aan het einde van deze reeks een speciaal teken. Ook komt het voor dat men in de programmatuur aangeeft hoe lang het getal behoort te zijn. Het zou buiten het bestek van deze inleiding vallen, hierop nader in te gaan.

Conversie van talstelsels

Er zijn twee algemene methodes om een bepaald talstelsel in een ander om te zetten. De eerste methode is die van het aftrekken van machten.

Men bepaalt in het oorspronkelijke talstelsel de grondtallen van het andere talstelsel, en kijkt hoe vaak men dit van het getal kan aftrekken. Men neemt daartoe een bepaalde macht van dit talstelsel en kijkt na het aftrekken van het oorspronkelijke getal of de rest kleiner is dan het getal dat men afgetrokken heeft. Is dat niet het geval, dan past men een correctie toe door een hogere macht te nemen en het dan opnieuw te proberen. Dit is een z.g. "try and error" methode, die bij rekenmachines ongebruikelijk is. Ze komt neer op een normale deling, zoals we deze verrichten in het decimale talstelsel; men weet uit ervaring dat men niet altijd de "meest gelukkige greep" doet bij het schatten van het juiste getal.

Een andere methode is die waarbij we heel systematisch te werk moeten gaan. Stel het getal is:

$$N = a_i.g^{m-1} \ldots a_1.g^1\ a_0.g^0.$$

Nu is een bekende stelling uit de rekenkunde dat bij deling van een getal door een ander geheel getal, er een quotiënt ontstaat benevens een rest, die niet meer door het desbetreffende getal deelbaar is, d.w.z. dat deze rest kleiner is dan de deler. Stel men deelt een getal N door b, dan is het resultaat:

$$N = Q.b + r \quad (r < b)$$

Het quotiënt kan men opnieuw delen door hetzelfde getal b, waardoor een nieuw quotiënt ontstaat en een nieuwe rest. Zo kan men doorgaan met telkens het nieuwe quotiënt te delen. Dit kan men herhalen totdat het nieuwe quotiënt niet meer deelbaar is en men dus uitsluitend overhoudt: $Q = 0.b + r$.

Dit proces is hieronder weergegeven:

$$
\begin{array}{ll}
N = Q_0 .b + r_0 & \\
Q_0 = Q_1 .b + r_1 & N = Q_1 .b^2 + r_1 .b + r_0 \\
Q_1 = Q_2 .b + r_2 & N = Q_2 .b^3 + r_2 .b^2 + r_1 .b + r_0 \\
\ldots\ldots\ldots & \ldots\ldots\ldots\ldots\ldots \\
Q_{n-1} = 0.b + r_n & N = r_n .b^n + r_{n-1} .b^{n-1} + \ldots + r_2 .b^2 + r_1 .b + r_0
\end{array}
$$

Dit is opnieuw een reeks, waarin nu echter het getal b het grondtal is. De symbolen r kunnen nimmer $\geqslant$ b zijn, immers dan is de rest bij de deling onjuist geweest. Deze methode geeft dus aan hoe men een bepaald talstelsel in een ander kan converteren.

Voorbeeld: neem het getal 47 in het decimale talstelsel; wat is dan de waarde in het binaire talstelsel?

	Quotiënt	Rest
2 /47\ 1	23	1
2 /23\ 1	11	1
2 /11\ 1	5	1
2 /5 \ 1	2	1
2 /2 \ 0	1	0
2 /1 \ 1	0	1

De "1" geheel onderaan de kolom van de resten is de meest significante rest en het binaire getal luidt dus: 101111. Men kan de waarde van dit binaire getal eenvoudig bepalen door deze reeks "nullen" en "enen" in de machtreeks met het grondtal 2. Het wordt aan de lezer zelf overgelaten dit na te gaan.

Negatieve getallen

In de rekenkunde komen veel getallen voor die negatief zijn. Het middel om dit aan te geven, is eenvoudig het plaatsen van een minteken voor het desbetreffende getal. De definitie van een negatief getal is:

$$a + (-a) = 0$$

Men geeft negatieve getallen vaak aan op een z.g. getallenrechte, zie fig. 6. Hierop ziet men de getallen –4 via 0 t/m g^n.

Positieve getallen worden wel met behulp van een + teken aangegeven, hoewel dit ongebruikelijk is. Gebruikt men digitale rekenmachines, dan is er voor een plus- of een minteken meestal een speciale afspraak aanwezig. Dit is echter bij moderne machines uitsluitend het geval als deze getallen deel uitmaken van een reeks tekens, waaronder de alfabetische. Zodra echter deze getallen gebruikt moeten gaan worden voor rekendoeleinden, dan worden ze meestal geconverteerd in een andere vorm. Werken rekenmachines echter met zuiver decimale getallen – en ook dit komt vaak voor – dan wordt het teken gehandhaafd. Het heet dan dat de machine werkt in een "sign and magnitude"-vorm.

Als een machine echter niet meer in deze vorm werkt, maar met binaire getallen, dan zal men een andere voorstellingsmethode prefereren. Veel gebruikte voorstellingsmethoden zijn die van het "1" of het "2"-complement.

Ook andere methoden treft men aan. We zullen ons hier beperken tot die talstelsels die het getal 2 als grondtal bezitten.

Stel dat een binair getal n posities heeft. In de absolute getallenreeks zijn hier nu de volgende mogelijkheden 0 t/m $2^n - 1$ (bepaal dit zelf aan de hand van de al eerder gegeven machtreeks-voorstelling). Willen we nu negatieve getallen invoeren, dan zullen we een deel van dit gebied van 2^n getallen dienen op te offeren voor de voorstelling van negatieve getallen. Meestal deelt men dat gebied dan in twee delen, zodat de ene helft van de getallen negatief is en de andere helft van de getallen positief, waarbij het getal 0 dan doorgaans tot de positieve getallen wordt gerekend. Men dient nu wel te bedenken dat de afspraken die gemaakt zijn voor de numerieke berekening van getallen, niet zonder meer van toepassing zijn op deze voorstellingswijze. De positieve getallen zullen hierop een uitzondering vormen. Voor negatieve getallen zullen we hier nu nader ingaan. Ter toelichting kan nog gesteld worden dat de positieve getallen in de hierna te bespreken methode altijd een 0 hebben op de meest significante plaats, de negatieve getallen daarentegen een 1. Het hoe en waarom, zal nu nader worden uiteengezet. Voor de positieve getallen geldt dezelfde rekenwijze als hiervoor is behandeld.

"1"-complement binaire getallen

In het 1-complement voorstellingssysteem van binaire getallen geldt per definitie dat een negatief getal voorgesteld zal worden door een reeks van nullen en enen, die verkregen zijn door de onderstaande formule consequent toe te passen:

$$a + (-a) = 2^n - 1$$

De n stelt het aantal posities van het getal voor. Bij 8 posities (1 byte) is n = 8 en dus 2^n = 256. Men kan aan de nullen en enen in dit geval geen directe binaire gewichten toekennen, maar er bestaat volgens bovenstaande formule

uiteraard een duidelijke relatie met de binaire gewichten. Hieronder volgen enkele voorbeelden van negatieve getallen. (Voor de eenvoud zijn hier slechts 4 bits genomen, waardoor er niet meer dan 2^4 in 16 getallen voorgesteld kunnen worden.)

	negatief (2^n-1)
0 = 0000	$-0 = 1111 - 0000 = 1111$
1 = 0001	$-1 = 1111 - 0001 = 1110$
2 = 0010	$-2 = 1111 - 0010 = 1101$
3 = 0011	$-3 = 1111 - 0011 = 1100$
4 = 0100	$-4 = 1111 - 0100 = 1011$
5 = 0101	$-5 = 1111 - 0101 = 1010$
6 = 0110	$-6 = 1111 - 0110 = 1001$
7 = 0111	$-7 = 1111 - 0111 = 1000$

Het blijkt onmiddellijk dat voor de positieve getallen een 0 op de meest significante plaats staat en voor de negatieve getallen een 1. Uit het geheel is duidelijk te zien dat de waarde $2^n - 1$ bestaat uit een reeks van 4 enen. Dit bleek al uit de eerder behandelde eigenschappen van de sommen van reeksen bij gelijke termen.

Een nadeel van deze methode is dat er twee waarden van "nul" bestaan, n.l. een positieve 0 en een negatieve 0. Voorts geeft het optellen van deze getallen in een zuiver binair rekenorgaan enige moeilijkheden. De som van twee negatieve getallen zal namelijk luiden:

$$a + (2^n - 1 - b) = a - b + 2^n - 1 = (a - b - 1) + 2^n$$

aannemende dat $a - b \geqslant 1$. Nu valt de waarde 2^n echter buiten de mogelijkheden van de rekenmachine. De rekenmachine heeft immers slechts plaats voor n symbolen en 2^n heeft een waarde die buiten dit gebied valt. Nemen we als voorbeeld $n = 4$, dan is $2^n = 16$, en dat dient in binaire code geschreven te worden als: 10000. Deze voorstelling neemt echter 5 plaatsen in beslag, terwijl de machine er slechts 4 bezit. Deze vijfde plaats valt dus buiten het bereik van de machine en wordt niet weergegeven. $2^n - 1$ echter heeft de waarde 1111, en hierdoor wordt de som van het positieve en het negatieve getal beïnvloed. Is het resultaat een negatief getal, dan zal dit volkomen terecht gebeuren. Is het getal echter positief, dan is het resultaat foutief, zoals uit onderstaande voorbeelden blijkt:

+6 + (−5) = ?

```
+6 =  0110
−5 =  1010
      ----  +
1 |   0000 = +0
```

+6 + (−0) = ?

```
+6 =  0110
−0 =  1111
      ----  +
1 |   0101 = +5
```

Deze voorbeelden laten zien dat, indien het resultaat positief is, de uitkomst onjuist zal zijn. De 1 die op de verkeerde plaats terecht is gekomen, levert echter een indicatie. Deze 1 kwam voort uit de term $2^4 - 1$, die 1111 voorstelt. Wat men er ook bijtelt, heeft tot gevolg dat de 1 op de meest significante plaats ontstaat, en wel links van de streep. Men heeft er nu een 1 te weinig bijgeteld, namelijk de 1 die van 2^n wordt afgetrokken. Men kan dit nu corrigeren door deze 1 alsnog aan het resultaat toe te voegen, zoals hierna is aangegeven.

+6 + (−5) = ?

```
+6 =  0110
−5 =  1010
      ----  +
1 |   0000
  \-------> 1
      ----  +
      0001 = +1
```

+6 + (−0) = ?

```
+6 =  0110
−0 =  1111
      ----  +
1 |   0101
  \-------> 1
      ----  +
      0110 = +6
```

We zullen nu twee andere voorbeelden nemen, namelijk dat −6 bij +5 wordt opgeteld:

+5 + (−6) = ?

```
+5 =  0101
−6 =  1001 +
      ----
      1110 = −1
```

We zien dat de uitkomst nu correct is en dat deze correcte uitkomst wordt gekenmerkt door het feit dat er geen "1" links van de 4 bits is verschenen. Hieruit blijkt dat het resultaat negatief is, en er behoeft dus geen correctie te worden toegepast.

Worden twee negatieve getallen opgeteld, dan is uiteindelijk het resultaat wederom negatief, maar nu is de correctie twee keer opgetreden, zoals uit onderstaand voorbeeld blijkt:

−3 + (−4) = ?

```
−3 =  1100
−4 =  1011 +
      ----
1     0111 = +7
  \-------> 1
      ----  +
      1000 = −7
```

Er staat opnieuw een "1" op buiten de getalgrootte, hetgeen betekent dat men alsnog een "1" bij het resultaat dient op te tellen. We zien hieruit dat het resultaat +7 in de juiste waarde −7 verandert. Het aantrekkelijke van de 1-complement methode is de eenvoudige manier, waarop een negatief getal uit een positief getal kan worden verkregen.

Als men de tabel goed beschouwt, dan komt het erop neer dat men aftrekt van een reeks "enen", d.w.z. men behoeft nimmer te lenen; op de plaats waar in het oorspronkelijke getal een 1 staat, komt een 0 te staan in het resultaat. Op dezelfde wijze, waar een 0 staat, komt een 1 te staan, m.a.w. alle "nullen" worden "enen" en alle "enen" worden "nullen". Dit is elektronisch zeer snel en eenvoudig te realiseren. Heel veel machines werken dan ook met deze methode voor het verkrijgen van een negatief getal. De complicatie met het al dan niet optellen van een "1", kan men vermijden door in de "2"-complement methode te gaan werken, die we nu zullen behandelen.

"2"-complement binaire talstelsel

Bij het "2"-complement binaire talstelsel wordt een negatief getal per definitie vastgesteld door de formule:

$a + (-a) = 2^n$

waar n weer het aantal bits van het getal voorstelt. Hieronder volgt, weer voor n = 4, een aantal getallen:

	2 complement
0 = 0000	–0 = 10000 – 0000 = 0000 (idem)
1 = 0001	–1 = 10000 – 0001 = 1111
2 = 0010	–2 = 10000 – 0010 = 1110
3 = 0011	–3 = 10000 – 0011 = 1101
4 = 0100	–4 = 10000 – 0100 = 1100
5 = 0101	–5 = 10000 – 0101 = 1011
6 = 0110	–6 = 10000 – 0110 = 1010
7 = 0111	–7 = 10000 – 0111 = 1001
	–8 = = 1000 (≠ +8)

Uit het bovenstaande voorbeeld blijkt weer dat alle positieve getallen, inclusief 0 (= –0), beginnen met een 0 en de negatieve getallen met een 1. Verder is het opmerkelijk, dat het getal –8 wèl bestaat, maar het getal +8 niet. Dit is moeilijk te rijmen met de definitie van een negatief getal, maar op zichzelf is deze vorm van definiëren volkomen geoorloofd. We zullen nu het rekenen met deze getallen onder de loep nemen en eenvoudigheidshalve dezelfde voorbeelden kiezen als bij het "1"-complement systeem.

+6 + (–5) = ?

+6 = 0110
–5 = 1011
1 | 0001 = +1

+6+(–0) = ?

+6 = 0110
–0 = 0000
0110 = +6

We zien dat hier de resultaten geheel in overeenstemming zijn met het gegeven. Bij de linker optelling is er een 1 links van de streep, die dus niet in onze rekenmachine past. Op zichzelf is dit niet bezwaarlijk; die 1 is afkomstig van een negatief getal, en wel van 2^n. Men kan telkens 2^n toevoegen zonder dat de uitkomst in gevaar wordt gebracht. Bij het voorbeeld rechts is het geheel vereenvoudigd. Daar wordt 0 opgeteld, en als zodanig klopt de uitkomst eveneens. Hoe zit het nu als een negatief getal met een absoluut grotere waarde bij een positief getal wordt opgeteld? Hieronder volgen twee voorbeelden.

+5 +(–6) = ?

+5 = 0101
–6 = 1010 +
1111 = –1

+2 +(–4) = ?

+2 = 0010
–4 = 1100 +
1110 = –2

Ook hier zien we dat het resultaat weer klopt. Dit is weer toe te schrijven aan het feit dat 2^n buiten het optelmechanisme valt. Het 2-complement kan eenvoudig verkregen worden door het 1-complement te nemen en vervolgens bij het resultaat 1 op te tellen. Het 1-complement bestond immers uit 2^n-1 en het 2-complement uit 2^n. De methodiek om een 1-complement te krijgen was heel eenvoudig: het veranderen van nullen in enen en enen in nullen. Het verkrijgen van een negatief getal in 2-complementvorm is dus een stap extra, namelijk het optellen van een 1 bij deze uitkomst.

In digitale rekenmachines wordt deze methodiek veelvuldig toegepast, in het bijzonder als twee positieve getallen van elkaar moeten worden afgetrokken. Bij het aftrekken geeft men het getal dat men aftrekt, in de 1-complementvorm door aan het rekenorgaan; in het rekenorgaan zelf wordt een 1 bij het resultaat geteld als een vaste routine bij het aftrekken van twee getallen. Deze 1 zal, zoals we later bij het rekenkundige apparaat zullen zien, op de minst significante plaats van het optelmechanisme worden toegevoerd. Een voorbeeld hiervan is onderstaand uitgewerkt.

+6 –(+5) = ?

+6 = 0110
–5 = 1010 ← 1-complement
extra 1 1
1 | 0001 = 1

+5 –(+6) = ?

+5 = 0101
–6 = 1001 ← 1-complement
1110
extra 1 1
1111 = –1

Uit deze voorbeelden blijkt dat zowel van 5 als van 6 het 1-complement wordt genomen, en daaraan eenvoudig een 1 extra wordt toegevoegd bij de optelling. De resultaten zijn dan toch correct.

Een bijzonder voorbeeld is het optellen van twee negatieve getallen, en ook het aftrekken van een negatief en een positief getal. De resultaten van de berekeningen komen weer overeen met de gewenste uitkomsten. Zie onderstaande voorbeelden:

$-4 + (-2) = ?$

$-4 = 1100$
$-2 = 1110$ +
$1\,|\,1010 = -6$

$5 - (-2) = ?$

$5 = 0101$
$-(-2) = 0001$ ⟵ 1-compl.
extra 1 1 +
$0111 = +7$

Overflow

In het vorige gedeelte, waar het 2-complement getal behandeld is, waren n plaatsen beschikbaar om symbolen in een woord te plaatsen. Hierdoor konden in totaal 2^n verschillende getallen worden gedefinieerd. Deze getallen waren voor de helft positief, voor de andere helft negatief, hetgeen betekent dat er $2^n - 1$ positieve getallen aanwezig kunnen zijn, inclusief de 0. Dit heeft tot gevolg dat het grootste positieve getal $2^{n-1} - 1$ kan zijn. Dat was in het vorige hoofdstuk ook duidelijk te zien bij de 7, die wordt voorgesteld door 0111. Door de beperkte woordlengte zal het niet mogelijk zijn getallen van onbeperkte grootte weer te geven. Men kan zich nu afvragen, als twee positieve getallen worden opgeteld, wat zal dan de som zijn? Uiteraard zal deze som de waarde van 7 niet mogen overtreffen; 7 is immers het grootste positieve getal dat bij 4 bits voorgesteld kan worden. Stel als voorbeeld dat de getallen 5 en 6 bij elkaar worden opgeteld. Zie hieronder.

$+5 = 0101$
$+6 = 0110$ +
$1011 = -5$
1 x 1 onthouden

De som is nu -5. Dit is uiteraard foutief. Welke fout is hierbij gemaakt? Tijdens de optelling is op de derde plaats een overdracht van een 1 ontstaan, die daarna op de vierde plaats is gezet. Dit betekent een verlies van een 8, en bovendien een verkeerde interpretatie van de overige getallen. De totale binaire som is nog volkomen correct, namelijk 11, maar door de afspraak over de afbeelding van negatieve getallen, is deze representatie van 11 niet geldig. Het overlopen van de 1 tussen de beide plaatsen noemt men een *overflow*. Deze overflow kan men vaststellen doordat bij een optelling van 2 positieve of 2 negatieve getallen de meest significante bit verandert. In het bovenstaande voorbeeld waren de beide meest significante bits nullen, en in het resultaat verscheen een 1. Dat is een aanwijzing dat er een getal is ontstaan dat boven de rekencapaciteit van het rekenkundige orgaan uitgaat. Hetzelfde kan optreden bij het optellen van twee negatieve getallen, zoals in onderstaand voorbeeld:

$-5 = 1011$
$-6 = 1010$ +
$1\,|\,0101$
1 x 1 onthouden.

In het voorbeeld is eveneens de meest significante bit gewijzigd, terwijl bovendien een 1 buiten het rekenbereik van de machine is gekomen! Dit gezamenlijk wijst erop dat er sprake is van een overflow. In het algemeen geldt dat er een overflow bij de optelling, resp. aftrekking heeft plaats gehad indien er transport van een 1 plaats heeft van de op één na meest significante bit naar de meest significante bit, dan wel van de meest significante bit naar de plaats die niet meer aanwezig is:

x x x x ⟵ x x x x overflows
1 x 1 onthouden

Bij het optellen van negatieve getallen zal er altijd een 1 buiten het rekengebied getransporteerd worden. Blijven de negatieve getallen echter binnen het bereik van de machine, dan zal er ook van de op één na meest significante plaats naar de meest significante bit een overdracht van een 1 optreden, zoals uit onderstaand voorbeeld blijkt:

$-3 = 1101$
$-4 = 1100$ +
$1\,|\,1001 = -7$
2x 1 onthouden.

Het optreden van twee overdrachten of het optreden van geen enkele overdracht duidt er dus op dat er géén overflow plaats heeft. Om dit bij gemengde getallen te illustreren zijn hieronder nog twee voorbeelden gegeven:

$+4 = 0100$
$-3 = 1101$ +
$1\,|\,0001 = +1$
2x 1 onthouden

$+3 = 0011$
$-4 = 1100$ +
$1111 = -1$
géén onthouden

Vermenigvuldigen

Het vermenigvuldigen in processoren kan op verschillende manieren geschieden. Enkele ervan zijn:

a. Langs schakeltechnische weg over een aantal cijfers (bits).
b. Door gebruik te maken van tabellen in geheugens.
c. Door herhaald schuiven en optellen.

De schakeltechnische methode vindt vooral toepassing bij binaire getallen. Er waren omstreeks 1970 al geïntegreerde schakelingen die twee getallen van elk 16 bits konden vermenigvuldigen. Heeft een woord een nog groter aantal bits, dan moet bij deze methode het woord in groepen worden gesplitst en worden deze groepen afzonderlijk vermenigvuldigd, waarna d.m.v. optelling het resultaat wordt verkregen.

Wenst men decimale getallen te vermenigvuldigen, in het bijzonder als ze variabel van lengte zijn, dan wordt meestal van tabellen gebruikt gemaakt. Het zijn dan in wezen de welbekende vermenigvuldigtabellen zoals ze op de lagere school worden onderwezen. In digitale rekenmachines worden steeds twee getallen met elkaar vermenigvuldigd.

Als voorbeeld zullen we twee getallen nemen die uit decimale cijfers bestaan, en deze vermenigvuldiging uitvoeren zoals dit zou geschieden in een rekenmachine. Wil men twee getallen van n cijfers onderling vermenigvuldigen, dan ontstaan daarbij alle machten van het grondgetal, en wel de machten van 0 t/m 10^{2n-1}. Het vermenigvuldigen van de cijfers gebeurt ook hier in wezen met de vermenigvuldigtabel die we uit het hoofd geleerd hebben. Het uitlijnen van de vermenigvuldiging doen wij door telkens een 0 aan het einde van het resultaat te plaatsen. Dit komt overeen met het vermenigvuldigen met het grondtal in de tweede macht, enz.

Bij de methodiek die men op de scholen leert, worden eerst alle vermenigvuldigingen uitgevoerd, waarna men de optelling verricht. Dit leidt vaak tot fouten. Men kan eenvoudiger één vermenigvuldiging uitvoeren en vervolgens één optelling, opnieuw één vermenigvuldiging en dan één optelling, enz. Het onderstaande voorbeeld illustreert dit:

	A = 214	
	B = 8236	
som	00000000	
6x 214	1284	+
som	00001284	
30x 214	6420	+
som	00007704	
200x 214	42800	+
som	00050504	
8000x 214	1712000	+
som	01762504	

Het vermenigvuldigen in een rekenmachine kan men uitvoeren m.b.v. twee registers en een accumulator. In register 1 (zie fig. 7) wordt, zoals in ons voorbeeld, het getal 214 geplaatst, en in register 2 het getal 8236. Het resultaat van een vermenigvuldiging heeft de dubbele lengte en staat in de accumulator en in register 2. Daartoe moet het getal B plaats maken voor het resultaat. Dit is mogelijk omdat, na het vermenigvuldigen met het eerste cijfer, de 6, dit cijfer niet meer nodig is. We kunnen deze 6 dus verwijderen en het resterende getal, namelijk 823, naar rechts verschuiven, zodat de 3 op de plaats van de 6 komt. Dit heeft nog het voordeel dat we altijd naar de laatste plaats moeten kijken voor de vermenigvuldiging.

Bij het vermenigvuldigen van binaire getallen is het vermenigvuldigen bijzonder eenvoudig. Het vermenigvuldigen met 0 betekent dat wij niets bij de inhoud van de accumulator behoeven op te tellen; bij het vermenigvuldigen met 1 moeten wij de inhoud van het register 1 bij de accumulator optellen. Er is dus in feite op geen enkele wijze een vermenigvuldigtabel noodzakelijk. Positieve binaire getallen kan men aldus op een simpele wijze vermenigvuldigen. Een voorbeeld daarvan geeft fig. 8. In deze figuur worden de getallen 9 en 13 in binaire vorm met elkaar vermenigvuldigd. Men bedenke bij het beschouwen van deze figuur, dat elk getal uit 4 bits bestaat.

Delen

Het delen geschiedt door in de accumulator en register 2 (zie fig. 9) het deeltal te plaatsen. In register 1 bevindt zich de deler.

Het deelproces verloopt nu als volgt. Men trekt de deler af van de inhoud van de accumulator en stelt vast of het resultaat positief, dan wel negatief is (0 wordt als positief gerekend). Is het resultaat negatief, dan is de aftrekking mislukt en telt men de inhoud van de deler weer bij de accumulator op. Is het resultaat positief, dan is de deling wel gelukt en schuift men de inhoud van de accumulator en register 2 één plaats op. Het eerste cijfer, in dit geval een bit van het quotiënt, wordt rechts in register 2 geplaatst; deze plaats is immers vrijgekomen. In figuur 9 is een aftrekking niet uitgevoerd als deze zou mislukken, zodat de correctieslagen niet zijn weergegeven, maar in de plaats daarvan is dan een direct schuiven aangegeven. Op zichzelf is deze methode niet zo vreemd, want bij het rekenen is het heel wel mogelijk om een aftrekking uit te voeren zonder het resultaat in de accumulator te plaatsen. Als het resultaat negatief blijkt te zijn, dan wordt de inhoud niet in de accumulator gekopieerd, maar men verschuift eenvoudig de inhoud van de accumulator, en wel zo lang tot het resultaat van de aftrekking positief is. Is dat het geval, dan kopieert men het resultaat van de aftrekking in de accumulator. Na afloop van de deling vindt men in de rechtse 8 bits het quotiënt en treft men in de linker 8 bits de rest aan.

Drijvende-komma notatie (floating point)

Bevatten decimale getallen delen die kleiner zijn dan 1, dan is men gewend deze delen uit te drukken in tienden, honderdsten, duizendsten, enz. Om deze te scheiden van de eenheden en tientallen, plaatst men na de eenheden een komma. Bij het optellen en aftrekken van getallen plaatst men deze komma's onder elkaar, omdat op deze wijze wordt verzekerd dat de grondtallen met dezelfde macht ook onder elkaar komen te staan. In rekenmachines is het niet zonder meer mogelijk om komma's onder elkaar te plaatsen. Sterker nog, bij het rekenen zijn in het geheel geen komma's aanwezig en dient de programmeur rekening te houden met de plaats van de komma door het getal in feite in twee delen te splitsen, namelijk het deel met de eenheden, tientallen, enz. en een ander deel, op een andere plaats, met de tienden, honderdsten, enz. De laatste plaats, d.w.z. de minst significante rechts van de komma, geeft de absolute nauwkeurigheid van het getal aan. De verhouding tussen deze waarde en het meest significante cijfer geeft de relatieve nauwkeurigheid aan.

Voorbeeld: 6 plaatsen:		698,583
	kleinste absolute waarde (groter dan 0)	0,001
	grootste absolute waarde	999,999

Als zowel positieve als negatieve getallen in een register moeten kunnen worden geregistreerd, dan moet dit een bereik hebben van – 999,999 tot 999,999, waarbij dan de absolute nauwkeurigheid gelijk is aan 0,001. De verhouding tussen de grootste en de kleinste waarde (hierbij is dan gerekend de absolute waarde) is dan $1 : 10^6$.

Bij veel rekenproblemen is het noodzakelijk een groter gebied te bestrijken dan deze $1 : 10^6$. Men doet dit dan onder opoffering van de absolute nauwkeurigheid. Neem als voorbeeld het getal 698,583. Dit getal kan men ook schrijven als $0{,}698583.10^3$. Als we deze notatie bekijken, dan valt het op dat daar in feite een factor aan toegevoegd is, namelijk 10^3. Enkele andere voorbeelden zijn:

$$\begin{aligned} 738{,}25 &= 0{,}73825 \,.\, 10^3 \\ 0{,}076543 &= 0{,}76543 \,.\, 10^{-1} \\ 3{,}5875 &= 0{,}35875 \,.\, 10^1 \\ 0{,}65205 &= 0{,}65205 \,.\, 10^0 \end{aligned}$$

In deze voorbeelden valt het op dat rechts van de = tekens alle getallen beginnen met een 0, vervolgens een komma en daarna een cijfer. Men zegt, dat deze getallen genormeerd zijn. In wezen spelen de nullen en de komma's geen rol meer, evenmin als de getallen 10. Men kan nu dezelfde serie getallen op en andere wijze voorstellen, b.v.:

738,25	= 3	73825
0,07654	= 9	76540*
3,5875	= 1	35875
0,65205	= 0	65205

* Het cijfer 9 stelt hier het 10-complement voor van de macht 10^{-1}.

Het eerste cijfer stelt nu de macht van 10 voor en de overige cijfers het getal achter de komma. De exponenten van 10 worden vaak gekozen in het 10-complementaire systeem, d.w.z. binnen het gebied van –5 t/m +4. Het kleinste getal wordt dan:

$$5\ 10000 = 0{,}1 \,.\, 10^{-5} = 10^{-6},$$

en het grootste getal:

$$4\ 99999 = 0{,}99999 \,.\, 10^4 = 9999{,}9$$

De verhouding tussen het grootste en het kleinste getal is nu 10^{10}, en de relatieve nauwkeurigheid is nu slechts $1 : 10^4$.

Andere notatievormen van tweetallige codes

Naast de bekende binaire vorm wordt ook vaak de z.g. hexadecimale voorstellingsvorm van binaire getallen gebruikt. De bits worden nu in groepen van vier onderverdeeld. Elke groep heeft dan uiteraard 16 verschillende combinatiemogelijkheden, namelijk 0000 t/m 1111. In plaats van deze voorstellingen, in de binaire vorm dus, gebruikt men nu de symbolen 0 t/m 9, resp. A t/m F. F wordt gebruikt voor het getal 1111. In wezen is 16 het grondtal van dit talstelsel. Op zichzelf levert deze voorstellingsmethode voor microprocessoren geen voordelen, behalve dan dat men eenvoudig een code kan lezen en men bij het invoeren in oefenmechanismen m.b.v. 16 toetsen de groep van 4 bits eenvoudig in één keer aan de machine kan toevertrouwen. In de onderstaande tabel zijn de groepen van 4 bits gerangschikt met het erbij behorende hexademicimale teken en de overeenkomstige binaire waarde.

binair		*hexadecimaal*	*decimaal*
Gewicht 8 4 2 1			
0 0 0 0	=	0	0
0 0 0 1	=	1	1
0 0 1 0	=	2	2
0 0 1 1	=	3	3
0 1 0 0	=	4	4
0 1 0 1	=	5	5
0 1 1 0	=	6	6
0 1 1 1	=	7	7
1 0 0 0	=	8	8
1 0 0 1	=	9	9
1 0 1 0	=	A	10
1 0 1 1	=	B	11
1 1 0 0	=	C	12
1 1 0 1	=	D	13
1 1 1 0	=	E	14
1 1 1 1	=	F	15

Voorbeelden:

binair	1011	0101
hexadecimaal =	B	5

De BCD-code

De BCD-code is een afkorting van *B*inary *C*oded *D*ecimal. In de BCD-code worden de cijfers 0 t/m 9 in de overeenkomstige binaire combinatie met de gewichten 8 4 2 1 omgezet. De onderstaande tabel geeft de decimale cijfers 0 t/m 9 met de daarbij behorende voorstelling in de binaire code:

	gewichten			
decimaal	*8*	*4*	*2*	*1*
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

De binaire codes 0 t/m 15, die in de hexadecimale code worden voorgesteld door A t/m F, worden dus niet gebruikt.

Getallen behoeven nu niet meer voor rekenmachines te worden omgezet in de binaire code, maar men stelt de cijfers van een decimale code voor, door deze reeks van vier nullen en enen. Zo wordt b.v. het getal 735 voorgesteld als:

0111 0011 0101

De meeste microprocessoren kunnen met deze BCD-getallen werken en zorg dragen voor de overdracht van een 10 met de z.g. decimal adjust (DA)-instructie.

We zullen bij wijze van voorbeeld twee getallen beschouwen, b.v. 92 en 89. Als we deze getallen optellen, dan zal bij decimale optelling een 1 overgedragen worden van de eenheden naar de tientallen en een 1 van de tientallen naar de honderdtallen. We schrijven deze code echter in binaire vorm, en wel als volgt:

1001 0010 en 1000 1001

Uit deze schrijfvorm blijkt dat indien de getallen binair worden opgeteld, er nu geen overdracht geschiedt van de eenheden naar de tientallen, doch wel van de tientallen naar de honderdtallen. De oorzaak hiervan is daaraan toe te schrijven dat er bij binair rekenen uitsluitend een overdracht plaats heeft als bij een groep van 4 bits de waarde 16 of meer voorkomt na de optelling. De overdracht van de 1 betekent echter dat er 16 aan de desbetreffende groep van 4 bits wordt onttrokken. In dit geval is het onttrekken van 10 noodzakelijk, maar de onttrekking van een 6 is teveel van het goede. Voor de volgende groep van 4 bits maakt het niets uit, daar deze – zoals het in decimale code gebruikelijk is – eenvoudig een 1 ontvangt. Deze moeilijkheden zijn te omzeilen door na de decimale optelling nog een 6 op te tellen bij elke groep van 4 bits. Er worden dus even veel zessen bij de sommen geteld als dat er cijfers zijn. Zie onderstaand voorbeeld:

```
      1001  0010      (9 2)
      1000  1001      (8 9)     (92 + 89 = 181)
      ----  ----  +
1  | 0001  1011      (1 11)    (16 + 1 = 17,
      0110  0110      (6 6)      dus 16 onttrokken
      ----  ----  +
      0111  0001      (7 1)
        ^
        1 <-- carry
0001  1000  0001 = 181
```

Het resultaat is nu dus correct.

Op die plaatsen waar een overdracht van een 1 plaats heeft, is nu het resultaat van de optelling correct, daar de te grote onttrekking d,m.v. 16 is gecompensenseerd door een 6 extra toe te voegen. Op die plaatsen echter waar nu een "Carry" ontstaat, zoals bij de eenheden, is de optelling ook volkomen correct verlopen.

Het ligt echter anders als een optelling geen overdracht van een 1 heeft. Dan is het resultaat na het optellen van de zessen 6 teveel geworden. Deze moeilijkheden worden opgeheven door na deze optellingen een z.g. Decimal Adjust Instructie (DA) uit te voeren Deze instructie kijkt of er een 1 is overgedragen. Is dit niet geschied, dan wordt er alsnog op de desbetreffende plaats een 6 van het resultaat afgetrokken. Hierdoor worden op die plaatsen, waar geen overdracht van een 1 gewenst was, weer een correctie uitgevoerd. Zie onderstaand voorbeeld.

```
      1001  0010      (9 2)
      0110  0101      (6 5)      (92 + 65 = 157)
      ----  ----  +
      1111  0111      (15 7)
      0110  0110      (6 6)
      ----  ----  +
      0101  1101      (5 13)
        ^
1  |    0           carry
^  |    0110        DA
   ----  ----  -
0001  0101  0111 = 157
```

Ook in dit geval is het resultaat dus correct.

De ASCII-code

De ASCII-code is een afkorting van *A*merican *S*tandard *C*ode for *I*nformation *I*nterchange.

De ASCII-code (de Amerikaanse versie van de ISO-code) is de meest gebruikte en daardoor de meest bekende code. Op bepaalde posities komen een aantal andere tekens voor. In fig. 10 is deze code volledig weergegeven. Fig. 11 geeft een vertaling van deze code in de hexadecimale code en omgekeerd.

De EBCDIC-code

Moderne computers werken vaak met de EBCDIC-code. De EBCDIC-code is een afkorting van *E*xtended *B*inary *C*oded *D*ecimal *I*nterchange *C*ode.

Het is een 8-bits code, waarbij 264 coderingen mogelijk zijn, namelijk de bekende 256 (2^8) combinaties, terwijl men bovendien acht combinaties gebruikt die gevormd worden door twee groepen van 8 bits. De eerste groep van de 8 bits is dan de ESC-code, wat een afkorting is van "escape". Deze ESC geeft aan dat de nu volgende 8 bits anders geïnterpreteerd dienen te worden. Het is een redelijk vrije combinatie, die nog heel veel toepassingen kan gaan vinden.

Een deel van de afspraken van deze code wordt gebruikt voor het weergeven van hoofd- en kleine letters, cijfers en tekens, zie fig. 12. De EBCDIC-code bestaat uit twee gebieden, namelijk de bits 1, 2, 3 en 4 die een bepaalde groep aanduiden, de z.g. zône, en de bits 5, 6, 7 en 8 die een numerieke presentatie geven. De bits op de plaatsen 8 en 7 worden gebruikt om aan te geven welk type letter, cijfer of teken wordt toegepast. Hieronder volgt daarvan een overzicht.

bitpositie 8 – 7		betekenis
0	0	niet in gebruik
0	1	speciale tekens
1	0	kleine letters
1	1	hoofdletters en cijfers

De bits op de plaatsen 6 en 5 geven op hun beurt een onderverdeling van deze 4 mogelijkheden.

bitpositie 6 – 5		betekenis
0	0	A t/m I
0	1	J t/m R
1	0	S t/m Z
1	1	cijfers

Hieruit blijkt het alfabet in onderdelen te worden gesplitst, namelijk A t/m I, J t/m R, S t/m Z, en tien cijfers. De overige 4 bits, namelijk 1, 2, 3 en 4 geven in numerieke waarden het cijfer, resp. de plaats van de letter in een groep. Zo is de hoofdletter A de eerste in een groep, de hoofdletter B de tweede, enz. Op deze wijze verkrijgt men voor de kleine letter a.

bits	1	2	3	4	5	6	7	8
a =	1	0	0	0	0	0	0	1

Voorbeeld:

Jan	10001011	10000001	10101001
	J	a	n

LOGISCHE FUNCTIES

De bases voor alle processoren zijn logische functies. De meest voorkomende zijn:

EN – functie
OF – functie
Exclusieve OF-functie (EXOR)
INVerse functie.

Laten we twee variabelen A en B beschouwen, die elk slechts de waarden "0" en "1" kunnen aannemen. Dat dit vaak voorkomt, kon men waarnemen bij het rekenen met binaire getallen.

De bovenstaande functies zijn operatoren en geven dus een bepaalde operatie tussen een tweetal variabelen (A en B) weer. In onderstaande diagrammen zijn enkele functies weergegeven:

EN	A 0	A 1
B 0	0	0
B 1	0	1

OF	A 0	A 1
B 0	0	1
B 1	1	1

EXOR	A 0	A 1
B 0	0	1
B 1	1	0

INV	
0	1
1	0

Deze vorm van voorstellen zijn wij gewend bij het optellen en vermenigvuldigen van getallen. Men kan ze ook op een iets andere wijze weergeven in de vorm van z.g. carthesische produkten, zie hieronder. Onder deze tabel zijn de formules van de betreffende functies weergegeven, namelijk A . B, A + B, A EXOR B en NOT A (inverse).

A	B	EN
0	0	0
0	1	0
1	0	0
1	1	1

A . B

A	B	OF
0	0	0
0	1	1
1	0	1
1	1	1

A + B

A	B	EXOR
0	0	0
0	1	1
1	0	1
1	1	0

$A \oplus B$

A	INV
0	1
1	0

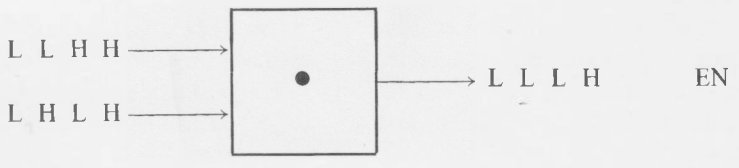
$\overline{A}$

De beperking bij A + B is dat 1 + 1 eveneens 1 is.

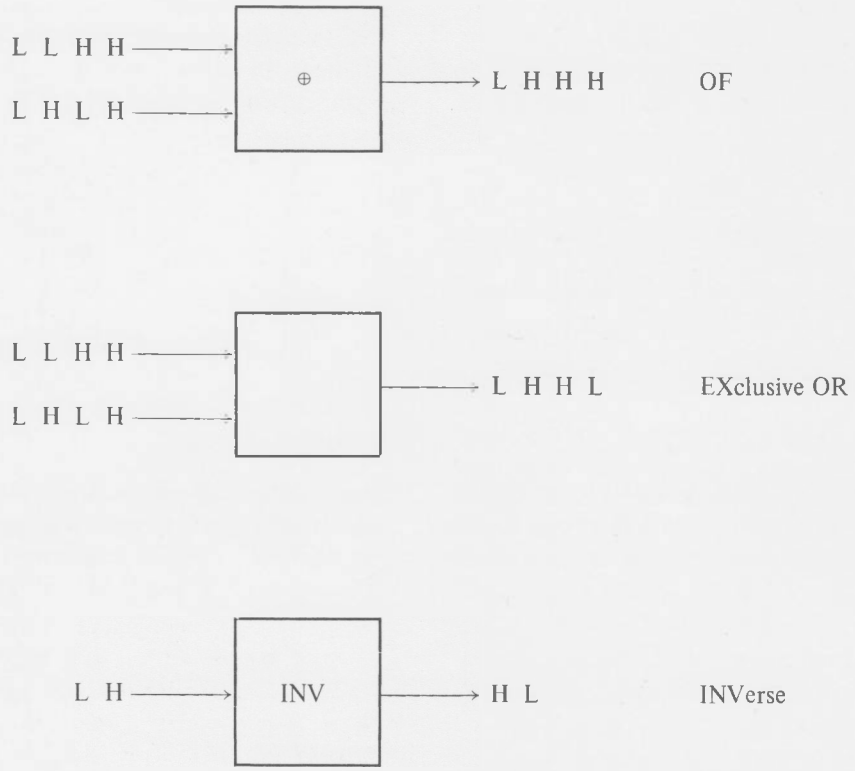
De waarden van deze variabelen, namelijk de 0 en de 1, kunnen gerelateerd worden aan b.v. "waar" en "onwaar" (true en false). Het zijn in tabellarische vorm weergaven van uitspraken, z.g. "beweringen". Het resultaat bij een "EN"-operatie is dat de bewering pas "waar" is, als beide deelbeweringen "waar" zijn. Bij een "OF" is de uiteindelijke bewering "waar" als één van de beide beweringen "waar" is. Bij de EXclusive-OR is het resultaat "waar" als er één bewering "waar" is, maar niet als beide "waar" zijn. De EXclusive-OR vertoont in grote mate overeenstemming met het optellen van bits in een binair talstelsel. De "EN" met twee enen correspondeert met de z.g. "1 onthouden". Het geheel vormt een onderdeel van de z.g. Boole Algebra, die zich bezig houdt met het formaliseren van uitspraken.

De hierboven vermelde functies kunnen m.b.v. zogenaamde logische elektronische elementen door elektronische signalen worden voorgesteld. Duiden we de 1 aan m.b.v. een "hoge" elektrische spanning en de "0" m.b.v. een "lage" elektrische spanning, dan kan men het elektronische element als volgt beschrijven voor het geval dat het een "EN"-schakeling is; hieruit blijkt dat het patroon van nullen en enen identiek is aan de signalen "laag" (L) en "hoog" (H).

Een soortgelijk elektronisch element kunnen we ons



voorstellen voor de logische "OF"- en voor de "EXclusive-OR", evenals voor de "INVerse"-schakeling:



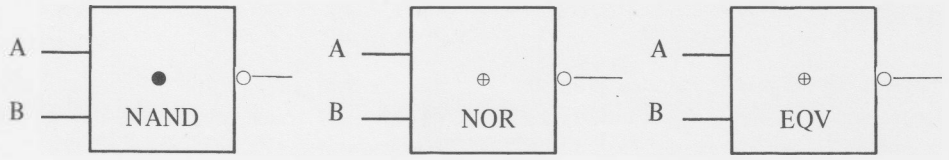
Met behulp van bovenstaande elementen kan men gemakkelijk de volgende karakteristieke eigenschappen bepalen:

$A + 0 = A$
$A + 1 = 1$
$A + A = A$
$A + \overline{A} = 1$
$A + B = B + A$
$A + B + C = (A + B) + C$
$A + B.C = (A + B) . (A + C)$

$A.1 = A$
$A.0 = 0$
$A.A = A$
$A.\overline{A} = 0$
$A.B = B.A$
$A.(B.C) = (A.B) .C$
$A.(B + C) = A.B + A.C$

Men kan de bovenstaande stellingen snel verifiëren aan de hand van de schakelingen. Men vult daartoe slechts overeenkomstige tabellen in, zoals bij de functie-beschrijving van de componenten al eerder is geschied. Dergelijke tabellen heten "waarheids"-tabellen.

De "EN", "OF" en "EXclusive-OR" worden ook vaak gebruikt met een invers element ermee in serie geschakeld. Het zijn dan de z.g. "NAND", "NOR"- en "equivalent" (EQV)-schakelingen. De waarheidstabellen zijn hieronder weergegeven.



A B	NAND
0 0	1
0 1	0
1 0	0
1 1	0

$\overline{A} . \overline{B} = \overline{A+B}$

A B	NOR
0 0	1
0 1	0
1 0	0
1 1	0

$\overline{A+B} = \overline{A} . \overline{B}$

A B	EQV
0 0	1
0 1	0
1 0	0
1 1	1

$A = B$

De bovenstaande functies zijn speciale gevallen van de "stellingen van de Morgan", volgens welke "de inverse van een som van termen gelijk is aan het produkt van de inverse termen" en "de inverse van een produkt gelijk is aan de som van de inverse termen".

Combinatie van deze schakelingen kan een binaire optelleer voor twee bits vormen, waarbij een "1" onthouden van de vorige trap aanwezig is en een "1" onthouden, naar de volgende trap wordt doorgegeven, zie fig. 13. In de bovenste EXclusive-OR wordt vastgesteld of het resultaat een "1" resp. een "0" is van de beide bits a en b. De "1", afkomstig van de vorige tel-elementen, wordt hierbij weer m.b.v. de EXclusive-OR opgeteld, waarna men een deelsom van het geheel krijgt. Indien a en b beide "1" zijn, dan zal er een "1" onthouden dienen te worden. Of ze beide "1" zijn, wordt vastgesteld in de met a en b verbonden "EN"-schakeling, en via een "OF"-schakeling doorgegeven naar het volgende tel-element. Zijn a of b "1", dan is ook de uitgang van de betreffende "EXclusive-OR"-schakeling "1" en zal bij een "1" op de uitgang van de vorige teltrap wederom een overdracht van een "1" moeten ontstaan. Dit wordt vastgesteld in de bijbehorende "EN"-schakeling en via de "OF"-schakeling doorgeven naar de volgende trap. Een dergelijk element heet in het engels een "full-adder". Een aantal van deze onderling verbonden elementen vormen een optelmechanisme voor een rekenmachine (zie fig. 14). De carry-signalen C_o ... C_n vormen de overdrachten van "enen onthouden".

Een ander logisch element vormt de z.g. "flip-flop". Een flip-flop is een bistabiele multivibrator, een elektronisch element dat het best te vergelijken is met een lichtschakelaar, die in de stand "aan" of "uit" kan staan. De mechanische bestuurde schakelaar (d.m.v. een vinger) kan nu met behulp van elektrische signalen bestuurd worden. Een "hoog" signaal op de z.g. RESET-ingang stuurt deze flip-flop in wat genoemd wordt de logische "0" en een "hoog" signaal op de z.g. SET-ingang stuurt hem in de logische "1". De flip-flop heeft twee uitgangen, die complementair zijn, d.w.z. als de flip-flop in de "1" staat, zal een van beide uitgangen "hoog" zijn en de andere "laag". Staat de flip-flop daarentegen in de "0", dan zal het omgekeerde het geval zijn. Fig. 15 toont een dergelijke "SET-RESET" flip-flop gevormd door 2 "NOR" schakelingen. Normaal zijn de SET en de RESET beide "laag", d.w.z. een logische "0". Dan kan de Q of $\overline{Q}$ "hoog" zijn, d.w.z. een logische "1", dan wel "laag", d.w.z. een logische "0". Men wordt nu uitgenodigd, aan de hand van een functietabel met SET en RESET, te bepalen wat er met de uitgangen Q en $\overline{Q}$ gebeurt. Het is daarbij niet toegestaan, om beide ingangen, de SET en RESET dus, tegelijk "1" te maken.

Een bijzonder soort flip-flop is de z.g. D-flip-flop, die uitsluitend een SET-signaal kent. Dit SET-signaal wordt echter niet direct gebruikt voor de besturing van de flip-flops, maar kan slechts gebruikt worden op een bepaald tijdstip, namelijk wanneer over een stuurlijn (de z.g. "klok"-lijn) een signaal wordt gegeven. Bij de wisseling van dit signaal, b.v. van "laag" naar "hoog", kopiëert de D-flip-flop de toestand van de stuurlijn. Ook zijn er flip-flops waarbij deze kopiëring geschiedt als een signaal van "hoog" naar "laag", d.w.z. bij de dalende flank van de klokimpuls (zie fig. 16).

Een combinatie van een aantal van deze flip-flops vormt een z.g. register, waarvan al eerder sprake was (zie fig. 17). Een zeer groot aantal van deze registers, samen ondergebracht in een schijfje silicium, vormt een geheugen. Om de juiste plaats van de informatie in dit geheugen vast te stellen, is een adres nodig dat in binaire vorm wordt verstrekt (zie fig. 18). Voorts is het mogelijk, aan dit geheugen op te dragen informatie van de buitenwereld over te nemen, d.w.z. de ingangen te kopiëren, dan wel de inhoud van een register naar buiten kenbaar te maken, het z.g. "lezen". In moderne geheugens is het lezen niet destructief, d.w.z. als men de inhoud leest, blijft deze (evenals bij een krant) ongewijzigd.

PROCESBESCHRIJVING

Voor een probleem dat men op wil lossen, zal men in eerste benadering trachten richtlijnen te vinden volgens welke men het probleem kan beschrijven in een algoritme. Deze richtlijnen kan men verfijnen, zodat ze tenslotte het algoritme vormen dat de beschrijving is van de methodiek voor het vinden van de oplossing van het probleem.

Volgens ons huidig inzicht betreffende automaten, zal een algoritme aanleiding geven tot een aantal beweringen, de zogenaamde statements, die aanleiding zijn tot een proces dat stap-voor-stap wordt uitgevoerd. Dit is dan een zogenaamd sequentieel proces. Een sequentieel proces is in zijn algemeenheid niet onvermijdelijk. Het is een systeem dat gebonden is aan een processorvorm.

Parallelprocessen worden b.v. uitgevoerd op analoge rekenmachines. Deze bootsen in de natuur optredende processen zo goed mogelijk na. De natuur zelf is een gigantisch proces dat bestaat uit een enorm aantal parallel en sequentieel verlopende processen.

Na het opstellen van een algoritme, moet dit uitgedrukt worden in een vorm die het liefst onze expressievormen zo dicht mogelijk benadert. Deze uitdrukkingsvorm is onderworpen aan strenge regels wat betreft de ondubbelzinnigheid. Met een dergelijke beschrijving is het mogelijk de reeks opdrachten op te stellen die voor een sequentiële automaat voldoende zijn om een gewenst proces uit te voeren.

De beschrijvingsmethode die nauw aansluit bij onze expressievorm, noemt men een hogere programmeertaal. Bijna nooit kan de processor deze taal direct gebruiken voor de uitvoering van het door ons gewenste proces. De taal van de machine sluit alleen maar aan bij de hogere programmeertaal. Met behulp van een zogenaamde compiler kan de machine de uitdrukkingen van deze hogere taal transformeren naar zijn eigen, primitievere, taal, die gericht is op machinefuncties.

Deze compiler is een programma, en het door ons geschreven programma in de hogere taal is data voor het compilerprogramma die ingevoerd wordt. We krijgen nu het volgende beeld van de tijdloze beschrijvingen:

a. probleembeschrijving
↓
b. formeel (event. wiskundig) model
↓
c. algoritmisch model
↓
d. programma in hogere taal
↓
e. programma in machinetaal

De punten a, b en c behoren tot het systeemontwerp en de systeembeschrijving; d behoort tot het gebied van het coderen. De overgangen van a tot en met d worden door ons zelf verricht. De transformatie van d naar e is het werk van de processor onder de besturing van de compiler.

Het volgende voorbeeld geldt voor het ontwikkelen van een procesbeschrijving met behulp van een algoritme. Deze procesbeschrijving is in een hogere programmeertaal gewenst. In dit voorbeeld wordt gezocht naar het maximum van een gemeten grootheid, b.v. de hoogste spanning, stroom, temperatuur, enz. van een groot aantal meetresultaten. De metingen zijn verricht op elkaar opvolgende tijdstippen. Noem de meetwaarden in volgorde van het meten a(1), a(2), a(3) ... a(n). Gevraagd wordt nu naar een algoritme voor het vinden van de grootste meetwaarde. Het zou kunnen zijn:

neem het eerste getal a(1),
neem het tweede getal a(2),
vergelijk beide,
bewaar het grootste getal,
neem het derde getal,
vergelijk dit met het grootste getal,
bewaar het grootste getal,
enz

We hebben voortdurend het grootste getal tot onze beschikking. Het statement "bewaar het grootste" is een daad die het verloop telkens bepaalt. Het kan zijn dat het "grootste" ongewijzigd kan blijven, maar het kan ook zijn dat de andere meetwaarde, waarmee het vergeleken wordt, de nieuwe "grootste" wordt. Het "grootste" verandert dus indien nodig. Het is een variabele, noem deze GR. Deze GR bevindt zich ergens in het geheugen van de machine; waar precies, is nog niet interessant. Het is alleen van belang dat de machine telkens GR oplevert als erom gevraagd wordt. Om te beginnen wordt GR = a(1) gemaakt. Beweerd wordt nu dat GR kleiner is dan de waarde waarmee wordt vergeleken. Dit is waar (true, T) of onwaar (false, F).

In fig. 19 treffen we het stroomdiagram aan. We kunnen dat opschrijven in een taal die de elementen bezit die in dit stroomschema voorkomen.

GR:=0; (:= betekent: wordt gelijk aan)

ALS GR < a(1) DAN GR:=a(1);
ALS GR < a(2) DAN GR:=a(2);
ALS GR < a(3) DAN GR:=a(3);

enz.

De ; dient om een duidelijke scheiding aan te geven.

In het voorgaande programma zijn een aantal beweringen (statements) neergeschreven. De eerste regel geeft aan GR een waarde. We noemen dit een toekenning. Woorden zoals *dan, als* enz. worden vaak onderstreept. Men noemt ze sleutelwoorden. Deze sleutelwoorden zijn aanwijzingen voor de compiler.

In het behandelde programma komen een groot aantal, vrijwel identieke, beweringen voor. Ze verschillen alleen in de naam van de variabele en dat nog alleen voor de waarde tussen de haken, de zogenaamde index. Met het wijzigen van de index lopen we als het ware de lijst met waarden met onze vinger na, vandaar de naam index.
De algemene vorm is:

```
ALS GR<a(i) DAN GR:=a(i);
```

We moeten i een beginwaarde geven, én om bij te kunnen houden waar we zijn, moeten we de index i bijwerken. Voorts stellen we vóór het begin GR=0. Dit mag alleen als we zeker weten dat a(i) $\geqslant$ 0 is, daar het anders de indruk wekt dat 0 de grootste waarde is. Dan kan GR = a(1) gesteld worden en i = 2. Nu wordt het programma:
Na de actie i := i+1 is de variabele i één groter geworden, d.w.z. we kiezen de volgende meetwaarde.

```
BEGIN GR:=0; i:=1;
ALS   GR<a(i) DAN GR:=a(i);
      i:=i+1

EINDE
```

Hoe weten we nu of alle meetwaarden (n stuks) behandeld zijn? Als de nieuwe i = n+1, dan hebben we zojuist de laatste meetwaarde gehad. Zolang i < n+1, moeten we opnieuw beginnen. Dit kunnen we doen met een zogenaamde sprong naar het begin van de berekening. Waar dit begin is, duiden we aan met een zogenaamd label, die we hier "start" zullen noemen, zie het stroomdiagram van fig. 20.

```
BEGIN GR:=0; i:=1;
START: ALS GR<a(i) DAN GR:=a(i);
       i:=i+1;
       ALS i<n+1 GA DAN NAAR START
       DRUK DE GROOTSTE WAARDE AF

EINDE
```

Het geheel is een zogenaamde lus. *Er dient altijd voor gezorgd te worden, dat er een voorwaarde aanwezig is om een lus te beëindigen.* Men bereikt dit hier door de expressie i := i+1.

In plaats van de grootste waarde te onthouden, kunnen we ook onthouden welke a(i) de grootste waarde bevat. Stel deze a(j). Nu kan b.v. de OMDAT-constructie gebruikt worden:

```
BEGIN j:=1; i:=1;
  OMDAT i≤n DOE
    BEGIN als a(j)<a(i)
      DAN j:=i;
    j:=i+1
    EINDE
  DRUK AF a(j)
EINDE
```

DIGITALE APPARATUUR

Digitale processor-systemen bestaan uit een groot aantal elektronische en mechanische componenten die men tot op heden verdeelt in een aantal groepen:

a. de centrale besturing (CPU)
   - rekenkundig/logisch orgaan (ALU)
   - instructieteller
   - instructieregister
   - dataregisters
   - toestandregisters

b. het werkgeheugen

c. periferie-apparatuur
   - in- en uitvoerapparatuur
   - achtergrondgeheugens

De technologische ontwikkeling en de prijs/prestatie verhouding zijn thans aan grote wijzigingen onderhevig, waardoor het nauwelijks mogelijk zal zijn een "architectuur" van een systeem als algemeen juist te accepteren. De architectuur is niet alleen afhankelijk van de onderdelen en de prijs ervan, maar ook van het toepassingsgebied waarvoor de machine het meest gebruikt zal worden. Zo zal in een machine waarvan de bewerkingen voornamelijk bestaan uit het verrichten van vermenigvuldigingen, een dergelijke vermenigvuldiger in elektronische componenten uitgevoerd worden (16 x 16 bits is minder dan 160.10^{-9} s). Een machine waarmee vrijwel niet vermenigvuldigd wordt, zal hiervoor geen specifieke onderdelen bezitten, maar het vermenigvuldigen (relatief traag) door middel van een programma realiseren.

Verder speelt de grootte en de snelheid van het zoge-

naamde werkgeheugen een belangrijke rol. Een groter geheugen betekent immers meer plaatsen, maar ook grotere adressen om data en programma's te adresseren. Men kan een grote woordlengte kiezen, waardoor men per toegang tot het geheugen veel data verkrijgt. De data per woord van een geheugen met een grote woordlengte moet dan wel in zijn geheel bruikbaar zijn, anders daalt het rendement van het geheugen. Als b.v. de woordlengte 32 bits is en de instructielengte slechts 20 bits, dan worden 12 bits per woord niet gebruikt, en doet slechts 60% van het programma-geheugen dienst.

Ook van belang zijn factoren zoals de hoeveelheid periferie-apparatuur, de vraag of veel dan wel weinig datatransport nodig is en de mate van verscheidenheid van datastructuren.

Geheugens

Dit hoofdstuk geeft een globaal overzicht wat betreft uitvoering, grootte, snelheid, prijs, enz. van verschillende geheugens die bij digitale processoren worden toegepast.

De functie van een geheugen in een processorsysteem is het registreren van data gedurende een zekere tijd. Deze data kan veelsoortig zijn en bestaan uit:

instructies
getallen
logische waarden
teksten

In een processorsysteem komen verschillende soorten van geheugens voor. De hiërarchische volgorde is (aangevende hoe "dicht" een geheugen bij de CPU staat):

flip-flops registers cash memories werkgeheugens	halfgeleider-geheugens
schijven- en trommelgeheugens magnetische banden	magneto-mechanische geheugens

Er is een relatie tussen het gebruikte aantal geheugenelementen, de kosten per geheugenelement en de toegangstijd voor het verkrijgen, resp. registreren van data in een dergelijk geheugenelement (zie fig. 21).

Uit de figuur blijkt dat het aantal bits in een systeem aanzienlijk toeneemt als de kosten per bit gering zijn. Oorzaak en gevolg moeten echter niet verwisseld worden. Er bestond een sterke behoefte aan grote geheugens, die dan uiteraard zeer goedkoop dienden te zijn. Dit heeft de ontwikkeling van magnetische geheugens sterk bevorderd.

Het is duidelijk dat de gewenste capaciteit en de kosten van een registratieruimte bepalen welk type geheugen wordt gekozen. De keuzecriteria zijn de laatste jaren sterk gewijzigd en zijn nog altijd aan grote wisselingen onderhevig.

Geheugenelementen

Binaire informatie kan opgeslagen worden in een verzameling binaire elementen. Ieder element (geheugen) van deze verzameling bevat een 1 of een 0. Vanwege de hoge snelheden waarmee processoren werken (in de orde van grootte <1 μs per actie) komen voor deze elementen alleen elektronische schakelingen in aanmerking.

Flip-flops

Een veel toegepast geheugenelement is de al beschreven flip-flop. Vaak kan in een processorsysteem de informatie op een bepaald punt afkomstig zijn van diverse flip-flops, waarvan er dus b.v. slechts één gekozen mag worden, zie fig. 22. Dit zou betekenen dat parallel lopende lijnen zouden moeten lopen van iedere flip-flop naar een OR-poort voor dat punt. Om te kunnen volstaan met één enkele lijn die langs alle flip-flops loopt, worden de collectoren van de laatste transistor van de flip-flops rechtstreeks uitgevoerd ("wired or") en via een gezamenlijke weerstand R gevoed, zie fig. 23. De onderling verbonden collectors vormen dan de inverse ingang voor de volgende schakeling. Vanwege de lekstroom van de transistoren is de uitgangsspanning, als alle transistoren geblokkeerd zijn, afhankelijk van het aantal transistoren. Om het juiste niveau op de uitgangsklem Uit te krijgen, moet R dan ook aangepast worden aan het aantal transistoren. Staan veel transistoren parallel, dan moet R klein zijn en wordt de stroom door één geleidende transistor dus groot. Daarom is bij de wired-or het aantal parallel te schakelen flip-flops (fan-out) beperkt. Een andere, veel toegepaste methode, is de tri-state uitgang. Iedere flip-flop wordt hierbij gevolgd door twee transistoren in serie, zie fig. 24. Via EN-poorten en een OE-commando (Output Enable), wordt één van beide transistoren in geleiding gebracht. Als OE=0 dan zijn beide transistoren geblokkeerd en zweeft de uitgang (hoogohmig). Bij deze schakeling wordt de uitgang dus door één van de parallel geschakelde flip-flops laagohmig aan aarde of aan de voedingsspanning gelegd. Via de besturing moet ervoor gezorgd worden dat tegelijkertijd maar één van de flip-flops geselecteerd wordt.

Dynamische geheugenelementen

Om meer geheugenelementen in een zelfde ruimte onder te brengen, worden dynamische geheugenelementen toegepast. Deze elementen bestaan uit slechts één transistor. Via deze transistor wordt de lading op een condensator vastgehouden. Een nadeel is dat deze lading langzaam weglekt. Daarom zijn geheugens met deze dynamische elementen altijd voorzien van een refresh-schakeling. Door middel van deze schakeling wordt elk element periodiek afgetast en de weggelekte lading weer aangevuld.

Read-only geheugenelementen

Uit een read-only geheugenelement kan data alleen maar gelezen worden. Deze elementen worden o.a. gebruikt om hulpprogramma's die niet gewijzigd behoeven te worden, op te slaan. Geheugens met deze elementen worden ROM genoemd (*R*ead *O*nly *M*emory).

Registers

Een register is een verzameling flip-flops met een gemeenschappelijke besturing. Een register is dus een geheugen voor een binair woord (een binaire vector).

In fig. 25 is een schematische voorstelling gegeven van een register bestaande uit 8 D-flip-flops met tri-state uitgang.

Zijn verscheiden registers op dezelfde bedrading aangesloten, dan wordt deze bedrading een bus genoemd. In fig. 26 is een bus getekend die o.a. langs de in- en uitgangen van 3 registers loopt.

De juiste keuze van in- en uitgang wordt verkregen door selectie van de stuursignalen, aangegeven door de vector (CL[1],CL[2],CL[3]) en van de out-enables, aangegeven door de vector (OE[1],OE[2],OE[3]).

De data (N lang) wordt dus parallel in de bus gelezen en van daar parallel in een register. Er mag altijd maar één bron (source) zijn, dus slechts één OE. Er kunnen echter wel verschillende bestemmingen (destinations) zijn.

Halfgeleidergeheugens

Een halfgeleidergeheugen is een matrix van geheugenelementen, waarbij de rijen een gemeenschappelijke besturing hebben (zoals bij de registers). Door middel van een adresselectie kan één van de rijen geselecteerd worden voor het lezen of schrijven van data per rij. Fig. 27 geeft het blokschema van een halfgeleidergeheugenelement. Het adres komt binnen via de adresbus en wordt in een adres-decoder omgezet naar één geselecteerde rij geheugenelementen. Via de stuurcommando's MEMR (memory read, lees) en MEMW (memory write, schrijf) worden de uitgangen resp. de ingangen van de geselecteerde rij doorverbonden met een bidirectionele DATA-BUS (hierover later meer).

Bij de definitie van lezen en schrijven wordt de besturing van de processor altijd als centrum beschouwd. Lezen uit een geheugen is dus het transport van data uit dat geheugen naar de centrale besturing van de processor. Bij dit lezen blijft de inhoud van de geheugenelementen ongewijzigd: er wordt een kopie gemaakt van het geheugenwoord. Bij het schrijven wordt de data in het geheugenwoord geplaatst, de oude data wordt daarbij overgeschreven.

De toegangstijden tot halfgeleidergeheugens zijn zeer klein ($< 0{,}3\ \mu s$) en gelijk voor alle geheugenwoorden (rijen van de matrix). Vanwege deze gelijke bereikbaarheid voor alle woorden worden deze geheugens RAM genoemd (*R*andom *A*ccess *M*emory). De praktische uitvoering van deze geheugens is altijd met behulp van large-scale integration (LSI) met honderden (of duizenden) geheugenelementen per chip. Enige praktische voorbeelden zijn hieronder opgesomd (anno 1978):

	Adresgrootte (bits)	woordlengte (bits)	aantal woorden
RAM	8	1 of 4 of 8	256
	12	1	4096
RAM (dynamisch)	16	1	65536
ROM	8	1 of 4 of 8	256
	11	8	2048
	16	1	65536

Deze uitvoeringen komen lang niet altijd overeen met het gewenste aantal woorden en de gewenste woordlengte. De woordlengte is eenvoudig samen te stellen uit veelvouden van de woordlengte per chip. De data-bus wordt daarbij dus opgebouwd uit een concentratie van de woorden per chip.

Om het aantal woorden te vergroten, moeten verscheiden chips gebruikt worden, waarvoor er steeds één wordt geselecteerd. Daartoe zijn de chips voorzien van een commando-ingang chip select (CS), die o.a. een enable van de data naar en van de data-bus verzorgt.

In fig. 28 is een geheugen weergegeven van 2K-woorden van 1 byte, dat is opgebouwd uit chips met 256 woorden van 4 bits.

Om de woordlengte van 8 bits te krijgen, zijn altijd 2 chips nodig. Voor 2K-woorden (2048) zijn adressen nodig van 11 bits. Hiervan worden de laagste 8 bits naar alle chips gevoerd (256 adressen). De drie hoogste bits worden 1 uit 8 gedecodeerd en naar de chip-selects van de chips gestuurd. De commando's LEES en SCHRIJF gaan parallel naar alle chips.

Werkgeheugens

De besproken halfgeleidergeheugens zijn zeer snel en willekeurig toegankelijk (random access). Zij zijn dan ook bijzonder geschikt om te dienen als werkgeheugen voor processorsystemen. Onder werkgeheugen wordt verstaan het geheugen van de processor waarin zich het lopende programma en de bijbehorende data bevindt. De grootte van dit werkgeheugen hangt af van de grootte en de toepassing van het systeem.

Voor een microprocessorsysteem, ontworpen voor een bepaalde toepassing, is een werkgeheugen van 2 Kbytes vaak voldoende. Wordt de microprocessor echter ontworpen voor algemene toepassing, dan is soms een werkgeheugen van 64 Kbytes gewenst.

Voor tijdelijk opslaan van tussenresultaten van berekeningen, het bijhouden van tellers, e.d. worden in de meeste processorsystemen hulpregisters gebruikt, ook wel scratch

pad registers genoemd. Door deze registers toe te passen verkrijgt men een aanmerkelijke tijdwinst bij de uitvoering van programma's omdat het mechanisme voor de selectie van de betrokken data dan veel eenvoudiger en sneller kan zijn.

Achtergrondgeheugens

Achtergrondgeheugens zijn doorgaans zeer grote geheugens (schijven, banden, e.d.) waarin programma's en databestanden zijn opgeslagen. Ze vormen dus een soort bibliotheek van data en programma's. In een processorsysteem wordt een achtergrondgeheugen op dezelfde wijze behandeld als de in- en uitvoerapparatuur.

Het onderscheid tussen werkgeheugen en achtergrondgeheugen heeft verschillende oorzaken, namelijk:

a. De prijs/prestatieverhouding en de snelheid van verschillende opslagmedia (verg. fig. 21).
b. De wisbaarheid van de media. (Voor een achtergrondgeheugen is het noodzakelijk dat de informatie bewaard blijft, ook b.v. bij het uitvallen van de voedingsspanningen. Bij halfgeleidergeheugens zou de informatie dan verloren gaan, tenzij er bijzondere maatregelen worden genomen.)

Hoewel het tegenwoordig tot op zekere hoogte mogelijk is om door de mens geschreven informatie op papier direct door een processor te laten lezen, zullen hieronder de achtergrondgeheugens beperkt worden tot media in "computerschrift".

Van deze media zijn, in de huidige stand van de technologie, de magneto-mechanische geheugens het goedkoopst per bit. Ze zijn echter niet bijzonder snel en niet willekeurig toegankelijk.

Magneto-mechanische geheugens

Magneto-mechanische geheugens bestaan uit magnetische oppervlakken die zich ten opzichte van een of meer lees/schrijfkoppen bewegen. Deze oppervlakken zijn op een speciale manier gemagnetiseerd, zodat bij de beweging in de koppen een spanning wordt opgewekt afhankelijk van die magnetisatie.

De diverse bits die door een kop gelezen worden, vormen samen een spoor (track). De verschillende bits van een woord kunnen parallel over verschillende sporen staan of in serie op hetzelfde spoor. De voornaamste magneto-mechanische geheugens zijn hieronder vermeld.

Schijvengeheugens (disks)

Een schijvengeheugen bestaat uit een of meer platte ronde schijven met magnetisch oppervlak (meestal aan beide zijden) waarover lees/schrijfkoppen mechanisch kunnen worden bewogen, zie fig. 29. De schijven van een "disk" roteren gezamenlijk eenparig om een as. Als de koppen stil staan, worden cirkelvormige banen op de oppervlakken gelezen of geschreven. Deze banen noemt men cylinders. Tijdens het lezen/schrijven staan de koppen altijd stil. Door instelling van de koppen kunnen bepaalde concentrische cirkels gelezen of geschreven worden. Om te lezen of te schrijven worden eerst de lees/schrijfkoppen mechanisch boven de juiste cylinder gebracht en vervolgens wordt één lees/schrijfkop elektronisch geselecteerd. De data wordt in serie op het geselecteerde spoor gelezen/geschreven in groepen. Iedere groep begint met een identificatiewoord en synchronisatietekens.

Ook maakt men gebruik van optische of magnetische schakelingen (of hulpsporen) om het begin van de data per spoor te kunnen vaststellen. Het duurt gemiddeld een halve omwenteling van de disk voor het begin van een blok is gevonden.

Het bewegen van de koppen geschiedt mechanisch of hydraulisch. De minimale tijd om de koppen in te stellen op een volgend spoor is daarom ca. 20 ms. De gemiddelde insteltijd ligt tussen 60 en 80 ms.

Enige gegevens ter oriëntatie voor een disk uit het jaar 1970 zijn:

Aantal oppervlakken	20
aantal cylinders	200
aantal bytes/spoor	7200
snelheid	2400 omw/min
capaciteit	28,8 Mbytes

Er zijn tegenwoordig al disks met meer dan 100 Mbytes.

Vooral voor kleine processorsystemen worden momenteel veel zogenaamde floppy disks toegepast. Dit zijn kleine schijvengeheugens met maar één soepele schijf (zoals b.v. bij grammofoonplaten voor reclame wel wordt toegepast). Ter oriëntatie:

aantal oppervlakken	1 (soms 2)
aantal sporen	77
aantal bytes/track	6656
capaciteit	0,5 Mbytes

Trommelgeheugens

Een trommelgeheugen bestaat uit een roterende cylinder met magnetische laag op het cylindervlak. Dicht boven het oppervlak zijn lees/schrijfkoppen aangebracht, zodat iedere kop een beschrijvende cirkel van het oppervlak bestrijkt; deze cirkel wordt een spoor (track) genoemd, zie fig. 30. Er zijn dus evenveel sporen als koppen. De informatie wordt in serie per spoor opgeslagen. Per trommel zijn 2 à 3 hulpsporen aanwezig, waarmee de juiste positie van de trommel door de besturing kan worden bepaald (begin en eind van een spoor, synchronisatie, enz.).

De lees/schrijfkoppen voor een bepaald spoor op de trommel worden elektrisch geselecteerd, waardoor het mogelijk is de data op een trommel met gemiddeld zeer korte toegangstijd te bereiken. Bij 3000 omw/min is dit gemiddeld 10 ms.

Magneetbandgeheugens

Bij een magneetband worden de bits van een teken (woord) loodrecht op de lengterichting van de band naast elkaar op het oppervlak geregistreerd, zie fig. 31. Meestal bestaat zo'n woord uit 9 bits (een byte plus een parity-bit: 9 sporen); er is dus een 9-voudige lees/schrijfkop. De verschillende tekens staan achter elkaar in de lengterichting van de band in "records".

Enige gegevens ter oriëntatie:

lengte	360 – 720 m
bandsnelheid	25 – 500 cm/s
bits/spoor/cm	80 – 250
aantal sporen	7 – 9

Deze records beginnen meestal met identificatie-en synchronisatietekens (pre-amble) en eindigen met enkele speciale tekens (post-amble), waarmee de informatie per spoor geverifiëerd kan worden. Door deze verificatiesymbolen en de paritybit kan gecontroleerd worden of de data van een record al dan niet verminkt is.

Vooral voor microprocessoren wordt ook wel gebruik gemaakt van de bekende standaard bandcassettes (voor audiogebruik). Daarbij is er maar één spoor. De schrijfdichtheid kan tot 32 bits/mm opgevoerd worden. Per cassette kunnen ca. 0,5 Mbytes opgeslagen worden.

Mechanische geheugens

Andere media voor het opslaan van data zijn tegenwoordig nog de ponskaart en de papieren ponsband. Vooral de laatste wordt nog vaak gebruikt voor kleine processor-systemen.

DE REKENKUNDIGE/LOGISCHE EENHEID

De *A*rithmetic *L*ogical *U*nit (ALU) is bestemd om bepaalde rekenkundige of logische bewerkingen op operanden uit te voeren. Een eenvoudig voorbeeld werd al in fig. 3 gegeven. Men kan onderscheid maken tussen een tweetal soorten van bewerkingen, n.l. de monadische, die betrekking hebben op één operand en de dyadische, die betrekking hebben op twee operanden. Het aantal rekenkundige en logische operaties is bij een microprocessor meestal uiterst beperkt. Veel microprocessoren kunnen uitsluitend optellen, andere optellen en aftrekken en evt. vergelijken. Onderstaand een overzicht van de mogelijkheden van de microprocessor 2650.

rekenkundig

dyadisch	monadisch
optellen	complementeren
aftrekken	rotatie linksom
vergelijken	rotatie rechtsom
	(de rotaties kunnen uitgevoerd worden mèt of zònder CARRY-bit)

logisch

dyadisch	monadisch
EN	inverteren
OF	rotatie linksom
Exclusive Or	rotatie rechtsom
vergelijken	(de rotaties kunnen uitgevoerd worden mèt of zònder CARRY-bit)

Naast deze bewerkingen wordt de ALU vaak gebruikt als een eenvoudig transportkanaal voor data die naar of van de microprocessor gevoerd wordt. Dit heeft tot gevolg dat speciale kenmerken, die bij het rekenen behoren, ook gebruikt kunnen worden bij deze databewegingen.

Als gevolg van de bewerkingen in de ALU of bij een transport van data door de ALU kunnen, in een z.g. Conditie Code register (CC), twee bits onafhankelijk van elkaar in de 0- of de 1-stand worden geplaatst, en dit is alleen afhankelijk van het resultaat van de monadische of dyadische bewerking, resp. het transport. Deze twee bits maken deel uit van wat men noemt het Program Status Word (PSW). Op de precieze plaats van deze bits in het PSW wordt nog nader teruggekomen.

De CC zal worden beïnvloed door het resultaat van de bewerking. Bij rekenkundige bewerkingen zal de CC aangeven of het resultaat positief, 0, dan wel negatief is. Het al dan niet negatief zijn wordt beslist aan de hand van het feit dat de microprocessor 2650 geacht wordt in de 2-complement mode te werken. Bij logische operaties wordt het resultaat opgevat als een absoluut binair getal. Bij de behandeling van de afzonderlijke instructies zal telkens de stand die de Conditie Code per instructie aanneemt, nader worden verklaard.

Worden door een rekenkundige bewerking de CC-bits niet in de 1-stand geplaatst, dan komen ze, ongeacht de oude toestand, automatisch in de 0-stand.

De CARRY-bit

Een onderdeel van het PSW is de z.g. CARRY-bit. Deze CARRY-bit wordt in de 1-stand gezet bij een optelling of aftrekking waarvan het resultaat de maat van de 8 bits overschrijdt. Dat kan b.v. voorkomen bij twee negatieve getallen die opgeteld worden. Het resultaat behoeft dan nog niet ongeldig te zijn, maar het duidt aan dat de twee enen op de meest significante plaats volgens de binaire rekening een CARRY produceren, die dan geregistreerd wordt in de CARRY-bit. Zo kunnen twee positieve getallen, beide met een 0 op de meest significante plaats, geen CARRY produceren. Wordt daarentegen een positief getal van een negatief getal afgetrokken, dan zal deze CARRY bit wél in de 1-stand geplaatst worden.

De CARRY-bit kan ook in de 1-stand of de 0-stand gezet worden bij een rotatie naar links of naar rechts van data in een register. Of dit zal gebeuren, hangt mede af van een andere bit in het PSW, n.l. de bit WC. Hierop zal bij de behandeling van deze bit nog nader worden teruggekomen.

De microprocessor 2650 is voorzien van een mechanisme voor het optellen in de BCD-code. Als dit optellen geschiedt, zijn per byte twee decimale digits aanwezig. In de decimale digit kan nu eveneens een CARRY optreden, die de inter-digit-carry (IDC) wordt genoemd. Deze inter-digit-carry speelt ook een rol bij de rotatie van de inhoud van een register. Hierop wordt nader teruggekomen bij de behandeling van de rotatie-instructies.

Vergelijken

De ALU kan eveneens dienst doen voor het vergelijken van twee operanden. Bij een rekenkundige vergelijking wordt in feite het verschil van de beide operanden genomen, maar het resultaat wordt nergens geregistreerd. Alleen de Conditie-Code wordt beïnvloed. De vergelijking tast dus de oorspronkelijke operanden in hun waarde niet aan.

Er is een verschil tussen het vergelijken van getallen en van logische waarden. Als de microprocessor 2650 logische waarden moet vergelijken, dan zal men dit kunnen laten doen door een speciale instructie: vergelijk logisch. Deze instructie is echter niet afzonderlijk aanwezig en men moet de machine van te voren een opdracht geven om een logische vergelijking te maken. Daartoe is een speciale bit in het PSW aanwezig, genaamd COM. Bij de desbetreffende instructies wordt hierop nader teruggekomen.

Een ALU kan een rekenkundige bewerking uitvoeren waarvan het resultaat niet meer past in het bereik van de getallen waar de ALU op dat moment mee werkt. Gaat b.v. de som van twee getallen de waarde van 127 te boven, dan zal er een 1 op de plaats van de meest significante bit verschijnen. De betekenis daarvan is dat er een z.g. "overflow" plaats heeft. De overflow wordt in het PSW geregistreerd door een speciale bit, genaamd OVF. Deze OVF wordt na elke rekenkundige operatie opnieuw in de 1-stand of in de 0-stand gezet, afhankelijk van het resultaat van deze bewerking. Deze OVF-bit is ook betrokken bij de rotatie van de data. Zie hierover de desbetreffende instructies.

OPERATIES EN ADRESSERING (REDUCTIE)

In het onderstaande zal eerst een bewerking tussen twee operanden worden beschreven, waarbij gebruik gemaakt wordt van een zéér groot aantal adressen. Vervolgens zal aan de hand van een aantal beschouwingen dit aantal adressen gereduceerd worden, zodat tenslotte slechts één adres overblijft. Bij deze beschouwingen wordt ook het transport van de data in beschouwing genomen, evenals de tijd die daarmee gepaard gaat. Verkleint men het aantal adressen, dan zal ook het aantal mogelijkheden van de processor verminderen; we zullen nagaan wat hiervan de gevolgen zullen zijn.

Moet een rekenorgaan (ALU) een bewerking of "operatie" verrichten, dan dient hij daarvoor geïnstrueerd te worden en de beide operanden te ontvangen. Het rekenorgaan kan dan het resultaat presenteren. We formuleren dit als volgt:

$$RES \longleftarrow A.B$$

Hierin stelt de . een dyadische operatie voor, zoals + of −. De operanden A en B dienen uit het geheugen betrokken te worden en het resultaat (RES) moet weer op de daarvoor bestemde plaats in het geheugen geregistreerd worden. Dit geheel van gegevens, de operatiecode, de adressen van de beide operanden en het adres van het resultaat is verenigd tot een z.g. instructie. Deze instructie is bij moderne processoren in het geheugen aanwezig, bij microprocessoren in het bijzonder in het Read Only Memory (ROM). Uiteraard

dient de processor over de kennis te beschikken, waar de uit te voeren instructie zich bevindt. Om dit zo eenvoudig mogelijk te houden, kan men elke instructie voorzien van een extra adres dat aangeeft waar de volgende uit te voeren instructie zich bevindt. De instructie kan er aldus uitzien:

<OPCODE> <ADRA> <ADRB> <ADRRES> <ADRVI>

waarin <OPCODE> : operatiecode, een groep bits, die aangeeft welke operatie uitgevoerd moet worden.
<ADRA> : adres van de operand A
<ADRB> : adres van de operand B
<ADRRES> : adres waar het resultaat in het geheugen geplaatst dient te worden.
<ADRVI> : adres van de Volgende Instructie in het geheugen.

Een machine met instructies volgens dit formaat, noemt men een 4-adres machine (zie fig. 32).

De consequenties van dit grote aantal adressen in een instructie laat zich aan de hand van het volgende voorbeeld gemakkelijk aantonen.

Veronderstel dat het geheugen 32K woorden bezit ($K=1024=2^{10}$). Voor het adresseren van ieder woord zijn dan adressen nodig van 15 bits ($2^{15}=32K$). Als de operatiecode 8 bits lang is, dan wordt de totale instructielengte $8 + 4\times15 = 68$ bits. De woordlengte van 8 bits voor een microprocessor staat in schril contrast met de woordlengte voor deze instructie. In grote processor-systemen wordt veelal genoegen genomen met een woordlengte van 32 bits (4 bytes) voor de data. Veel processoren hebben een woordlengte van 8 of 16 bits. Zo heeft de microprocessor 2650 een woordlengte van 8 bits.

Welke maatregelen kan men nu tegen de grote instructielengte ondernemen?

Dit zal later besproken worden. Eerst zullen we nagaan welke handelingen na elkaar (en eventueel parallel) moeten plaats hebben om een instructie uit het geheugen te halen, de operatie te verrichten en het resultaat in het geheugen op te bergen. Stel dat een instructie wordt geregistreerd in een instructieregister (IR) dat uit vijf deelregisters bestaat, genaamd OP, IRA, IRB, IRES en VI voor resp. OPCODE, ADRA, ADRB, ADRRES en ADRVI (zie fig. 33).

Het IR bevat de data van de vorige instructie, dus in VI staat het adres van de nu *komende instructie*. Deze instructie moet uit het geheugen gelezen en geplaatst worden in de registers. Daartoe wordt de inhoud van VI via de ADRBUS (adresbus), zie fig. 33, in het register GEHADR (geheugenadres) geregistreerd en één uit M gedecodeerd (M geheugenplaatsen). De geheugenplaats INST is daardoor geselecteerd, zie fig. 34. Met het leescommando wordt de inhoud van deze geheugenplaats op de DATABUS geplaatst en van daar gebracht in de 5 deelregisters van IR (OP, IRA, IRB, IRES, VI). Deze gang van zaken voor het verkrijgen van de instructie noemt men de instructie-cyclus (I-cyclus).

Nu kan de processor de operatie op de operanden A en B uitvoeren. Daartoe dient de processor eerst de operand A uit het geheugen te betrekken (zie fig. 35); het adres van A (in register IRA) wordt via de ADRBUS naar GEHADR gebracht, waardoor de plaats OPAND A wordt geselecteerd. Na selectie van REGA wordt deze operand nu in REGA geplaatst. Hetzelfde geldt nu voor operand B met het adres in IRB en met de bestemming REGB, zie fig. 36. Nu kan de ALU de opdracht krijgen om de operatie op A en B uit te voeren. Het resultaat komt dan op de DATABUS. Tenslotte wordt de inhoud van IRES naar GEHADR gebracht en geheugenplaats RESULTAAT geselecteerd. Met het schrijfcommando wordt het resultaat, dat zich op de databus bevindt, in de geheugenplaats RESULTAAT geplaatst, zie fig. 37.

Deze stappen, nodig voor het verkrijgen van de operanden en voor het transport van het resultaat vormen samen de executiecyclus (E-cyclus).

Het voorgaande voorbeeld toont aan dat de instructielengte en de woordlengte voor data en adressen bij een 4-adres machine lang niet altijd goed t.o.v. elkaar zijn aangepast. Om een betere geheugenvulling te verkrijgen voor de verschillende soorten data die in het geheugen worden opgeslagen, bestaan een aantal mogelijkheden:

a. De instructie wordt in verschillende opeenvolgende woorden in het geheugen opgeslagen. Deze instructie wordt dan in verschillende stappen uitgelezen, waarbij steeds de inhoud van VI met één verhoogd moet worden.
b. De instructielengte wordt verkleind.
c. Een combinatie van a en b.

Beide maatregelen worden, in combinatie, vooral toegepast in machines met een kleine woordlengte (b.v. 1 byte). In machines met een grote woordlengte zal men vooral trachten de instructielengte te verminderen.

Het is van belang dat de volgorde waarin de instructies uitgevoerd moeten worden, vast ligt (ordening). Dat blijkt uit het feit dat men het adres van de volgende instructie in de voorafgaande vermeldt. Men kan deze ordening ook zo ver doorvoeren dat de elkaar opvolgende instructies aanwezig zijn op opeenvolgende geheugenadressen. Dit heeft dan tot gevolg dat de volgende instructie altijd even veel plaatsen verder in het geheugen staat als de lengte van de instructie in geheugenwoorden bedraagt. De plaats van de volgende instructie kan dus gevonden worden door bij het adres in het VI-register (voorbeeld van de 4-adres machine) de instructielengte (in woorden) op te tellen. Het VI-register wordt daardoor een zelfstandig register, uitgevoerd als teller, waarvan de inhoud niet meer in iedere instructie vermeld behoeft te worden, zie fig. 38. Dit zelfstandige register wordt wel instructieteller (IT), programcounter (PC) of Instructie Adres Register (IAR) genoemd.

Er zijn gevallen waarin de instructies elkaar niet opvolgen, zoals bij sprongopdrachten. Dat probleem zal later besproken worden.

De instructielengte in het voorbeeld van de 4-adres machine is nu door het ontbreken van VI met 15 bits verminderd tot 53 bits. Het instructieformaat is nu (3-adres machine):

<OPCODE> <ADRA> <ADRB> <ADRRES>

Om de instructielengte nog verder te reduceren, kan men materiaal van het geheugen uitwisselen tegen meer tijd voor het uitvoeren van de instructie en voor deze serie handelingen meer dan één instructie gebruiken.

Een ander middel is het aanbrengen van een extra register aan de uitgang van de ALU. Men kan daarin het resultaat van een operatie tijdelijk opbergen. Daarop moet een extra transportinstructie (monadisch) volgen, waarmee dit resultaat naar de juiste plaats in het geheugen wordt overgebracht. Het adres van het resultaat hoeft dan niet meer expliciet in (rekenkundige/logische) instructies vermeld te worden.

In het eerder gegeven voorbeeld wordt de instructielengte nu 38 bits. Het instructieformaat is aldus (vgl. ook fig. 38):

< OPCODE > < ADRA > < ADRB >

Er zijn nu twee extra registers aanwezig, namelijk de IT en het register voor het tijdelijk registreren van het resultaat. Het gebeuren is verminderd van 68 tot 38 bits per woord, zonder iets te verliezen! Er is echter een zekere transporttijd bijgekomen om het tussenresultaat te registreren.

Dit is een duidelijk voorbeeld van de uitwisselbaarheid van materiaalkosten tegen procestijd. De toeneming van de procestijd is echter vaak geringer dan op het eerste gezicht verwacht wordt.

Beschouw b.v. de berekening van een tweedegraadsfunctie:

$$Y = (AX + B)X + C = AX^2 + BX + C$$

Deze functie kan berekend worden met de volgende eenvoudige operaties (RES is het tijdelijke register):

RES ⟵ A
RES ⟵ RESxX
RES ⟵ RES+B
RES ⟵ RESxX
RES ⟵ RES+C
Y ⟵ RES

Voor RES kan b.v. het extra register ACC aan de uitgang van de ALU gebruikt worden, zie fig. 38. In dit register worden de tussenresultaten van de berekening steeds "geaccumuleerd". Een dergelijk register wordt daarom vaak accumulator (ACC) genoemd. In de microprocessor 2650 wordt deze accumulator Reg. 0 genoemd.

Er was telkens één tussenresultaat dat bij de volgende operatie gebruikt werd als operand. Als plaats voor het (tussen)resultaat kan ook het adres van één van beide operanden genomen worden (hier de operand Y), in plaats van de accumulator. Dan is echter bij iedere deeloperatie weer een toegang tot het geheugen nodig. De toegangstijd tot een geheugenplaats is evenwel in het algemeen een factor 10 langer dan de toegangstijd tot een register.

Als echter één van de operanden vóór het uitvoeren van de operatie van een vast geheugenadres (b.v. MEM[O]) of van een register (b.v. de ACC) wordt betrokken, dan hoeft het adres van die operand niet meer expliciet in de instructie vermeld te worden. Zo ontstaat een 1-adr. machine met instructieformaat (vgl. ook fig. 38):

< OPCODE > < ADRA >

De instructielengte, bij het gegeven voorbeeld, is nu nog slechts 23 bits.

De berekening van de tweedegraadsfunctie als boven wordt nu, in stappen onderverdeeld:

ACC ⟵ A
ACC ⟵ ACCxX
ACC ⟵ ACC+B
ACC ⟵ ACCxX
ACC ⟵ ACC+C
Y ⟵ ACC

Door de verkleining van de instructielengte, zoals hierboven beschreven, zijn enkele extra instructies toegevoegd, n.l. één waarmee de data uit het geheugen wordt gehaald (ACC←A) en één waarmee data naar het geheugen (Y←ACC) wordt gebracht. De operatiecodes van deze instructies zijn b.v. LOAD en STORE en het formaat is dan:

LOAD <ADRES>
STORE <ADRES>

De hier besproken machineconfiguratie bezit nog enkele essentiële tekortkomingen. Eén daarvan is dat, ondanks het vermogen van de machine te kunnen rekenen, het niet mogelijk is om het resultaat van een berekening naar het GEHADR-register in het geheugen te sturen. Een andere onvolkomenheid is dat dit register in het geheugen aanwezig is. In het algemeen veroorzaakt dit een zekere tijdvertraging. Dit register zou heel goed in de eigenlijke processoreenheid geplaatst kunnen worden, waarna men met de adresbus verder kan gaan naar het geheugen (zie fig. 39). Door de verbinding met de databus kan het resultaat van een berekening nu rechtstreeks uit de ALU in GEHADR geplaatst worden. Ook de instructieteller (IT) kan nu zijn data via de databus aan dit register aanbieden. In de processor zelf is nu maar één bus aanwezig, en er zijn twee bussen voor de communicatie met de buitenwereld, n.l. de adresbus en de databus. Doordat de processor intern slechts de beschikking heeft over één bus, zal deze trager werken.

Het ligt voor de hand om een aantal nieuwe interne bussen te introduceren, vooral als er meer dan één register aanwezig is. Zo kunnen twee bussen toegevoegd worden: één voor de A-ingang van de ALU en één voor de B-ingang (zie fig. 40). Deze bussen hebben uitsluitend zin als de registers gelijktijdig en onafhankelijk van elkaar data op de bussen kunnen plaatsen. De uitgang van de ALU kan nu leiden naar een aantal registers en deze kunnen dan data in ontvangst nemen en dienen als accumulator. Nu treedt het probleem op, hoe men data uit een register kan presenteren aan de buitenwereld, m.a.w. aan de externe databus. Daartoe kan de data via de ALU aan een databuffer worden aangeboden (zie fig. 40). Hetzelfde geldt voor het adres dat aan een adresregister gepresenteerd kan worden, eveneens via de ALU. Het is duidelijk dat de communicatie met de buitenwereld in dit geval altijd via de ALU zal verlopen, hetgeen betekent dat deze communicatie een zekere vertraging zal ondervinden. Het aanbrengen van een speciale adres-bus, die eveneens van de registers uit bereikbaar zou moeten kunnen zijn, zou hier een oplossing kunnen bieden. Deze methodiek wordt meestal niet gekozen omdat het vaak voorkomt dat een adres, zoals dit in een register aanwezig is, nog een bewerking moet ondergaan. Deze bewerking heeft plaats in de ALU, en zodoende kan het gewenst zijn, de ALU in het datapad voor de adressen te handhaven.

De processor communiceert niet uitsluitend met het geheugen. Het geheugen dient vooraf gevuld te worden met informatie, en deze informatie wordt uit de z.g. periferie-apparaten verkregen. Het mechanisme om deze periferie-apparaten te bereiken, is vooral bij kleine processen vrijwel identiek aan dat om het geheugen te bereiken. De adres-bus wordt dan ook dikwijls gebruikt voor het adresseren van periferie-apparaten en deze apparaten plaatsen de data op de externe databus. Teneinde dit in de juiste synchronisatie te kunnen doen, is nog een derde bus, de z.g. controlebus, aanwezig, zie ook fig. 39.

MACHINE-NIVEAU-TALEN

Digitale processoren kunnen tot op heden nog geen andere symbolen verwerken dan die van het binaire talstelsel, namelijk nullen en enen. Elke vorm van opdrachten en data moet daarom als een rij nullen en enen worden weergegeven (een z.g. vector met vectorelementen 0 en 1). Afhankelijk van de plaats waar een vector zich in het processorsysteem bevindt, kan deze op verschillende manieren geïnterpreteerd worden. Een vector in de instructieteller zal opgevat worden als een absoluut binair getal, dat het adres van de instructie voorstelt. Een vector in het instructieregister zal voor een deel opgevat worden als een binair codewoord (de op. code), en voor de rest als adressen en data (zie vorig hoofdstuk). De interpretatie is dus fundamenteel verschillend.

Het is vrijwel ondoenlijk om grote programma's te schrijven in deze binaire code, ook wel machinetaal genaamd. De kans op het maken van fouten is zeer groot. De codes zijn afhankelijk van het type processor, en de instructies in deze vorm zijn uiterst onoverzichtelijk. Het laten nalezen van programma's en het opsporen van fouten is bijzonder moeilijk en tijdrovend.

De machinegebruiker wil het liefst een taal gebruiken die zo dicht mogelijk aansluit bij de taal die hij zelf gebruikt om te communiceren, om wiskundige formules te noteren, om opdrachten te geven, enz. Deze hogere talen kan de processor tot op heden echter niet rechtstreeks gebruiken om de instructies uit te voeren.

In het algemeen bevatten de expressies in een hogere taal te veel variabelen en is hun structuur te ingewikkeld voor rechtstreekse analyse in microprocessoren. Men heeft voor iedere machine echter altijd een eenvoudige programmeertaal ontworpen die dicht bij de machinetaal staat. Een vertaalprogramma voor zo'n eenvoudige programmeertaal wordt meestal assembler genoemd (of cross-assembler als het vertalen op een andere machine geschiedt). Men noemt de taal (niet geheel correct) de assemblertaal.

De eenvoudigste vorm van een assemblertaal is de één-op-één vertaling van eenvoudige mnemonic-taal in machinetaal. Een mnemonic is een door de mens te begrijpen code, bestaande uit een combinatie van letters (en evt. cijfers of andere tekens) voor de instructie in machinecode. Dit houdt in dat bij elke instructie één mnemonic behoort en omgekeerd. Zo zou b.v. ADD 23 (in assemblertaal: tel de inhoud van geheugenplaats 23 op bij de inhoud van de accumulator) in machinetaal kunnen worden (na vertaling door de assembler):

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

code: tel op | binair adres (23)

Als uitbreiding hierop kent men de mogelijkheid om adressen symbolisch aan te geven en geheugenruimte te reserveren, b.v. namen voor "labels". Is deze mogelijkheid er niet, dan moet de programmeur zelf de adressen berekenen. Ook moet hij dan zelf nagaan, waar in het geheugen ruimte is voor het opslaan van informatie. Hij moet dus, uitgaande van het adres van de eerste instructie van het programma, de adressen van de volgende instructies berekenen. De adressen van sprongen vooruit in het programma zijn dan echter nog niet bekend en moeten later ingevuld worden. Door de instructies en geheugenruimten van zulk een label (d.w.z. symbolische naam) te voorzien, is dit mogelijk.

Helaas heeft iedere processor zijn eigen verzameling instructies, de instructieset genaamd. Daarom heeft iedere processor ook zijn eigen assemblertaal. We moeten ons hier beperken tot het geven van een globaal overzicht van mogelijkheden in assemblertalen. De instructies van de instructieset kunnen in verschillende categorieën onderverdeeld worden, en wel in:

a. Rekenkundige en logische instructies (monadische en dyadische). Dit soort werd al vermeld bij de behandeling van de rekenkundige en logische eenheid (ALU).
b. Transportinstructies voor het transporteren van data tussen geheugenstructuren (zoals werkgeheugen, registers, e.d.)
c. Input/output-instructies voor het transport van data van en naar de periferie-apparatuur.
d. Sprong-instructies, al dan niet conditioneel (afhankelijk van de waarden van de conditiecodes, in het laatste geval altijd springen) voor het veranderen van de volgorde waarin de instructies worden uitgevoerd.
e. Aanroep- en terugkeer-instructies, al dan niet conditioneel, voor het aanroepen van (sub)programma's en de terugkeer naar het hoofdprogramma.
f. Programma-onderbrekingsinstructies voor het onderbreken van de normale verloop van een programma (b.v. bij een z.g. interrupt) en voor het stoppen van de uitvoering van een programma.

HET PROGRAM STATUS WORD

Het Program Status Word (PSW), zie fig 41, is een uitermate belangrijke verzameling van flip-flops ter registratie van de z.g. FLAGS. Het PSW bevat in het totaal 16 van deze flip-flops. Daar de gehele organisatie van de microprocessor 2650 gebaseerd is op een datapad ter breedte van 8 bits, ligt het voor de hand dit PSW op te delen in twee delen, het z.g. PSW UPPER, afgekort PSU en het PSW LOWER, afgekort PSL. Indien men m.b.v. instructies de inhoud van het PSW wenst te wijzigen, dan zal men deze veranderingen dus moeten richten op beide groepen van 8 bits, en wel de een na de ander. De instructies hiervoor zijn vrijwel identiek. Ze verschillen uitsluitend door de aanduiding L of U.

Van het PSU zijn de bits 0, 1 en 2 gereserveerd voor een stapelmechanismeteller die het mogelijk maakt, de inhoud van het Instructie Adres Register (IAR) (of van de Program Counter (PC)) op de top van de adresstapel te plaatsen. Heeft men dat gedaan, dan kan de inhoud van het IAR gewijzigd worden, terwijl de oorspronkelijke inhoud toch nog aanwezig is op de gecreëerde stapel.

De bits 3 en 4 van het PSU zijn niet in gebruik.

De bit 5 wordt gebruikt om externe interrupts onmogelijk/mogelijk te maken. Het heet de Interrupt Inhibit bit (II). Als deze bit = 1, dan zal de buitenwereld de microprocessor 2650 niet kunnen interrumperen.

Bit 6 is rechtstreeks verbonden met pen 40 van de microprocessor 2650. Deze flip-flop wordt in de literatuur over de microprocessor 2650 de FLAG genoemd en kan gebruikt worden als één bits output.

Bit 7 is eigenlijk geen flip-flop. Het is een "dummy", die verbonden is met pen 1 van de microprocessor 2650 en kan evenals de andere flip-flops onderzocht worden op zijn waarde 0 of 1. Dit betekent dat men m.b.v. een instructie die betrekking heeft op het PSU, in wezen de toestand van pen 1 kan onderzoeken. Dit is dus een één-bits input.

Het PSL bevat eveneens 8 bits.

Bit 0 heeft betrekking op de CARRY, d.w.z. deze bit wordt automatisch gezet als b.v. bij het sommeren van twee negatieve getallen een 1 wordt gegenereerd die niet meer in het rekenorgaan past. Wordt deze 1 niet gegenereerd, dan wordt de CARRY, ook al staat hij in de 1, automatisch op

0 gezet. Niet-rekenkundige instructies (behalve rotatie) beïnvloeden het carry-bit niet. Na deze instructies is de carry ongewijzigd.

Bit 1 van het PSL (de COM-bit) heeft betrekking op de wijze waarop twee operanden vergeleken worden. Het vergelijken van twee rekenkundige getallen geschiedt anders dan het vergelijken van twee logische vectoren. Wegens de plaats van de tweede operand (zie instructies), zijn er vier vergelijkings-instructies noodzakelijk. Moeten deze instructies zowel voor de logische als voor de rekenkundige getallen dienst doen, dan zou er in het totaal aan acht verschillende vergelijkings-instructies behoefte zijn. Daar vergelijkingen meestal worden uitgevoerd tijdens bepaalde groepsbewerkingen, ligt het voor de hand dat men alvorens een dergelijke groepsbewerking te starten, met behulp van één instructie aan de ALU aangeeft of deze een rekenkundige, dan wel een logische vergelijking dient uit te voeren. Om dit te registreren, is bit 1 van het PSL aanwezig.

Bit 2 van het PSL is de z.g. overflow-bit (OVF), die aangeeft of de rekenkundige capaciteit van de microprocessor 2650 niet wordt overschreden. Dit zou kunnen geschieden als men twee positieve òf twee negatieve getallen bij elkaar optelt, of als men b.v. een negatief getal van een positief getal aftrekt, resp. een positief getal van een negatief getal aftrekt. Overschrijdt de absolute waarde een bepaalde grootte, dan zal de microprocessor dit kenbaar maken door het in de 1 zetten van de OVF-bit.

Bit 3 van het PSL is de z.g. WITH CARRY-bit (WC). Deze bit geeft aan of de CARRY flip-flop bij bepaalde rekenkundige functies een rol zal spelen of niet. Bovendien geeft deze flip-flop aan of, bij het roteren van de data in een register, de data al dan niet via de CARRY-flip-flop, d.w.z. PSL bit 0 wordt gevoerd.

Bit 4 van het PSL is de z.g. Register Select-bit (RS). Op het doel van deze bit wordt later bij de behandeling van de adresseermethodieken teruggekomen. Hij deelt de zes registers (R0 wordt uitgezonderd) in twee groepen in, de z.g. banks, d.w.z. BANK 0 en BANK 1, elk met registers 1, 2 en 3.

Bit 5 van het PSL is de interdigit-carry (IDC) en doet dienst bij het rekenen in de BCD-code, als twee decimale cijfers per byte aanwezig zijn. In feite verricht deze IDC een soort CARRY-functie, nu echter op de grens tussen de bits 3 en 4 van de byte.

Tenslotte geven de bits 6 en 7 van het PSL de Conditie Code (CC), die telkens optreedt na afloop van een rekenkundige of logische operatie. Bij een rekenkundige operatie geven deze CC-bits aan of het resultaat positief, negatief, dan wel 0 was. Op de exacte waarde van deze Conditie Code wordt bij de instructies teruggekomen.

ADRESSERING

Adresseermethoden

Praktisch alle tot nu toe gebruikte werkgeheugens voor digitale processoren zijn lineair adresseerbaar, d.w.z. dat iedere geheugenplaats bereikbaar is m.b.v. een adres. Zo'n adres heeft een bepaalde waarde, die de absolute plaats aangeeft t.o.v. het nulpunt, d.w.z. t.o.v. de eerste byte in het geheugen. Hierbij is aangenomen dat het geheugen een breedte heeft van 8 bits, ofwel één byte.

De inhoud van een geheugenplaats is een binair woord. Wat dit woord voorstelt (getal, letter, code, instructie, enz.) is voor het geheugen niet van belang, maar wordt pas bepaald als het woord in de processor gebruikt wordt. Een deel van het geheugen wordt gebruikt om het programma of de programma's op te slaan. Dit geheugen is juist bij microprocessoren vaak uitgevoerd als een Read Only Memory (ROM). Welk woord voor het verkrijgen van een instructie moet worden uitgelezen, wordt bepaald door de instructieteller (ofwel Program Counter (PC)) of, zoals deze bij microprocessor 2650 wordt genoemd, het Instructie Adres Register (IAR). Dit register bevat het adres van de *eerstvolgende* instructie die uitgevoerd dient te worden, nadat de in gebruik zijnde instructie is voltooid (zie fig. 42). Een ander deel van het geheugen wordt gebruikt voor de data die nodig is voor de programma's.

Soms is één geheugenplaats voldoende voor het opslaan van deze data, b.v. voor een constante waarvan de absolute waarde kleiner is dan 256, een teken dat straks uitgeprint moet worden, een indicatie voor de marge bij een regeldrukker, enz.

In veel gevallen bestaat de data echter uit vectoren of matrices van gegevens. Zo zal een tekst, die via een processor afgedrukt moet worden, als een vector in het geheugen staan. Ook een serie meetwaarden kan als een vector wor-

den beschouwd; de elementen voor deze vector zijn de afzonderlijke meetwaarden. Deze meetwaarden staan achter elkaar, d.w.z. op opeenvolgende plaatsen in het geheugen; ze zijn geordend (zie fig. 43). Op de eerste plaats staat de eerste meetwaarde, hier aangegeven met MEET[0]. De plaats daarop bevat de meetwaarde MEET[1], enz. t/m de n-de meetwaarde, aangegeven door MEET[n-1]. Men noemt de waarde tussen de haken de indices, en alle elementen van de meetwaarden staan bekend onder de vectornaam MEET. Het gewenste element van de verzameling wordt aangegeven door de index. Een gebruiker zal een programma het liefst zodanig samenstellen dat hij een bepaald programmadeel voor verschillende vector- of matrixberekeningen kan gebruiken. Dit bespaart niet alleen werk, maar bovendien ook ruimte in het geheugen. Het brengt echter met zich mee dat de adressen van de instructies, voor het geval dat van een matrix gebruik gemaakt wordt, variabel dienen te zijn. Voor het verwerken van de bovengenoemde vector MEET zou dus het eerste adres MEET moeten zijn, het tweede MEET+1, het derde MEET+2, enz. Dit zou b.v. kunnen als men een instructie zou kunnen lezen, rekenkundig zou kunnen bewerken en daarna weer terugzetten op dezelfde plaats. Bij microprocessoren levert dit echter onoverkomelijke bezwaren op omdat de instructies vaak in een Read Only Memory aanwezig zijn. Het zal ook grote bezwaren opleveren bij z.g. Random Access Memories (RAM's), daar in dat geval de plaats van de data en de instructies onderling zeer nauw verbonden zijn, hetgeen tot ongewenste complicaties aanleiding geeft. Deze methodiek moet dus sterk ontraden worden. Een betere methode van indexeren is het gebruik maken van een register, waarbij de verplaatsing t.o.v. de basis van de waarde MEET gegeven wordt. Nu kan men in de instructie volstaan met de plaats van MEET weer te geven. Bij het lezen van de instructie wordt de inhoud van dit (soms speciale) register, het indexregister, bij de waarde van het adres opgeteld, en op deze wijze wordt de plaats van het vectorelement gevonden (zie fig. 44).

In de beschreven methode kan men dus een geheugencel adresseren door het adres in de instructie zelf te vermelden, resp. door de basis te vermelden van waar men een bepaald element van een vector kan vinden. De beide adresseermethoden bevatten een absoluut adres, n.l. het adres van de MEET[0]. Deze methode van adressering wordt **absolute adressering** genoemd.

Als een geheugen, zoals bij de 2650 het geval is, maximaal 32K bytes groot is, dan heeft men voor een directe adressering 15 bits nodig. Telt men daarbij op het aantal bits voor de operatiecode en de bits voor het nummer van het betreffende indexregister, dan blijkt al gauw dat 24 bits onvoldoende blijkt te zijn voor een instructie. Dit betekent dat een instructie vier bytes in beslag zal gaan nemen. Stelt men echter bij het gebruik van programma's statistisch vast hoe groot een programma is, en hoe vaak buiten het eigenlijke programmadeel toegang tot ver verwijderde delen (dit in rekenkundige zin) van het geheugen toegang wordt verlangd, dan blijkt dat veel programma's (inclusief hun data) géén grotere gebieden dan 2K bytes nodig hebben. Bij een groot percentage van de programma's kan zelfs 128 bytes voldoende zijn. Met dit als uitgangspunt, kan men nu aan een adresseermethode denken die een wezenlijk geringer aantal bits nodig heeft voor het bereiken van een gegeven adres dat in de onmiddellijke nabijheid van de instructie ligt. Voorbeelden hiervan zijn bepaalde numerieke gegevens, bepaalde logische waarden, andere instructies, die b.v. in een lus doorlopen worden en waarbij de lus niet groter is dan een gegeven aantal bytes, enz.

Men kan geen al te grote sprongen in het programma maken t.o.v. het adres van een instructie. De vraag rijst nu, welk adres wenst men daarvoor precies te kiezen? Welnu, als een instructie wordt uitgevoerd, dan bevat het Instructie Adres Register (IAR) het adres van de *eerstvolgende* instructie die uitgevoerd zal gaan worden. Men beschikt dus al over een indicatie waaruit blijkt waar men zich in een programma bevindt. De afstand die men nu gaat bepalen, zal relatief t.o.v. deze plaats zijn, waarvan het adres zich in dit Instructie Adres Register bevindt. Vergelijkt men dit met de indexering, dan komt de inhoud van het IAR eigenlijk overeen met de beginwaarde van een bepaald gebied en de inhoud van het indexregister met de verplaatsing t.o.v. dit punt. Dit geldt dus ook voor het IAR. De inhoud van het IAR is het vertrekpunt ten opzichte waarvan men een bepaalde afstand kiest om in een andere geheugencel te komen. In de microprocessor 2650 varieert deze waarde van –64 tot +63; de waarde 0 is hiervan niet uitgesloten. Deze wijze van adressering noemt men **relatieve adressering** (zie fig. 45).

Deze adresseermethode heeft nog een ander groot voordeel bij het schrijven van programma's. Heeft men, om een of andere reden, een extra instructie nodig in het programma, dan zal het gehele volgende deel van het programma verplaatst worden, inclusief de desbetreffende instructie ten opzichte waarvan men straks de verplaatsing zal berekenen. Als de extra ingevoerde instructie niet binnen het gebied van de verplaatsing ligt, dan verandert er helemaal niets aan de afstand tussen het punt van vertrek en het punt van aankomst. Men behoeft dus de relatieve verplaatsingen niet te veranderen, hetgeen werk bespaart en minder aanleiding tot fouten geeft. Een voorbeeld treft men aan in fig. 45, die dit toegelicht. In het gearceerde deel wordt een bepaalde instructie uitgevoerd, die terug verwijst naar een vorige instructie. Dit is b.v. het geval bij het doorlopen van een lus. Stel dat deze vorige instructie op een afstand r ligt t.o.v. de eerstvolgende instructie die zou zijn uitgevoerd als dit terugvallen in de lus niet zou hebben plaats gehad. In het IAR is de plaats van deze instructie aanwezig, en wel H'0089'. Stel nu dat de relatieve verplaatsing negatief-H'15' is, hetgeen neerkomt op 21 bytes. De instructie die als eerstvolgende uitgevoerd zal worden, ligt nu op het adres H'0089' – H'15' = H'0074' (– H'15' = B' 01101011' = H'6B'). Zoals vermeld, beperkt deze afstand bij de microrpocessor 2650 zich van

–64 t/m +63. De adresberekening voor de relatieve adressering is m.b.v. de ALU eenvoudig uit te voeren. Een voordeel van deze relatieve adressering is dat voor het adresgedeelte slechts 7 bits zijn vereist (in de 2-complement code). De achtste bit van een byte wordt voor de z.g. indirecte adressering gebruikt, die later verklaard zal worden.

De relatieve adressering geeft dus een grote adresseermogelijkheid om vooral bij korte programmadelen bijzonder efficiënt te adresseren. Het gebruik van korte programmadelen is in het algemeen toch sterk aan te bevelen, daar men het programmadeel dan immers op een eenvoudige wijze kan onderzoeken op zijn fouten. Een groot programma kan men samenstellen op basis van veel kleine programmadelen.

Het is ook mogelijk **indirecte adressering** toe te passen. Ondanks het feit dat men op deze wijze het aantal adresbits weet te beperken, kan het toch noodzakelijk zijn om elders in het geheugen bytes te adresseren. Het behoeft immers niet noodzakelijk te zijn dat alle data of alle lussen zich in de gebieden van 128, resp. 8K bytes bevinden. De relatieve adresseermethode en die met pagina's zijn niet toereikend om het 32K geheugen te adresseren. Men lost dit probleem op d.m.v. zogenaamde indirecte adressering.

In de desbetreffende instructie is het volledige adres niet aanwezig. Er wordt echter 1 bit gebruikt om aan te geven dat het adres niet het adres zelf is, maar een geheugencel die vermeldt waar zich het werkelijke adres van de betreffende data bevindt. Men verwijst dus naar de inhoud van een geheugengebied in de onmiddellijke nabijheid, waar men zich kan oriënteren over de werkelijke verblijfplaats van de desbetreffende data. De adressering wordt als het ware in een tweetal stappen uitgevoerd. Het adres dat zich binnen het betreffende geheugendeel bevindt (relatief, pagina), is als het ware een voetsteun om zich naar een groter gebied te kunnen verplaatsen (zie fig. 47). Bij deze methode wordt het adres in het adresregister voor de operand vervangen door de inhoud van de geheugencel, die hierdoor wordt aangegeven.

Bij een woordlengte van 8 bits zal één woord niet voldoende zijn voor het adresseren. Twee opeenvolgende woorden (2 bytes dus) bevatten in totaal 16 bits, hetgeen wel voldoende is om in een geheugen van 32K volledig te adresseren. Het operand-adres in de instructie verwijst dus naar een koppel van 2 bytes. Daar men echter slechts één byte kan adresseren, is de geadresseerde byte de meest significante byte van het nieuwe adres. De tweede byte, die daarop volgt, is dan de minst significante byte van het nieuwe adres. Dit nieuwe adres heet het effectieve adres.

Indirecte adressen kan men ook heel goed gebruiken binnen een pagina. Als men namelijk een bepaalde operand diverse keren adresseert, d.w.z. dat hetzelfde adres in meer dan een instructie voorkomt, dan is het qua plaatsruimte voordeliger om één keer een indirect adres aan te brengen, en naar dit indirecte adres te verwijzen. Dit geldt in het bijzonder voor de relatieve adressering, waarbij het adresdeel slechts 7 bits in beslag neemt. Als men binnen een pagina adresseert, dan kan dit ook voordeliger zijn, daar men dan toch met verkorte adressen werkt en er meer faciliteiten, zoals indexering e.d., aanwezig zijn.

Tenslotte past men ook wel **immediate adressering** toe. Men kent aan het geheugen van een rekenmachine een bepaalde hiërarchie toe. Het meest voorkomende deel van zo'n geheugen dat geraadpleegd wordt of betrokken is bij operaties, is de accumulator of, zoals bij de microprocessor 2650, register R0. Vervolgens zijn er een aantal zeer snel toegankelijke geheugens, die we registers zullen noemen. Daarnaast is er een groter geheugen buiten de eigenlijke processor, dat 32K bytes kan bevatten. Diverse operaties worden uitgevoerd waarin, behalve de accumulator of een van de registers, ook een byte uit het geheugen betrokken is. Deze byte kan een constante waarde hebben of zeer specifiek bij de betreffende operatie behoren, zoals b.v. het optellen van het getal 1 of een dergelijke kleine waarde, resp. een vector. Dit speelt vooral een rol bij de z.g. constanten, maskers en dergelijke. Men kan een dergelijke byte in een geheugen onderbrengen, maar dan dient men deze wel te adresseren. Dit adresseren vergt over het algemeen meer tijd dan de betreffende byte direct in de instructie zelf onder te brengen. De data is dan onverbrekelijk met de instructie verbonden, en vormt er een integrerend bestanddeel van. Deze wijze van adressering noemt men immediate addressing.

Paginering

Bij het verrichten van rekenkundige en logische operaties tussen twee operanden zal veelal een van de operanden zich bevinden in de accumulator of in een register. Verder kan er een register gebruikt worden voor het registreren van een index. Om dit aan te geven, zijn er een aantal bits in de instructie vereist. Hierdoor blijven er onvoldoende bits over om wederom een volledig geheugen te adresseren. Bij de microprocessor 2650 zijn er voldoende bits voor het adresseren van een gebied van 8K bytes. Er zijn vier van deze gebieden aan te geven, en men noemt deze pagina's, zie fig. 46. De paginanummers zijn 0, 1, 2 en 3. Ook hier kan men, op grond van de distributie, zoals vermeld, voor bijna 96% van de handelingen binnen dit gebied werkzaam zijn. Men kan dus de 2 meest significante bits voor de adressering van een 32K geheugen weglaten. De informatie die daardoor verloren gaat, ontleent men nu aan de 2 meest significante bits van het IAR. Als men aldus op deze verkorte wijze adresseert, dan adresseert men automatisch slechts in die pagina waarin zich ook de instructie (met het verkorte adres) bevindt.

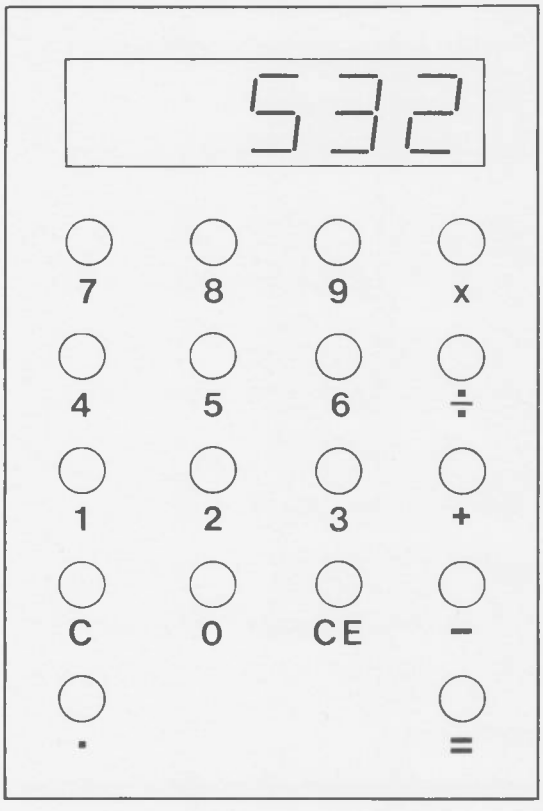


Fig. 1 De zakrekenmachine.

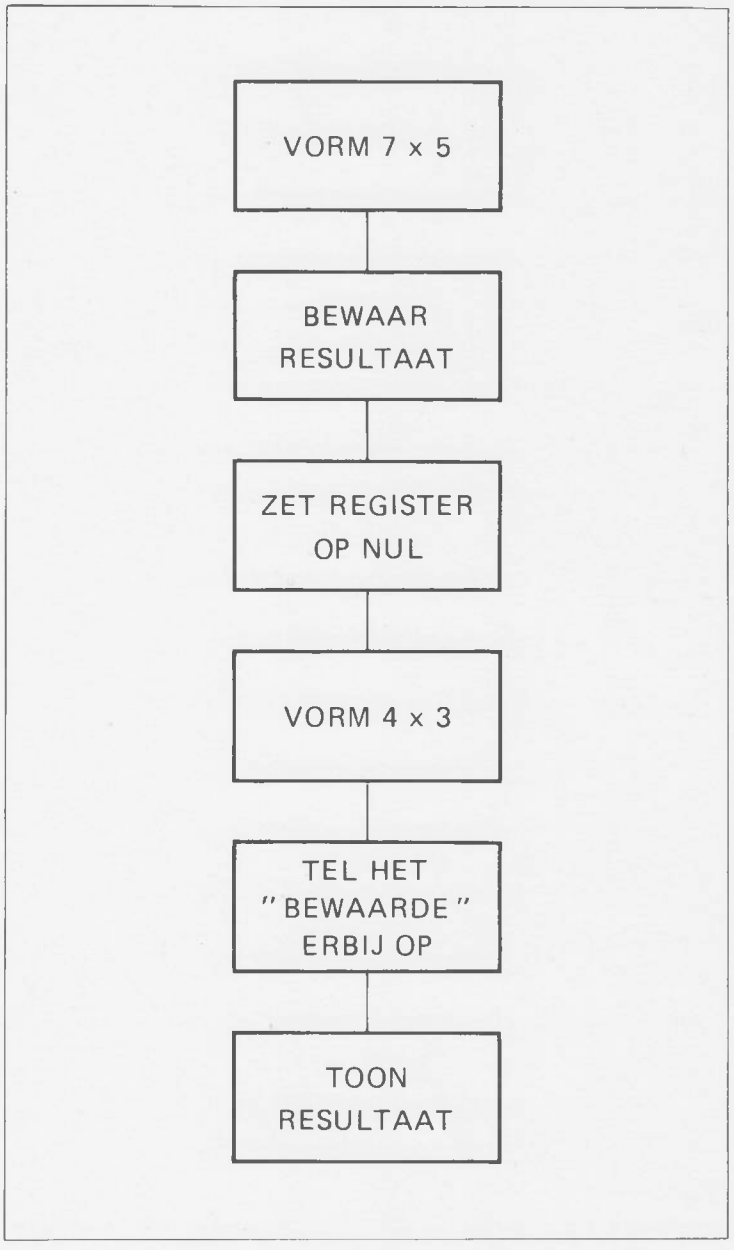


Fig. 2 Informatiestroom bij vermenigvuldigen van getallen.

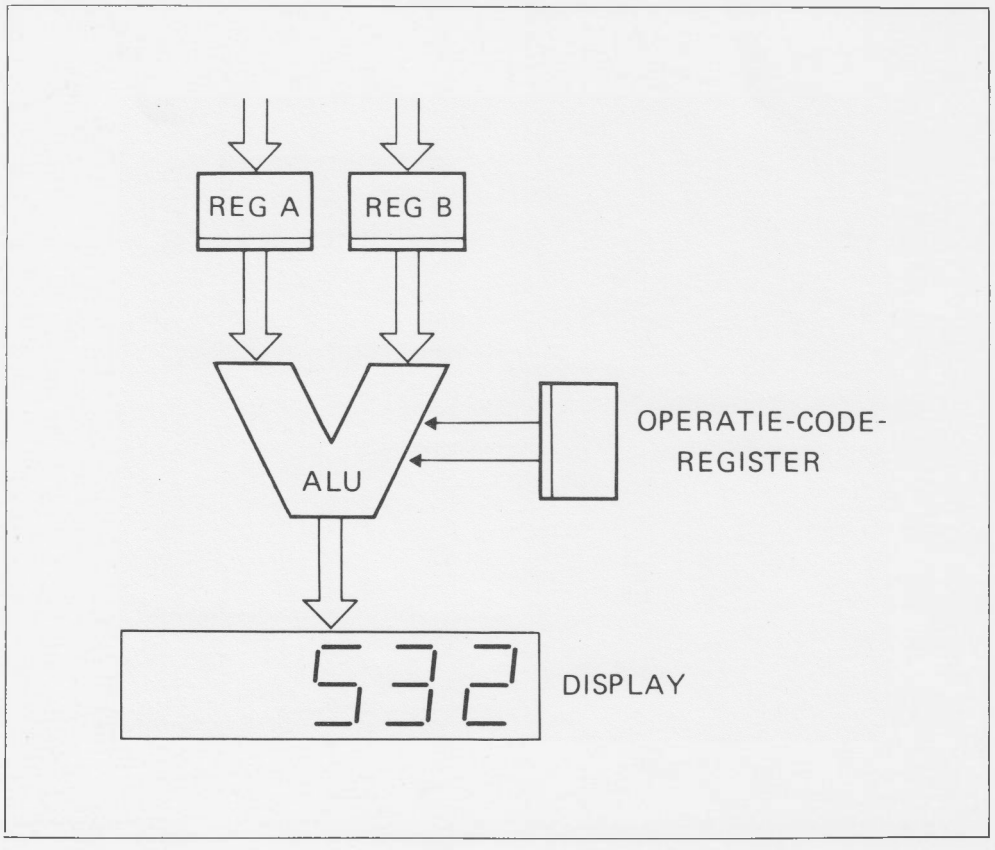


Fig. 3 De ALU van de zakrekenmachine.

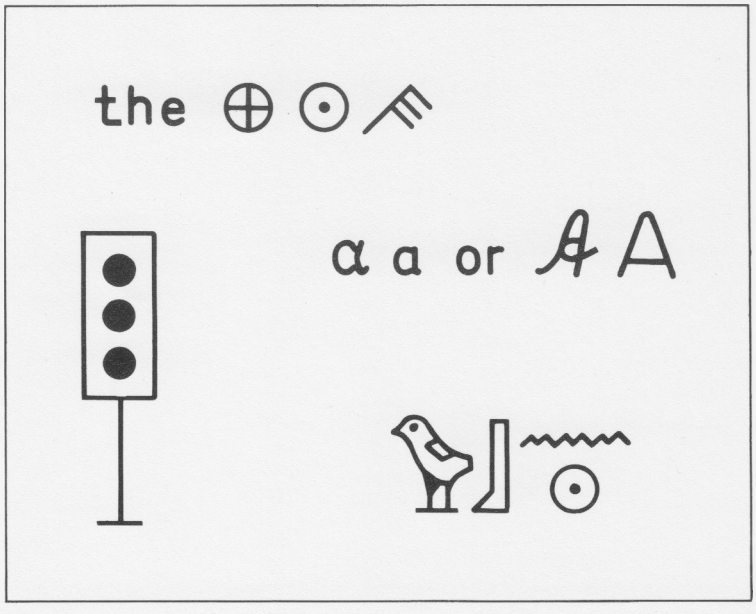


Fig. 4 Informatieoverdracht d.m.v. codes.

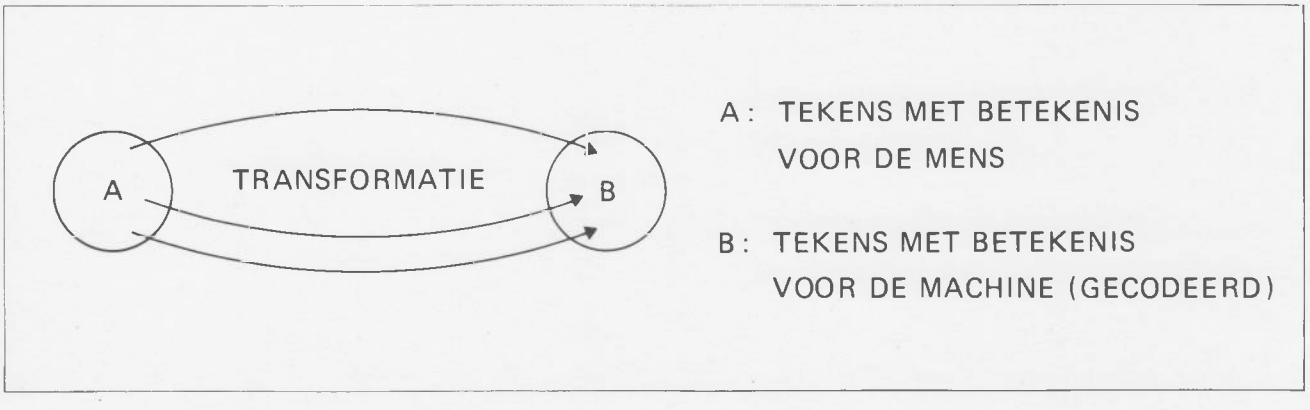


Fig. 5 Transformatie van codes.

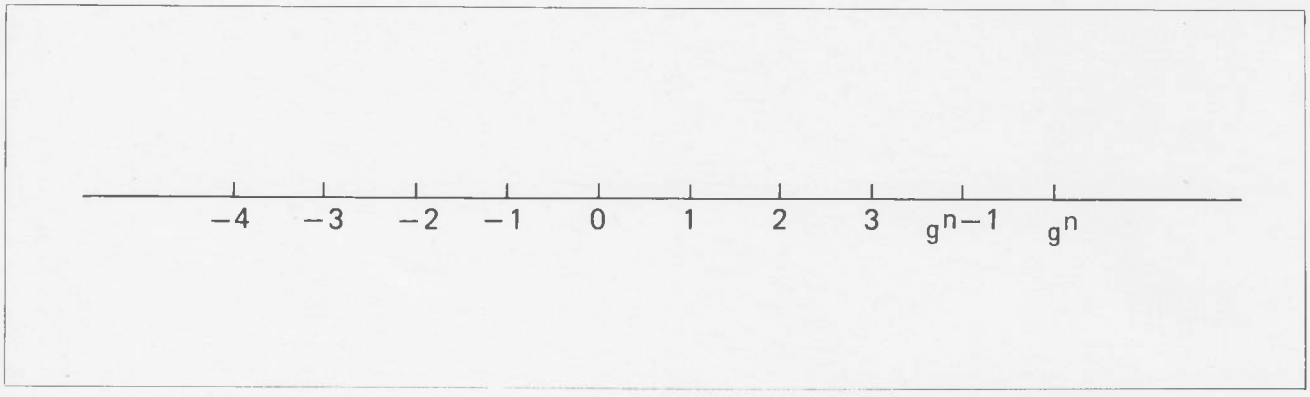


Fig. 6 De getallenrechte.

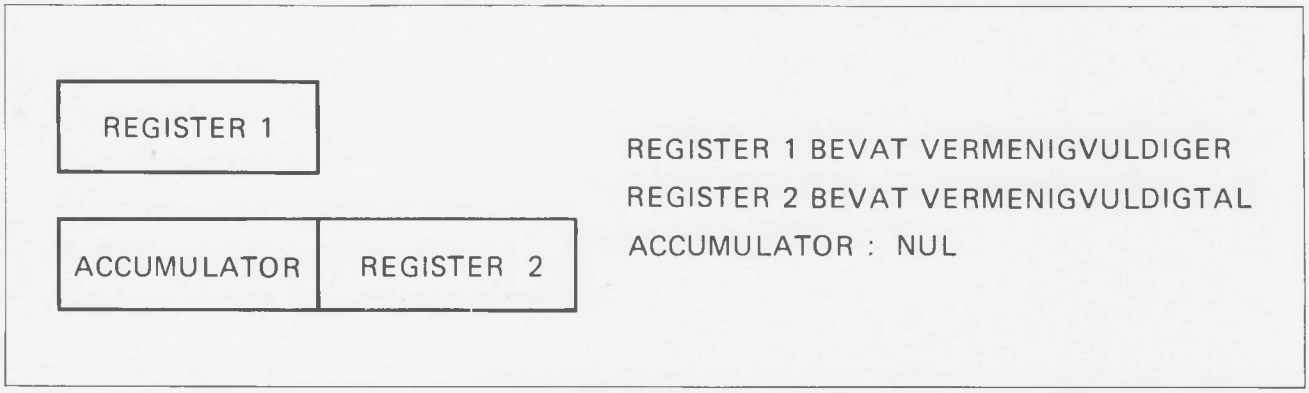


Fig. 7 De registers van de zakrekenmachine.

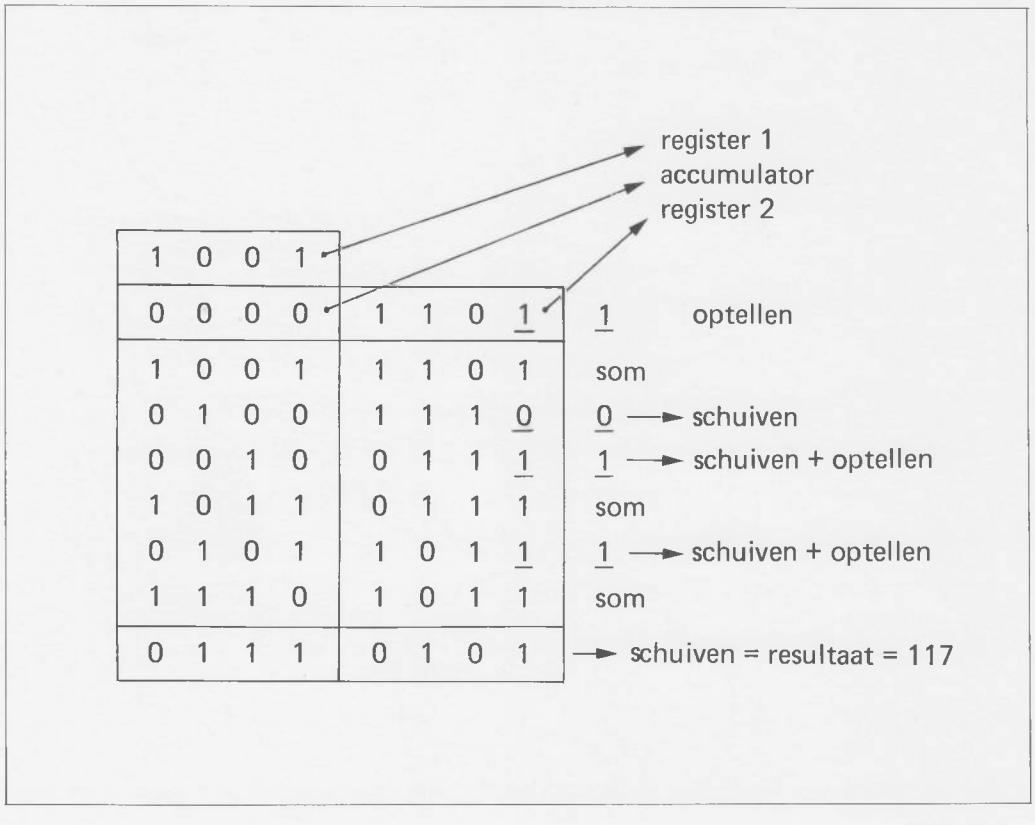
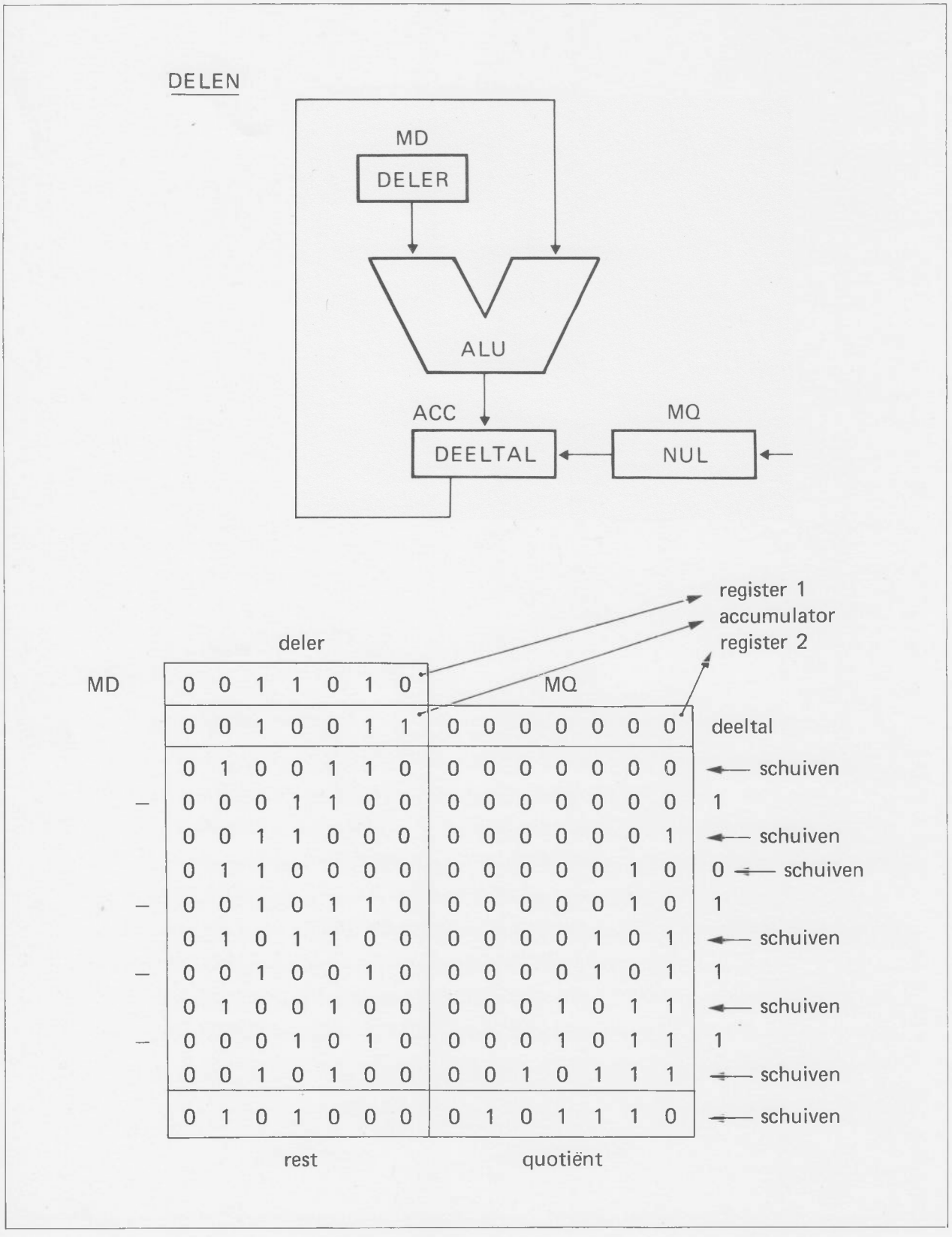


Fig. 8 Het vermenigvuldigen van binaire getallen.



MD	0	0	1	1	0	1	0								
	0	0	1	0	0	1	1	0	0	0	0	0	0	0	deeltal
	0	1	0	0	1	1	0	0	0	0	0	0	0	0	← schuiven
–	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1
	0	0	1	1	0	0	0	0	0	0	0	0	0	1	← schuiven
	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0 ← schuiven
–	0	0	1	0	1	1	0	0	0	0	0	0	1	0	1
	0	1	0	1	1	0	0	0	0	0	0	1	0	1	← schuiven
–	0	0	1	0	0	1	0	0	0	0	0	1	0	1	1
	0	1	0	0	1	0	0	0	0	0	1	0	1	1	← schuiven
–	0	0	0	1	0	1	0	0	0	0	1	0	1	1	1
	0	0	1	0	1	0	0	0	0	1	0	1	1	1	← schuiven
	0	1	0	1	0	0	0	0	1	0	1	1	1	0	← schuiven
	rest							quotiënt							

Fig. 9 Het delen van binaire getallen.

bits					7	0	0	0	0	1	1	1	1
					6	0	0	1	1	0	0	1	1
					5	0	1	0	1	0	1	0	1
bits						0	1	2	3	4	5	6	7
4	3	2	1										
0	0	0	0	0		NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1		SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2		STX	DC2	"	2	B	R	b	r
0	0	1	1	3		ETX	DC3	#	3	C	S	c	s
0	1	0	0	4		EOT	DC4	$	4	D	T	d	t
0	1	0	1	5		ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6		ACK	SYN	&	6	F	V	f	v
0	1	1	1	7		BEL	ETB	'	7	G	W	g	w
1	0	0	0	8		BS	CAN	(	8	H	X	h	x
1	0	0	1	9		HT	EM	)	9	I	Y	i	y
1	0	1	0	10		LF	SUB	*	:	J	Z	j	z
1	0	1	1	11		VT	ESC	+	;	K	[	k	{
1	1	0	0	12		FF	FS	,	<	L	\	l	¦
1	1	0	1	13		CR	GS	–	=	M	]	m	}
1	1	1	0	14		SO	RS	.	>	N	^	n	~
1	1	1	1	15		SI	US	/	?	O	-	o	DEL

Fig. 10 ASCII-code tabel.

HEXADECIMAAL–ASCII

00	NUL	33	3	66	f
01	SOH	34	4	67	g
02	STX	35	5	68	h
03	ETX	36	6	69	i
04	EOT	37	7	6A	j
05	ENQ	38	8	6B	k
06	ACK	39	9	6C	l
07	BEL	3A	:	6D	m
08	BS	3B	;	6E	n
09	HT	3C	<	6F	o
0A	LF	3D	=	70	p
0B	VT	3E	>	71	q
0C	FF	3F	?	72	r
0D	CR	40	@	73	s
0E	SO	41	A	74	t
0F	SI	42	B	75	u
10	DLE	43	C	76	v
11	DC1	44	D	77	w
12	DC2	45	E	78	x
13	DC3	46	F	79	y
14	DC4	47	G	7A	z
15	NAK	48	H	7B	{
16	SYN	49	I	7C	¦
17	ETB	4A	J	7D	}
18	CAN	4B	K	7E	~
19	EM	4C	L	7F	DEL
1A	SUB	4D	M		
1B	ESC	4E	N		
1C	FS	4F	O		
1D	GS	50	P		
1E	RS	51	Q		
1F	US	52	R		
20	SP	53	S		
21	!	54	T		
22	"	55	U		
23	#	56	V		
24	$	57	W		
25	%	58	X		
26	&	59	Y		
27	'	5A	Z		
28	(	5B	[		
29	)	5C	\		
2A	*	5D	]		
2B	+	5E	^		
2C	,	5F	–		
2D	–	60	`		
2E	.	61	a		
2F	/	62	b		
30	0	63	c		
31	1	64	d		
32	2	65	e		

Fig. 11 Hexadecimaal – ASCII conversie.

bits					8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
					7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
					6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
					5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
bits						0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
4	3	2	1																		
0	0	0	0	0		NUL	DLE			SP	&	–						{	}	\	0
0	0	0	1	1		SOH	DC1					/		a	j	~		A	J		1
0	0	1	0	2		STX	DC2		SYN					b	k	s		B	K	S	2
0	0	1	1	3		ETX	DC3							c	l	t		C	L	T	3
0	1	0	0	4										d	m	u		D	M	U	4
0	1	0	1	5		HT		LF						e	n	v		E	N	V	5
0	1	1	0	6			BS	ETB						f	o	w		F	O	W	6
0	1	1	1	7		DEL		ESC	EOT					g	p	x		G	P	X	7
1	0	0	0	8			CAN							h	q	y		H	Q	Y	8
1	0	0	1	9			EM							i	r	z		I	R	Z	9
1	0	1	0	A						¢	!	¦	:								
1	0	1	1	B		VT				.	$	,	#								
1	1	0	0	C		FF	FS		DC4	<	*	%	@								
1	1	0	1	D		CR	GS	ENQ	NAK	(	)	—	'								
1	1	1	0	E		SO	RS	ACK		+	;	>	=								
1	1	1	1	F		SI	US	BEL	SUB	|	¬	?	"								

Fig. 12 EBCDIC-code tabel.

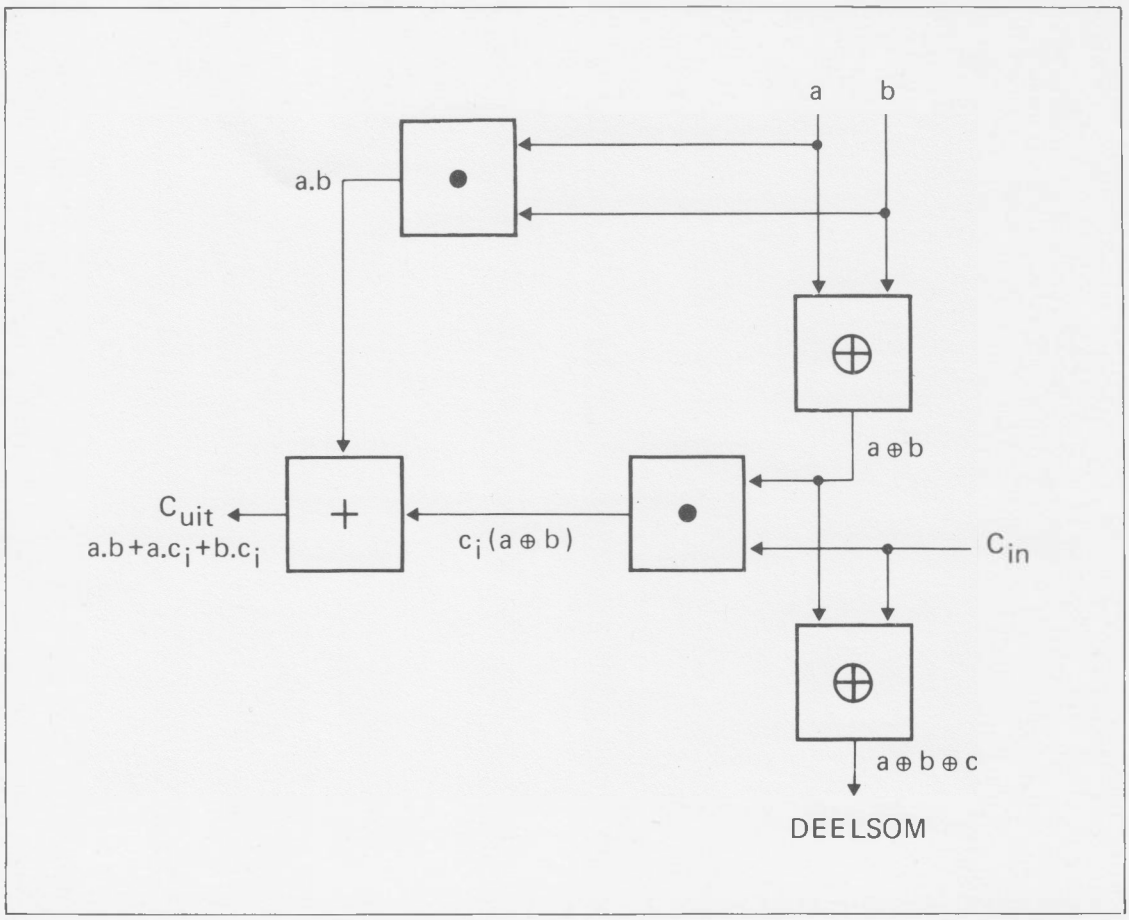


Fig. 13 Full-adder.

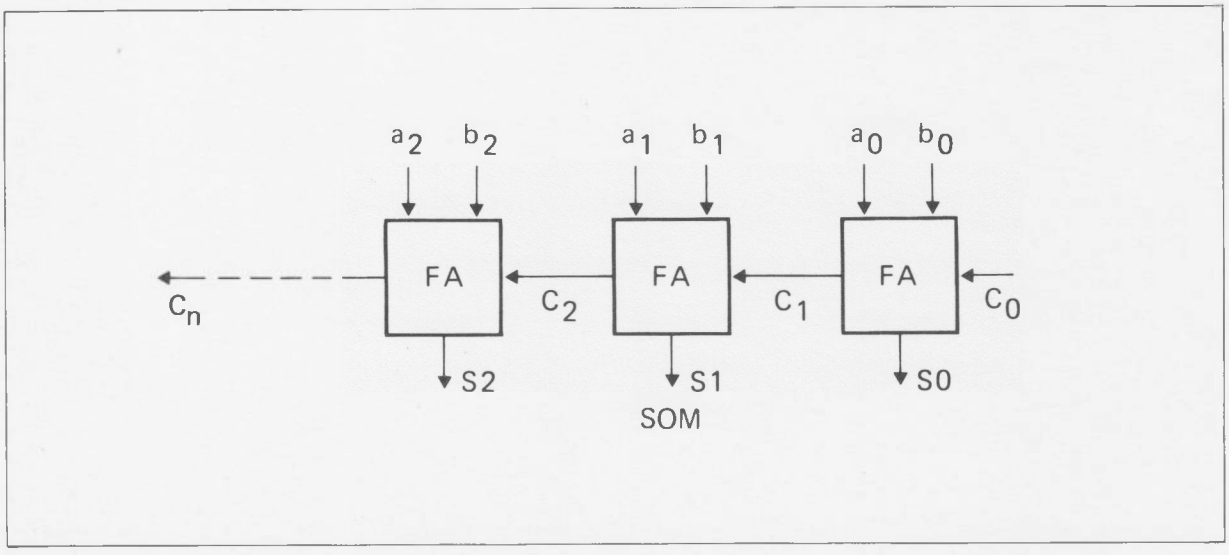


Fig. 14 Optelmechanisme van de zakrekenmachine.

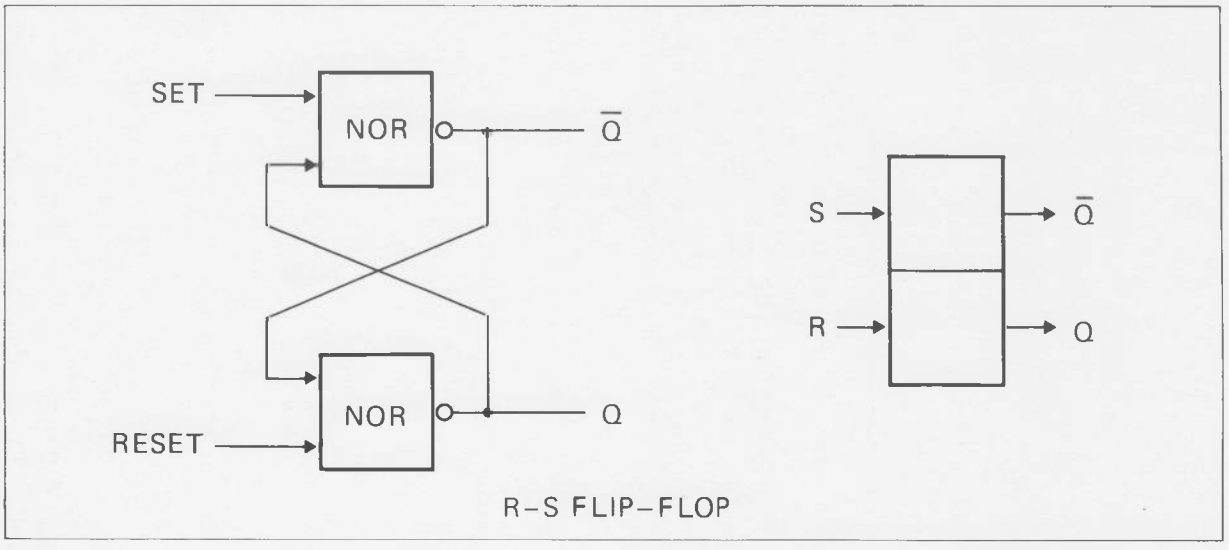


Fig. 15 Set-Reset flip-flop.

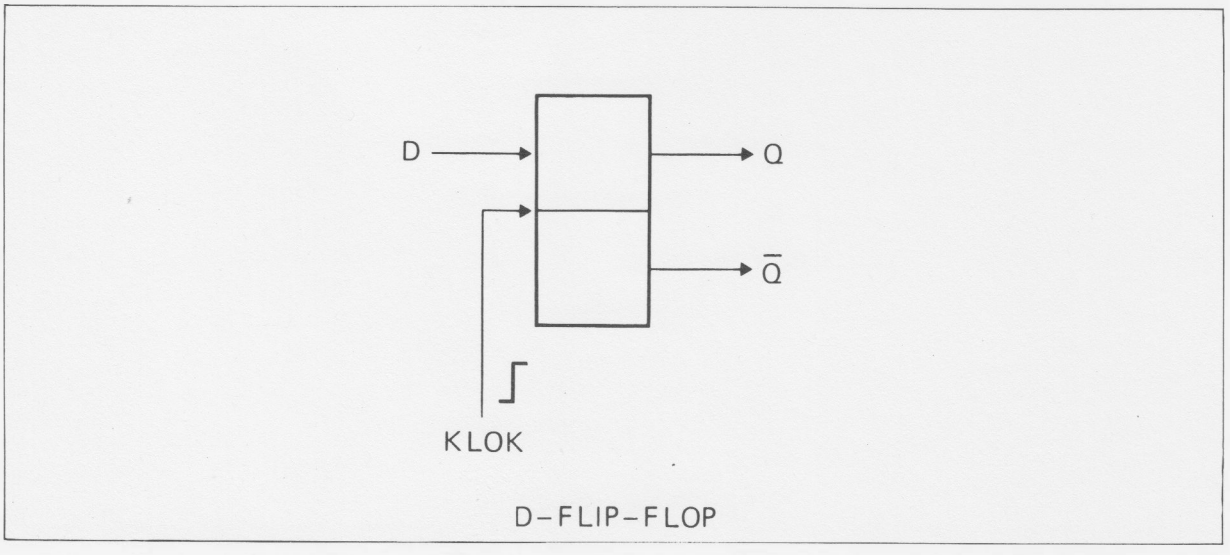


Fig. 16 D-flip-flop.

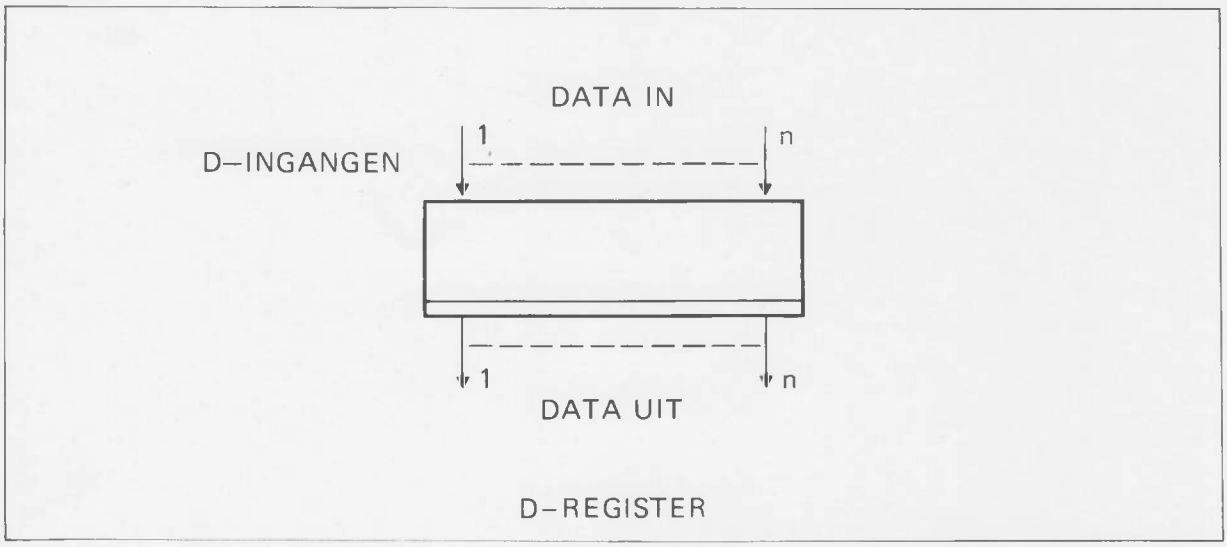


Fig. 17 D-register.

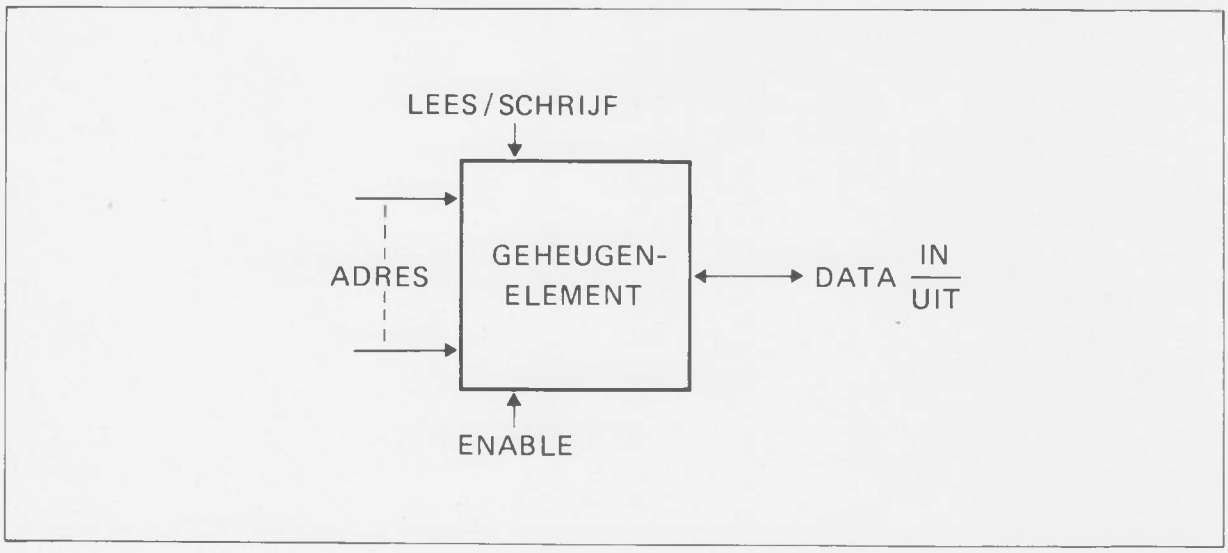
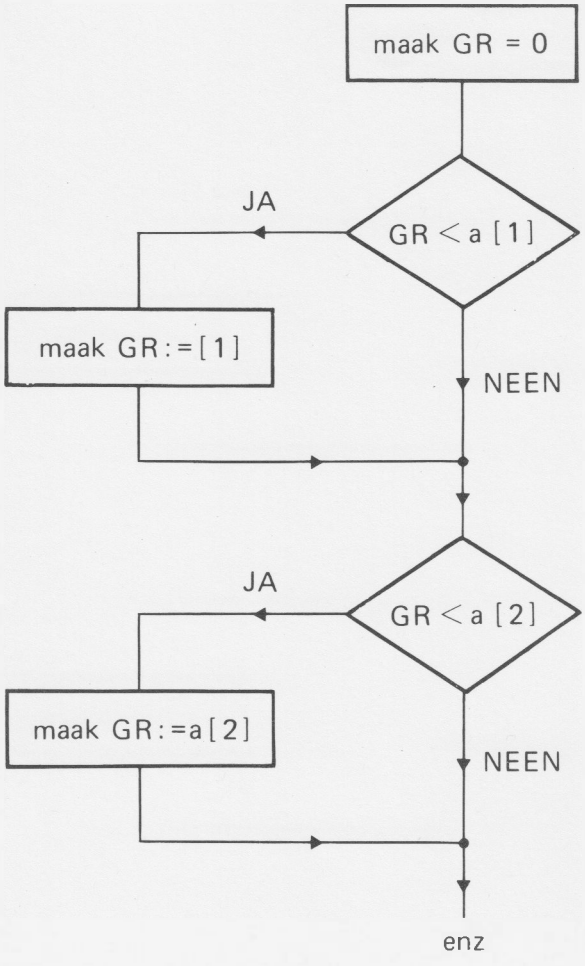


Fig. 18 Het geheugen.



GR : = 0 ;

als GR < a [1] dan GR : = a [1] ;

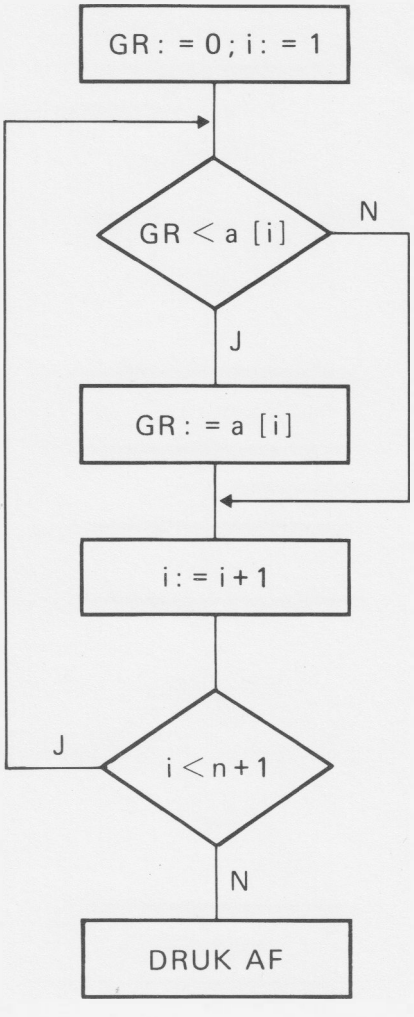
als GR < a [2] dan GR : = a [2] ;

als GR < a [3] dan GR : = a [3] ;

enz

a [1] , a [2] , a [3] , - - - - - heet een vector

Fig. 19 Stroomdiagram voor een proces I.




```
BEGIN     GR : = 0 ; i : = 1 ;

START :   ALS GR < a [i] DAN GR : = a [i] ;
          i : = i + 1 ;
          ALS i < n + 1  GA DAN NAAR START ;
          DRUK AF

EINDE
```

Fig. 20 Stroomdiagram voor een proces II.

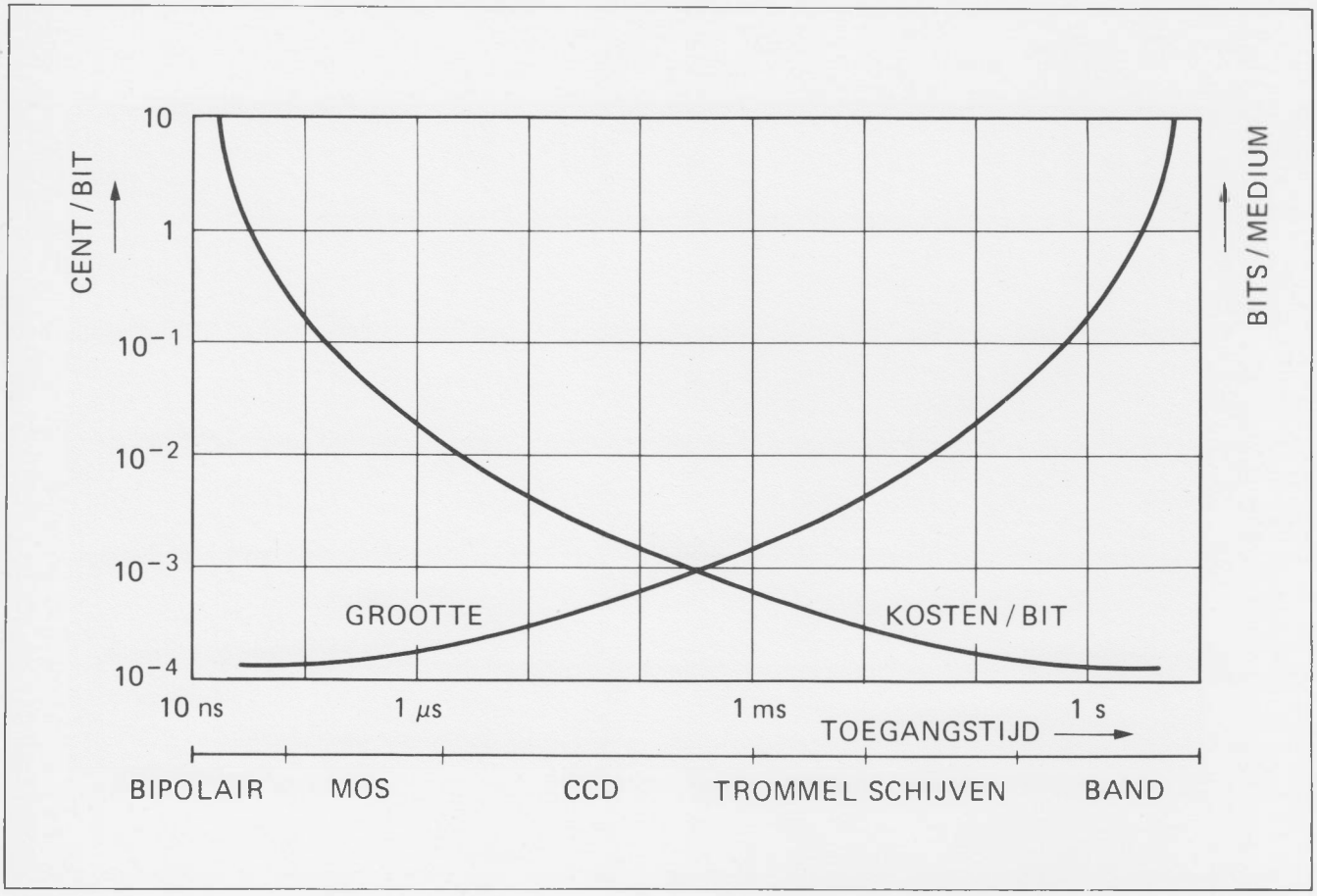


Fig. 21 Relatie tussen toegangstijd, kosten en grootte van geheugenmedia.

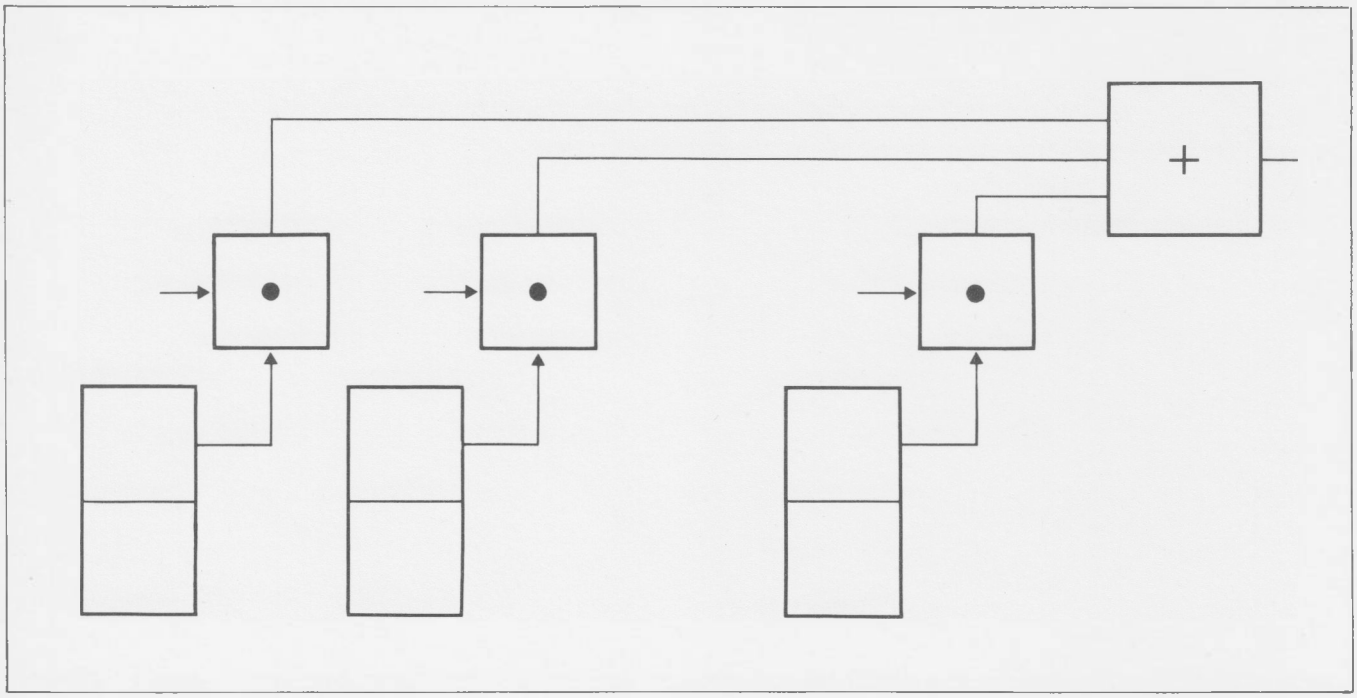


Fig. 22 Adresseren van een geheugencel I.

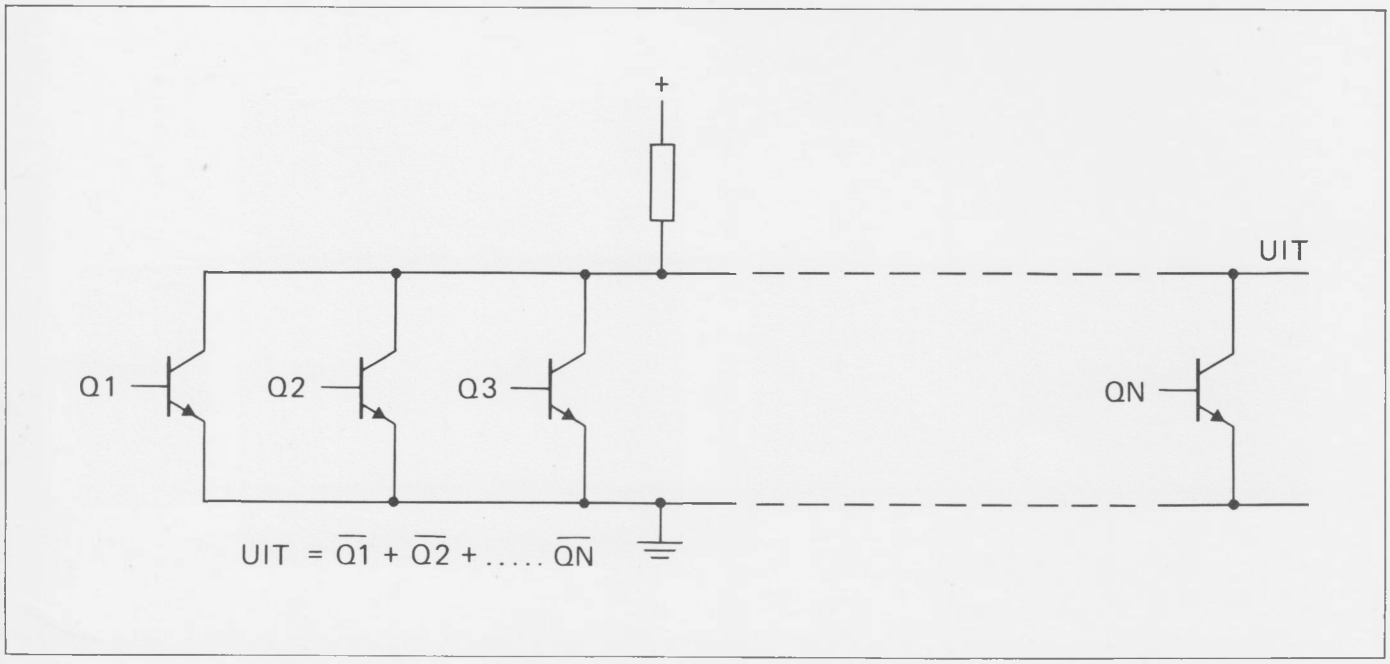


Fig. 23 Adresseren van een geheugencel II.

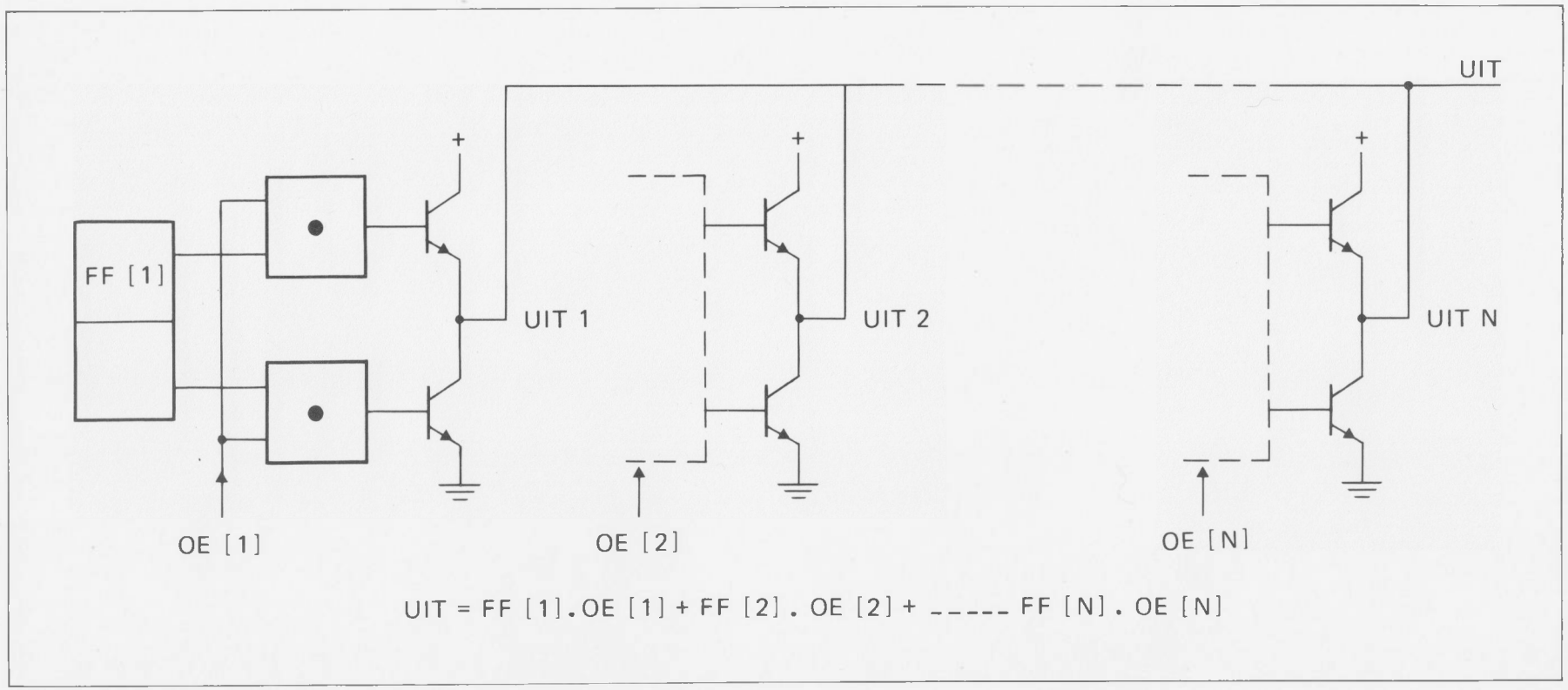


Fig. 24 Adresseren van een geheugencel III.

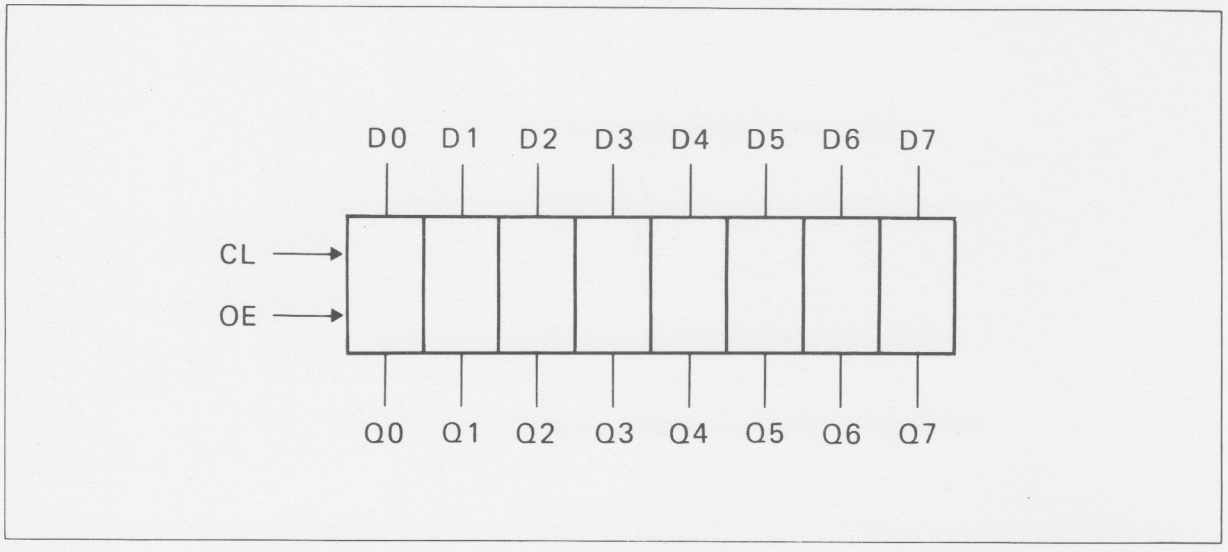


Fig. 25(a) Register.

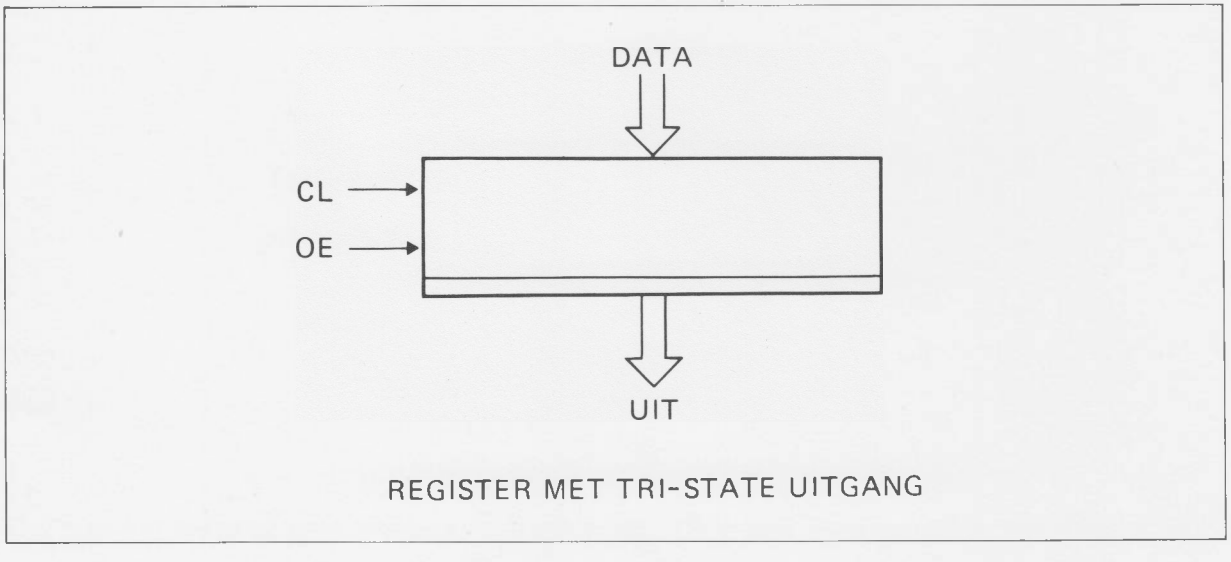


Fig. 25(b) Register met tri-state uitgang.

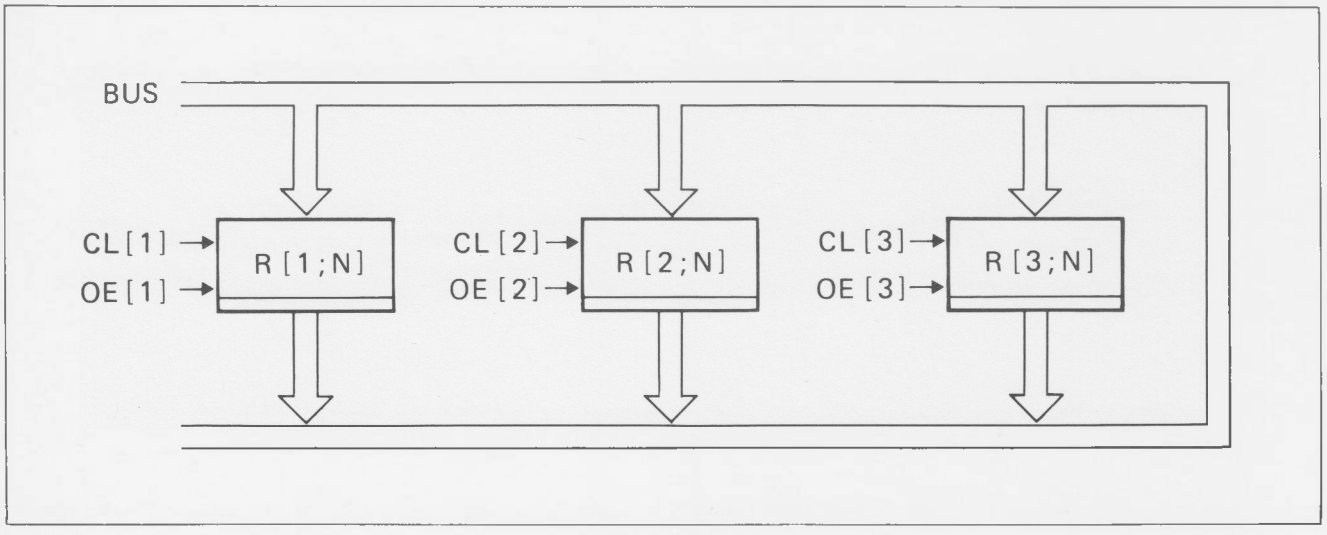


Fig. 26 De data-bus.

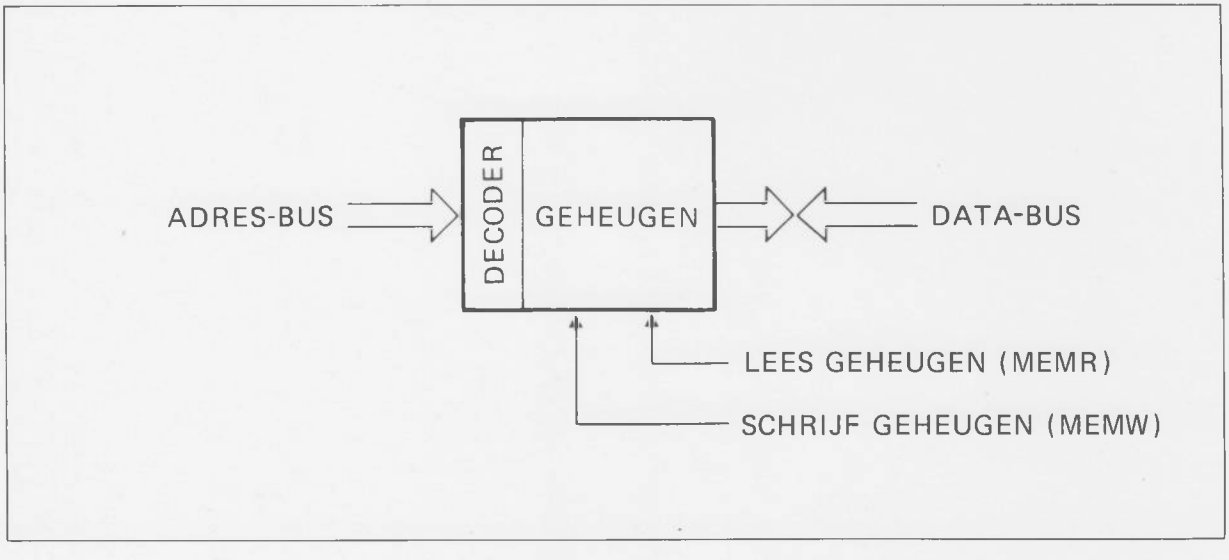


Fig. 27 Geheugenelement.

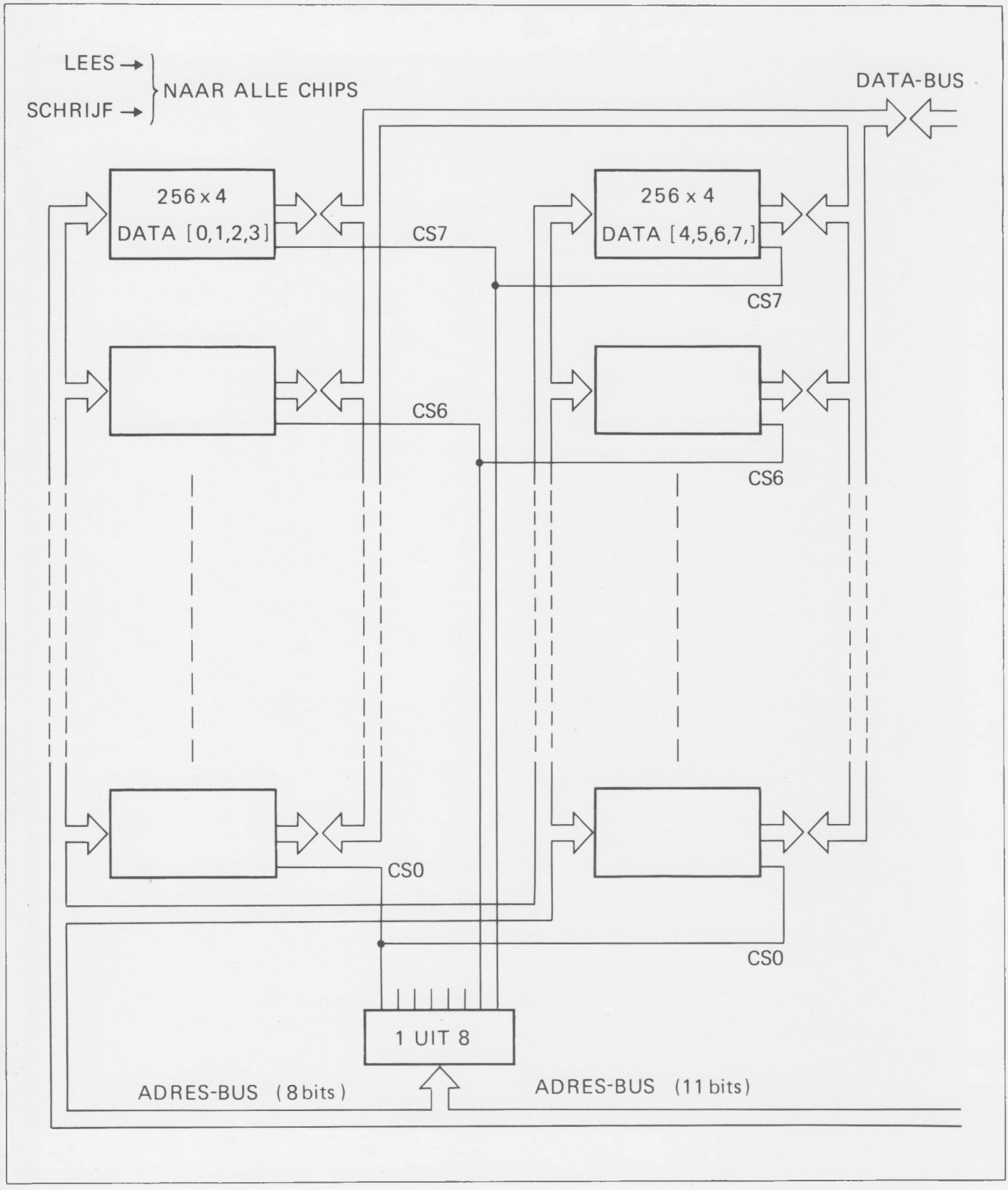


Fig. 28 Geheugen opgebouwd uit geheugenelementen.

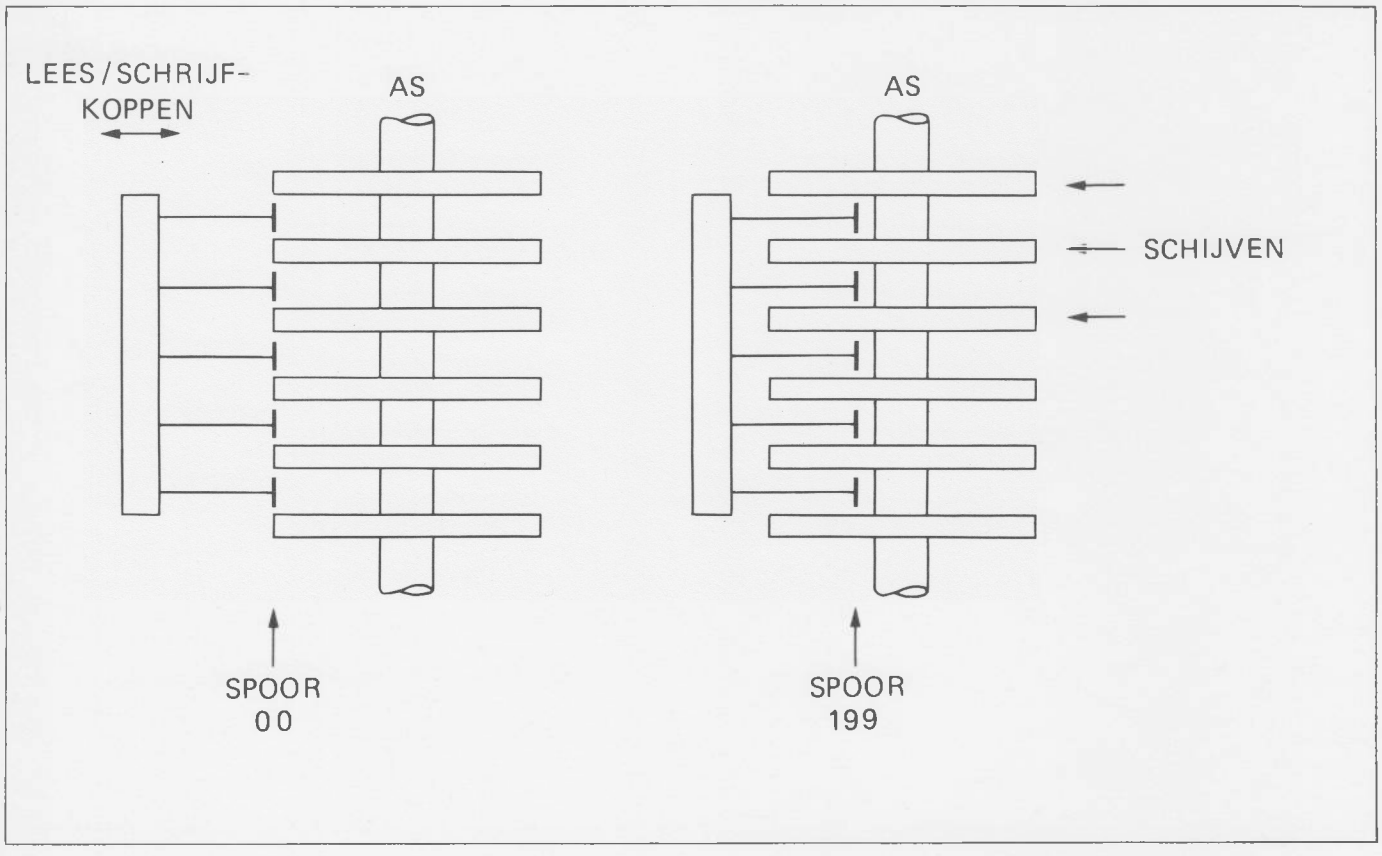


Fig. 29 Schijvengeheugen.

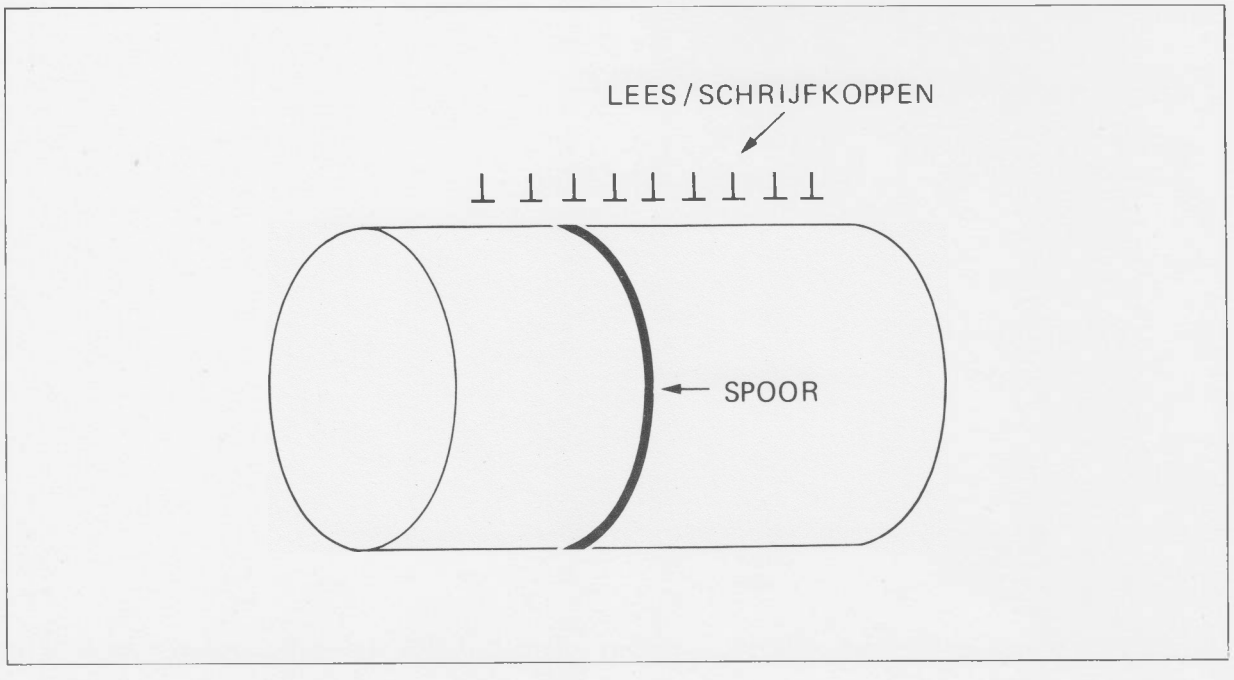


Fig. 30 Trommelgeheugen.

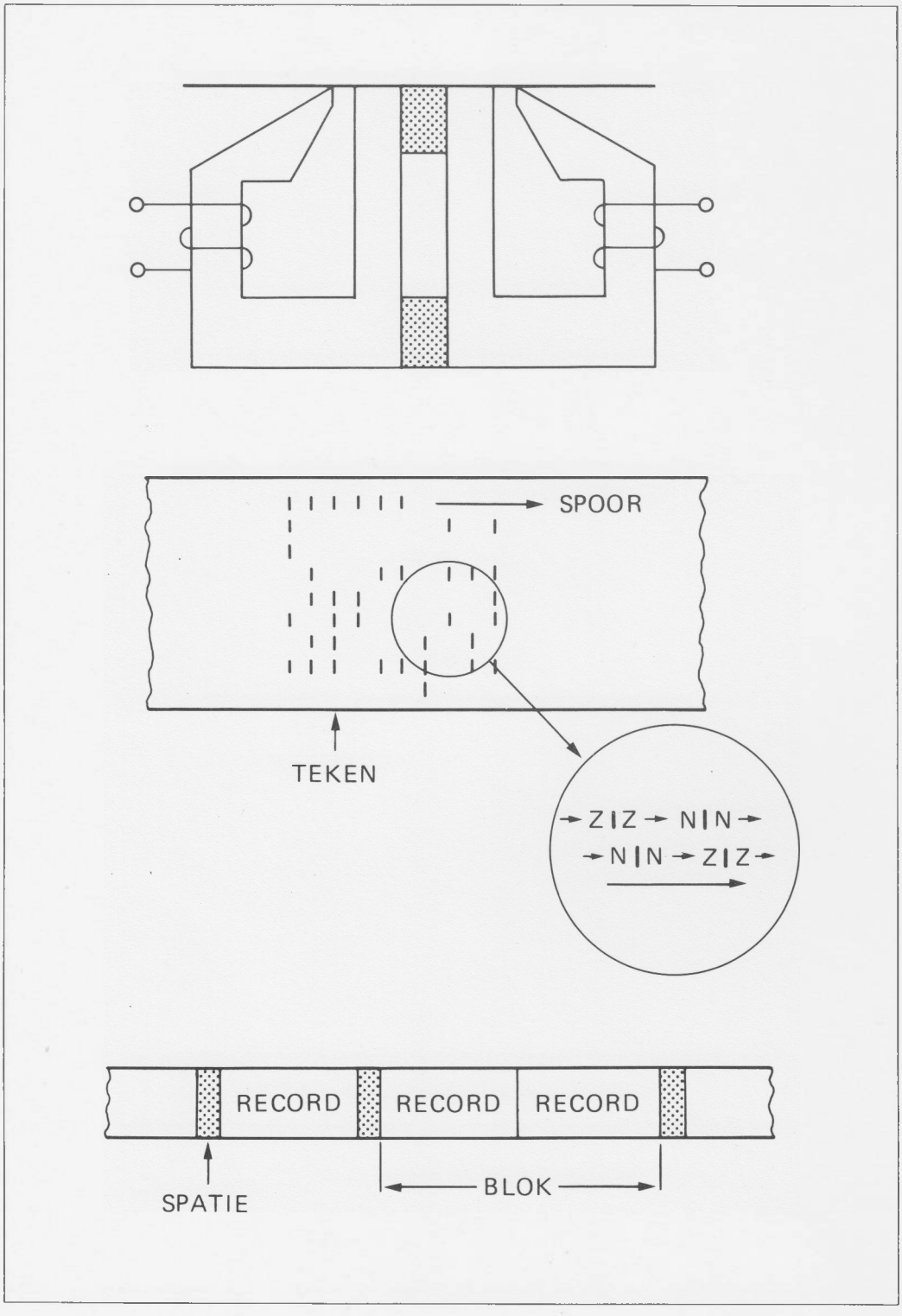


Fig. 31 Magnetisch bandgeheugen.

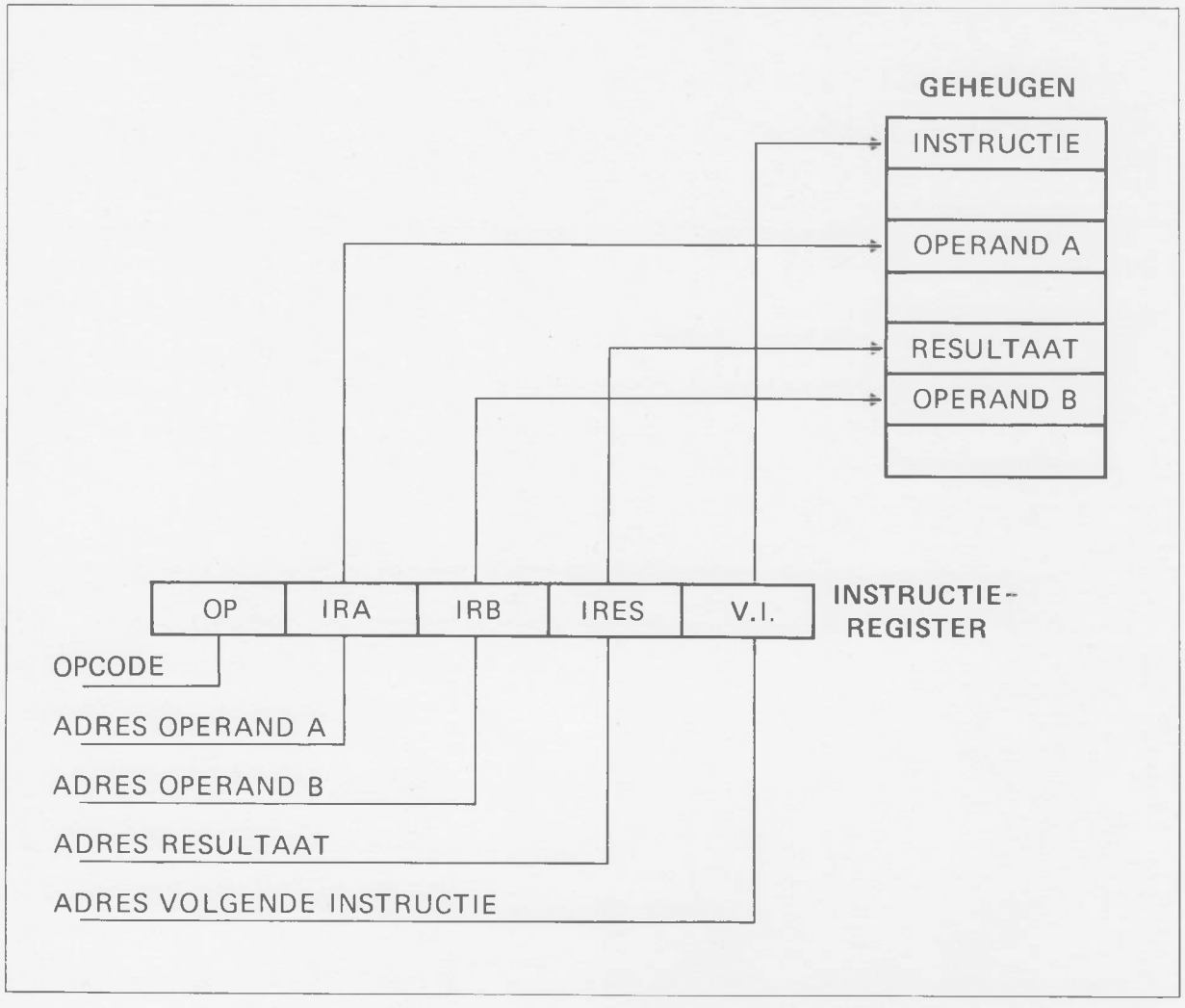


Fig. 32 De 4-adres machine.

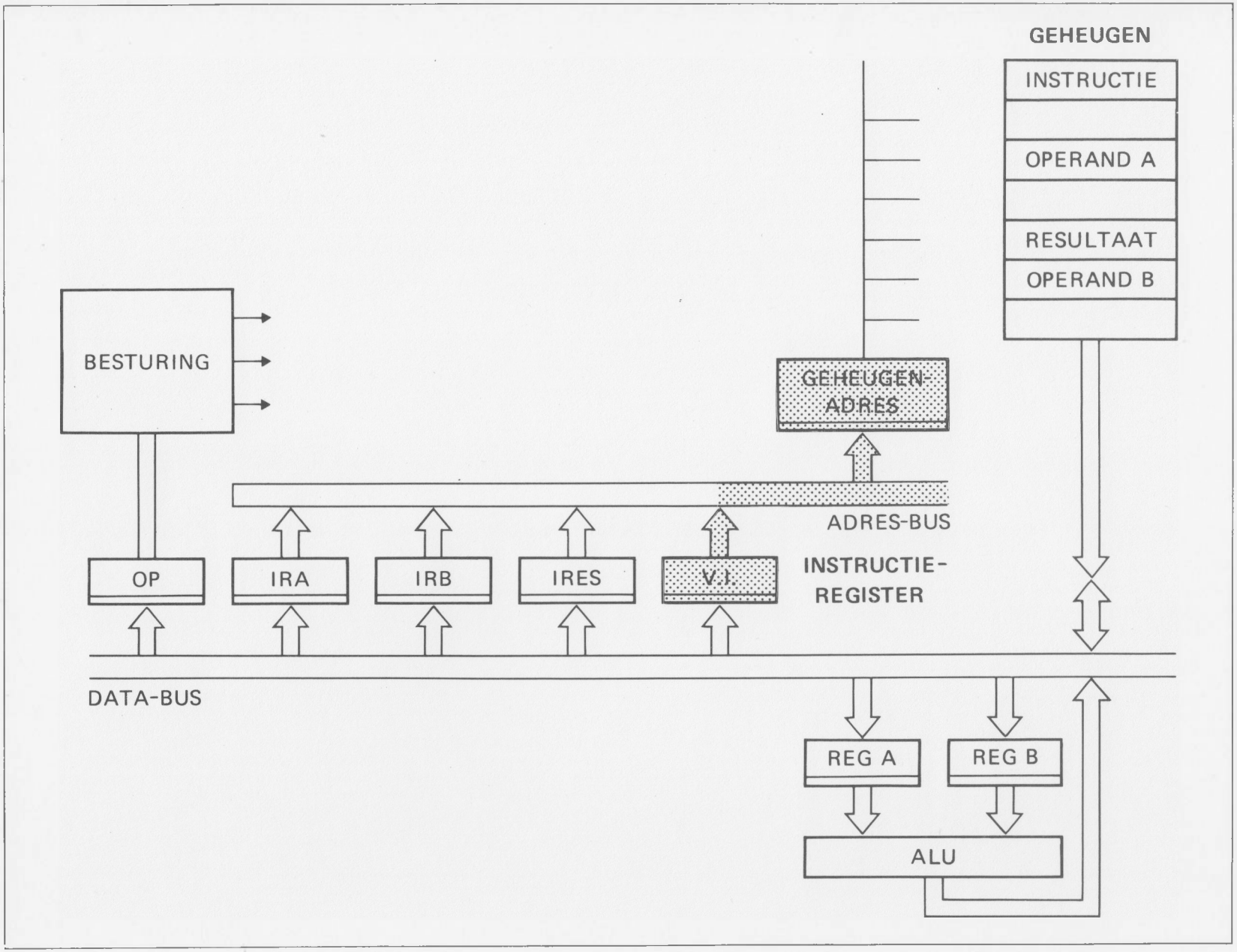


Fig. 33 Reductie van de 4-adresmachine tot de 1-adresmachine.

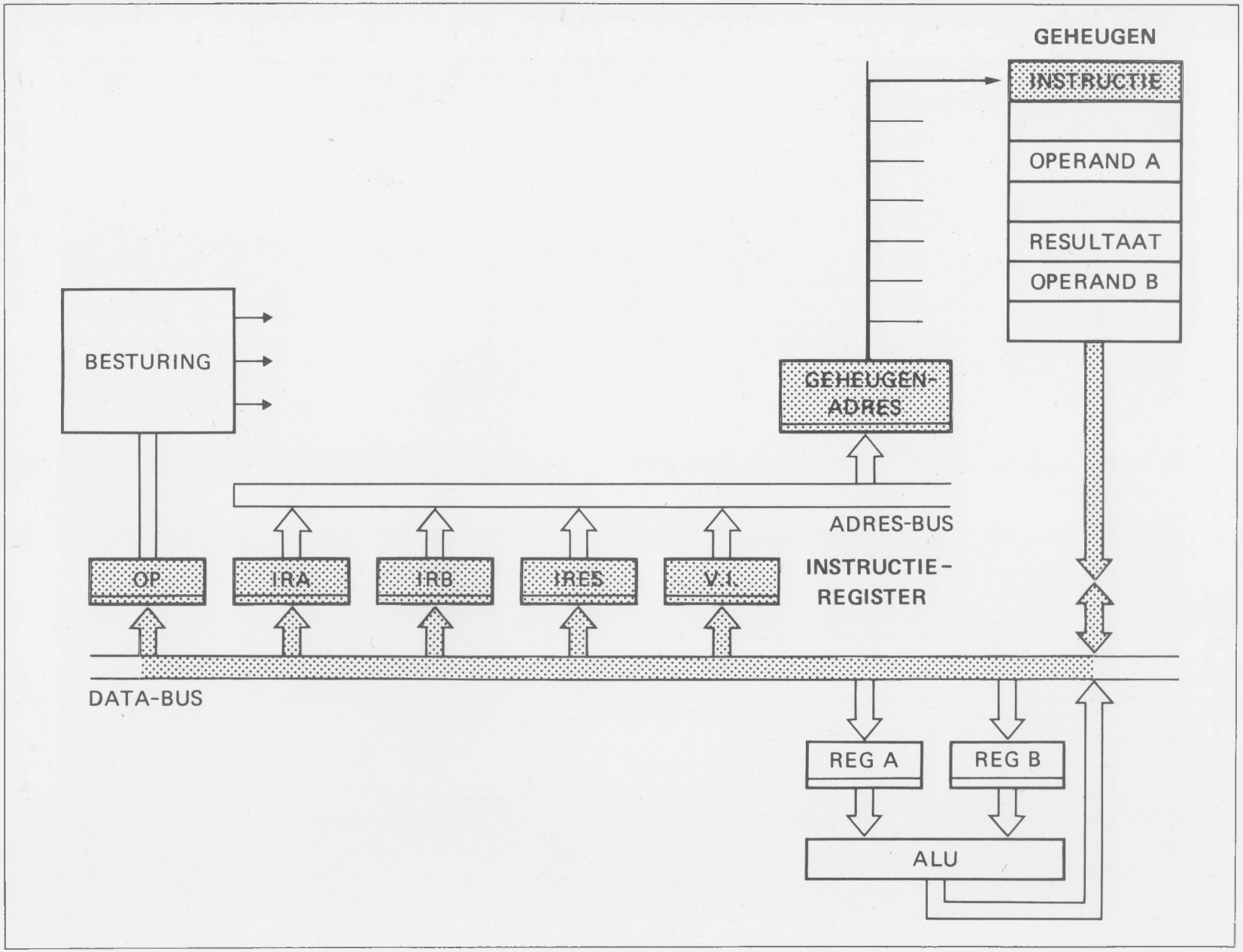


Fig. 34 Reductie van de 4-adresmachine tot de 1-adresmachine.

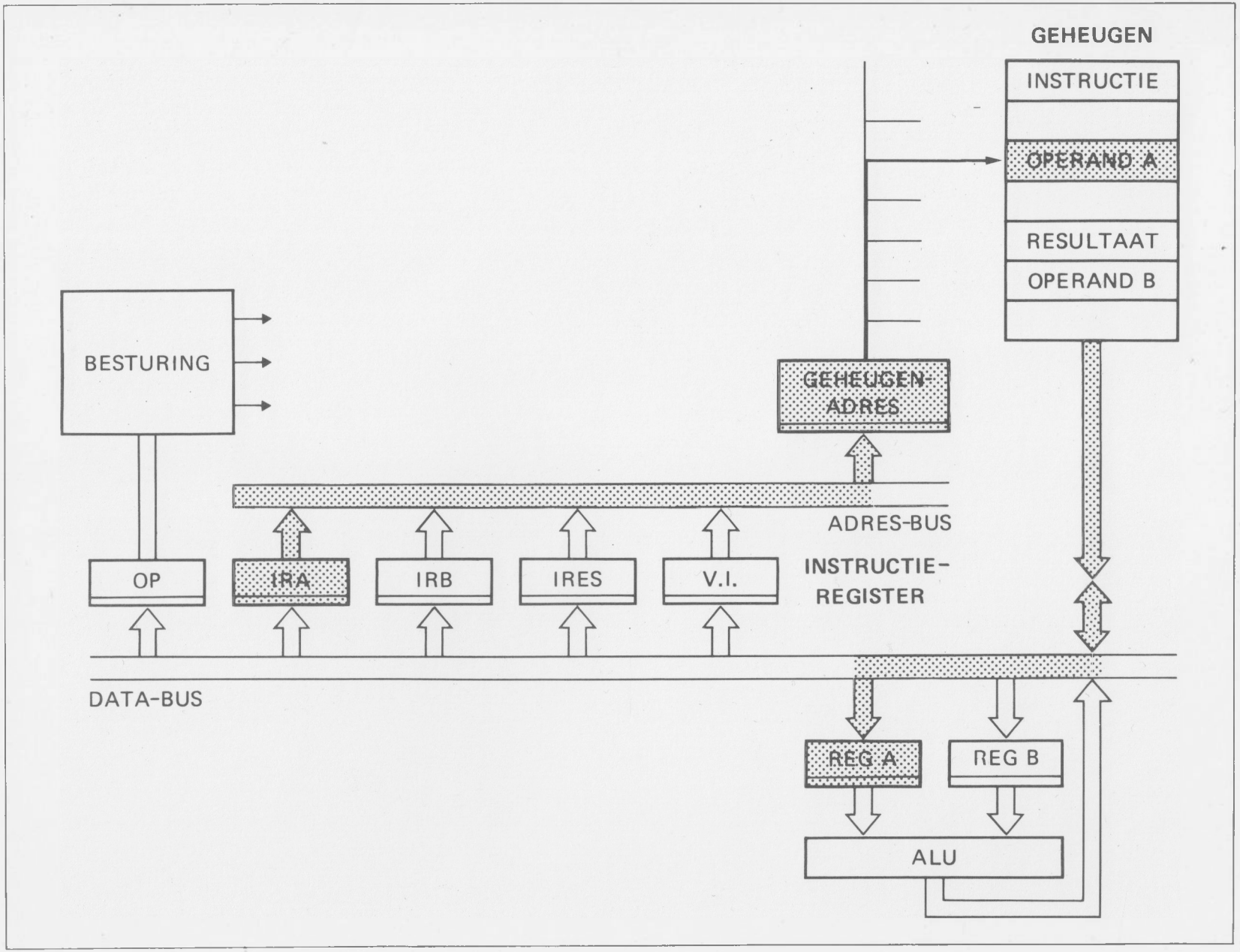


Fig. 35 Reductie van de 4-adresmachine tot de 1-adresmachine.

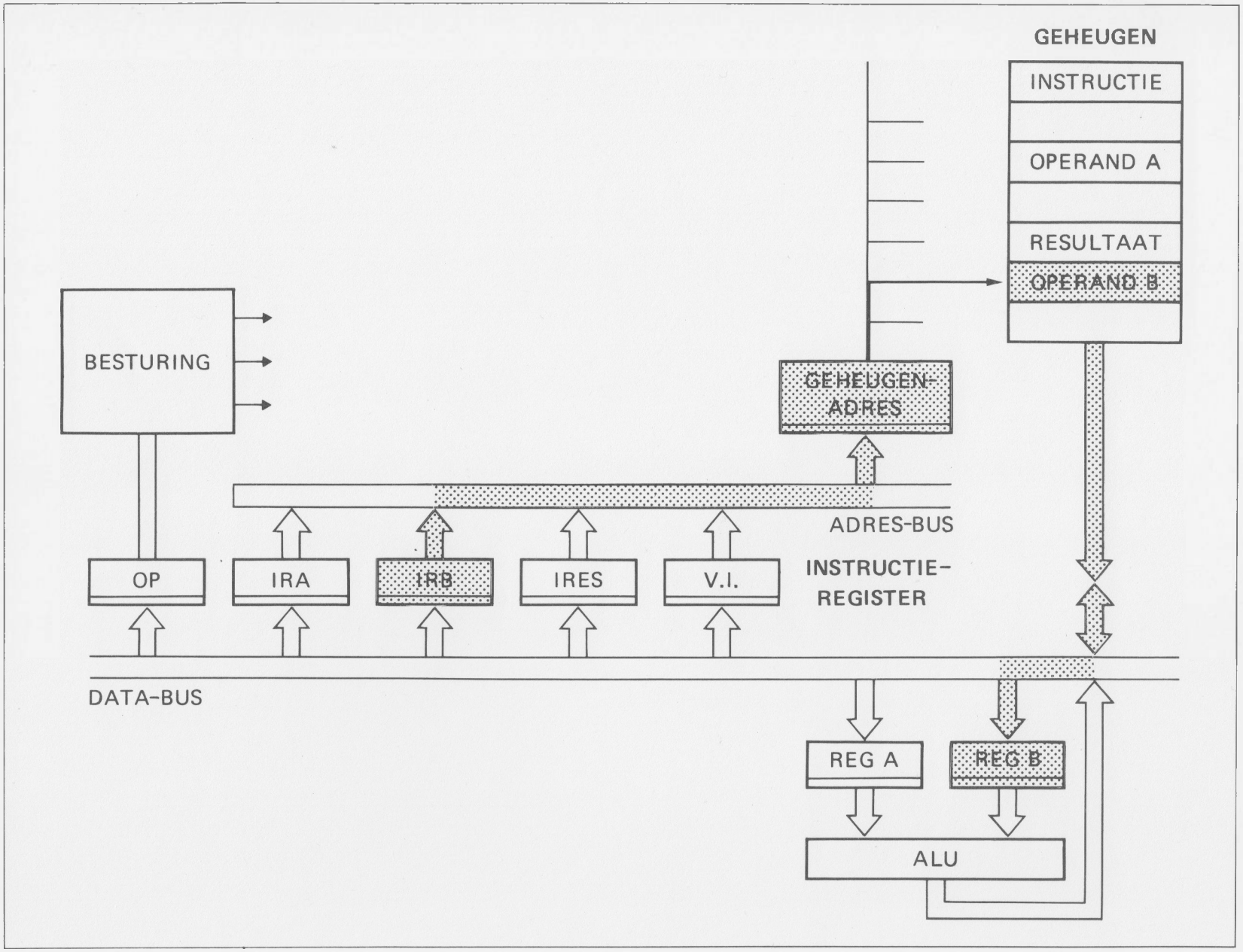


Fig. 36 Reductie van de 4-adresmachine tot de 1-adresmachine.

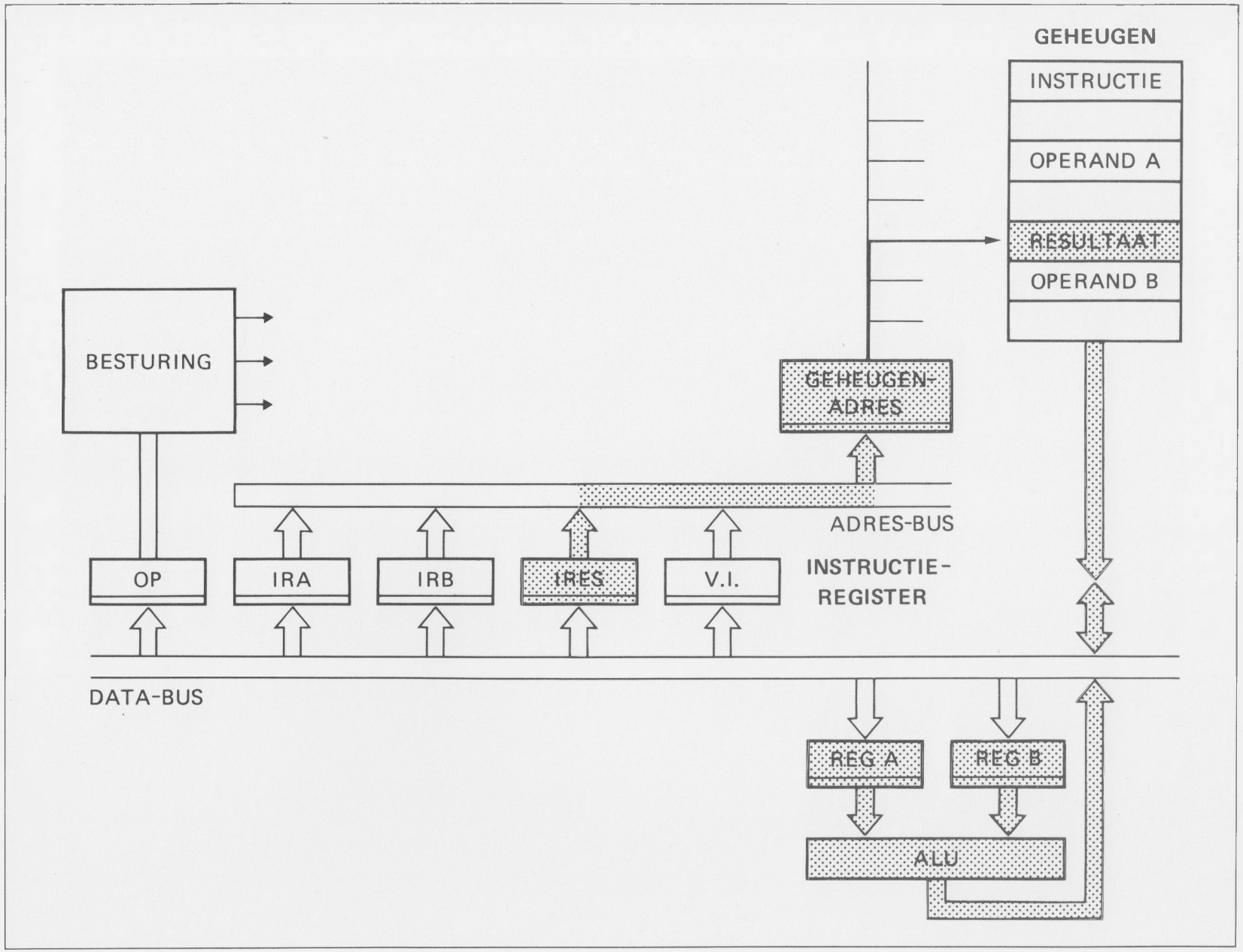


Fig. 37 Reductie van de 4-adresmachine tot de 1-adresmachine.

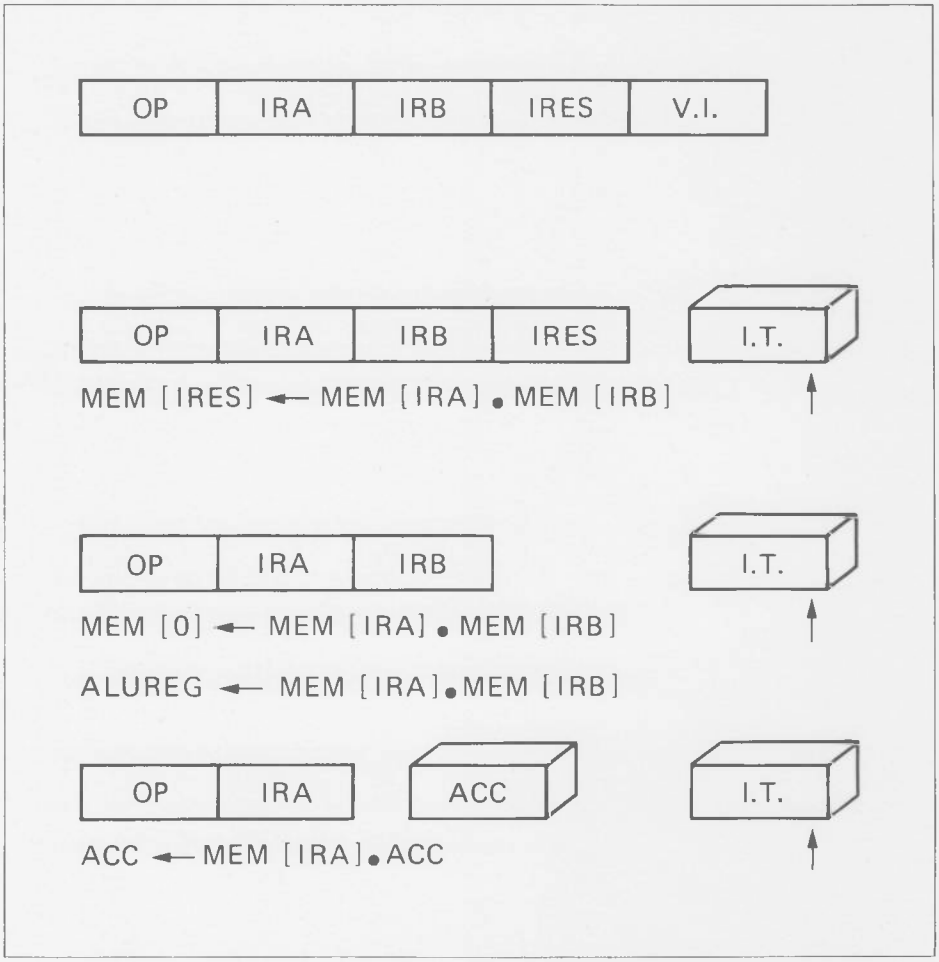


Fig. 38 Reductie van de 4-adresmachine tot de 1-adresmachine.

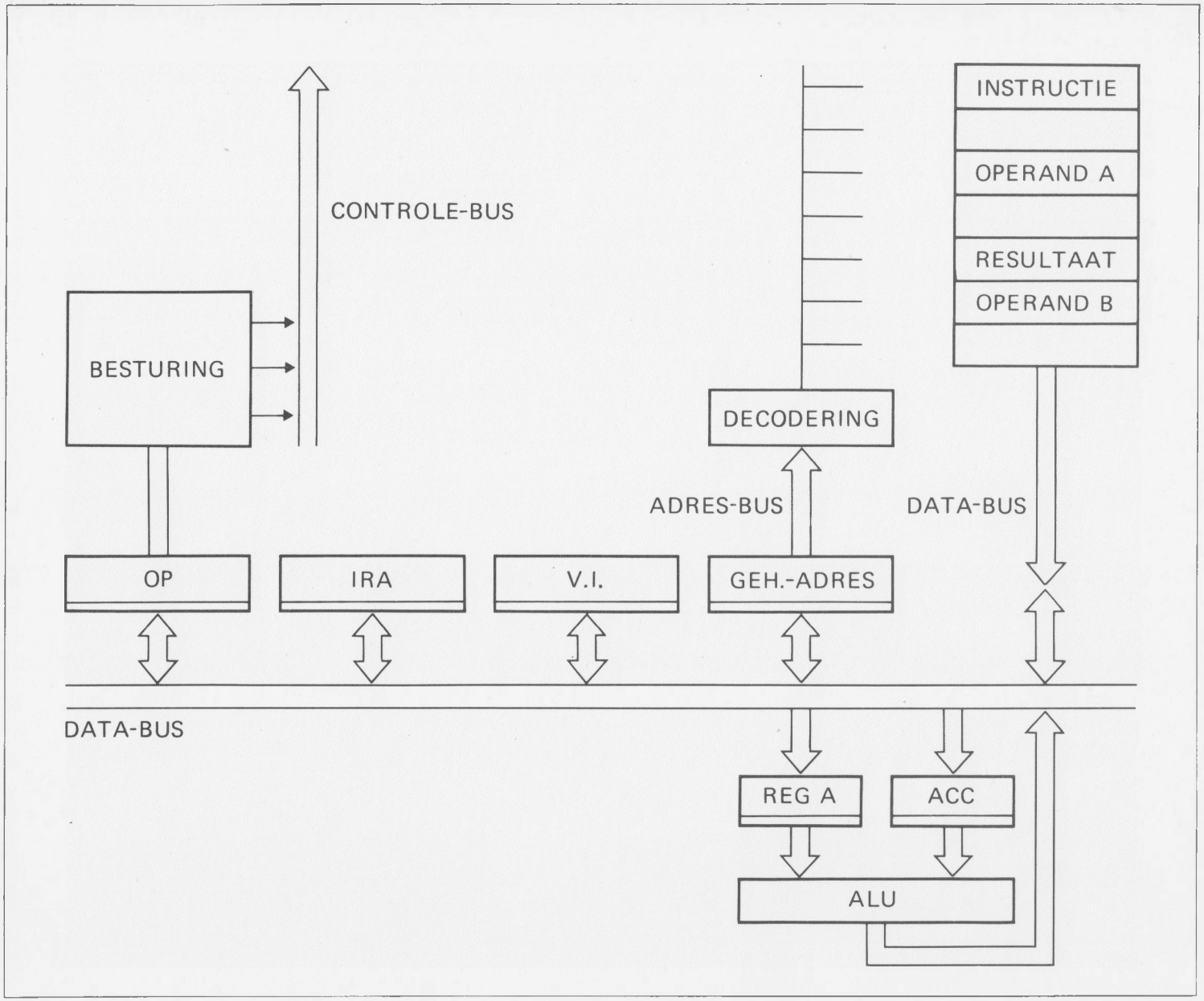


Fig. 39 Reductie van de 4-adresmachine tot de 1-adresmachine.

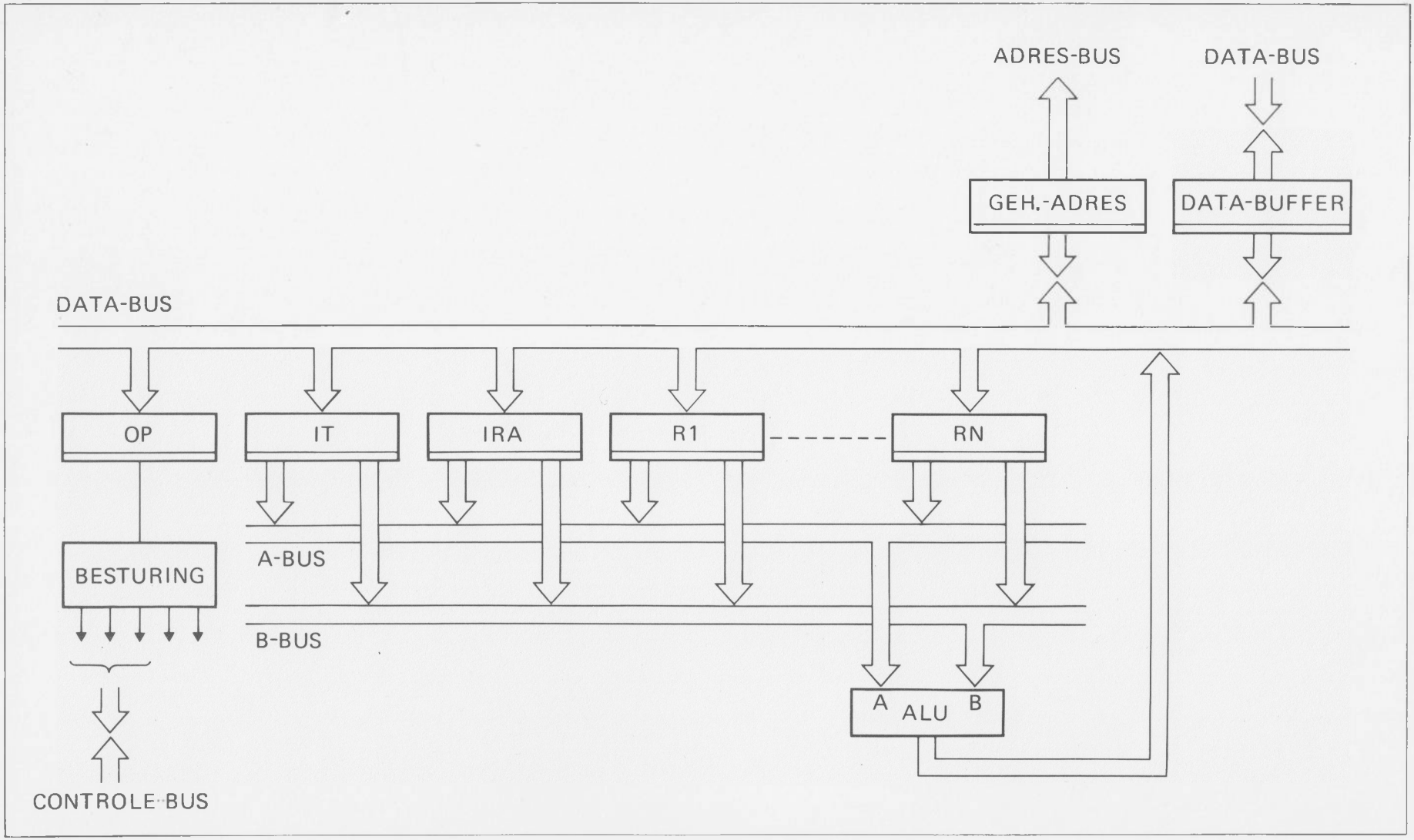


Fig. 40 Reductie van de 4-adresmachine tot de 1-adresmachine.

PSU

7	6	5	4	3	2	1	0
S	F	II	–	–	SP2	SP1	SP0

BIT	BETEKENIS
0, 1, 2	STAPELWIJZER (STACK POINTER)
3, 4	NIET IN GEBRUIK
5	INTERRUPT INHIBIT (INT. NIET GEHONOREERD)
6	FLAG (PEN 40)
7	SENSE (PEN 1)

PSL

7	6	5	4	3	2	1	0
CC1	CC0	IDC	RS	WC	OVF	COM	C

BIT	BETEKENIS
0	CARRY
1	COMPARE (0 REKENKUNDIG, 1 LOGISCH)
2	OVERFLOW
3	WITH CARRY (1 ALS CARRY WORDT GEBRUIKT)
4	REGISTER-BANK SELECTIE
5	INTERCIJFER CARRY (BIJ BCD CODES)
6, 7	CONDITIE CODE BITS

Fig. 41 Het programma status woord (PSW).

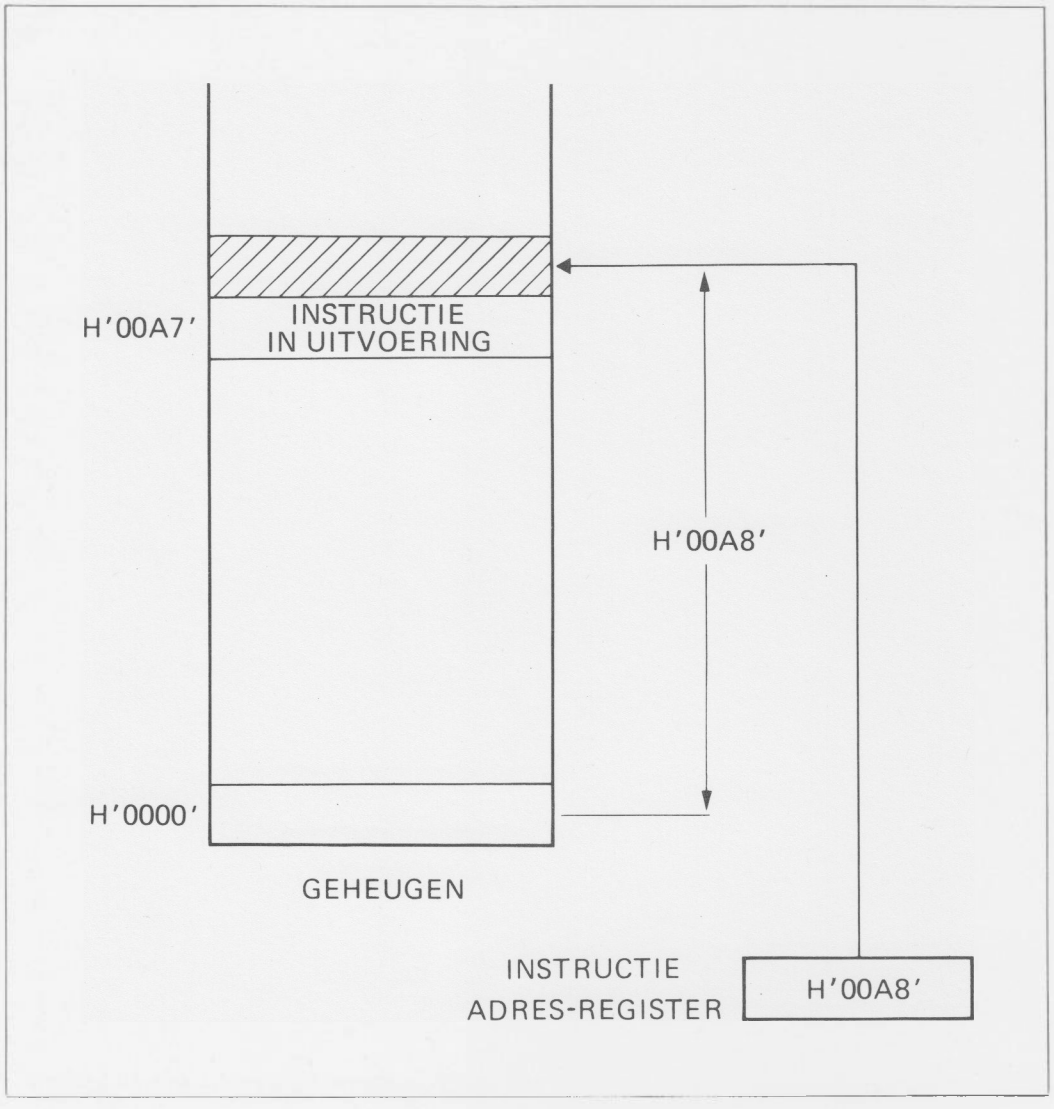


Fig. 42 Het instructie adres-register.

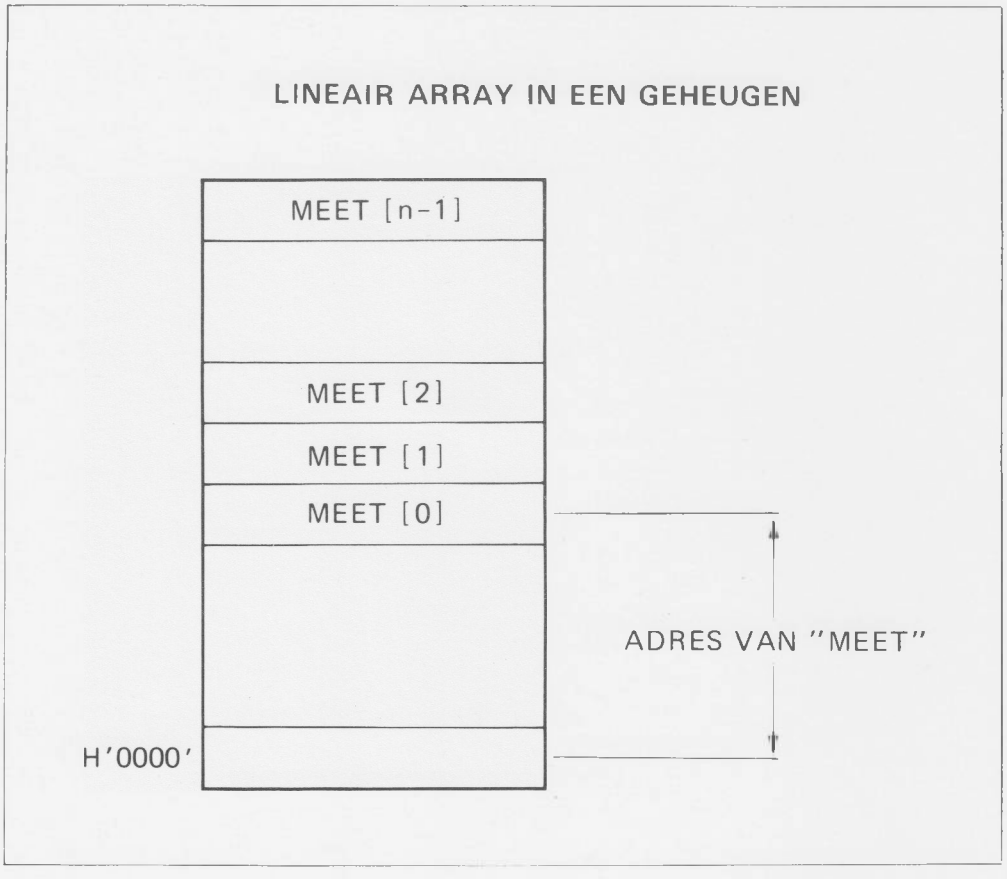


Fig. 43 Lineair array in een geheugen.

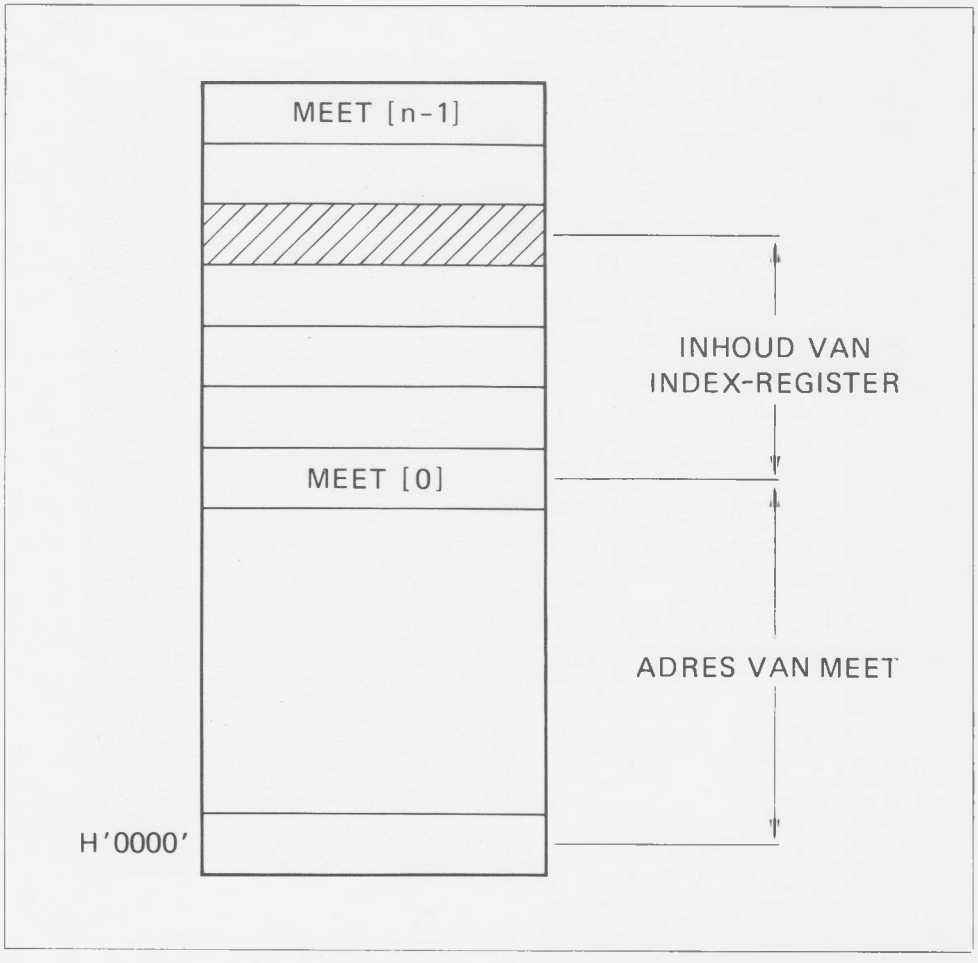


Fig. 44 Gebruik van een index-register.

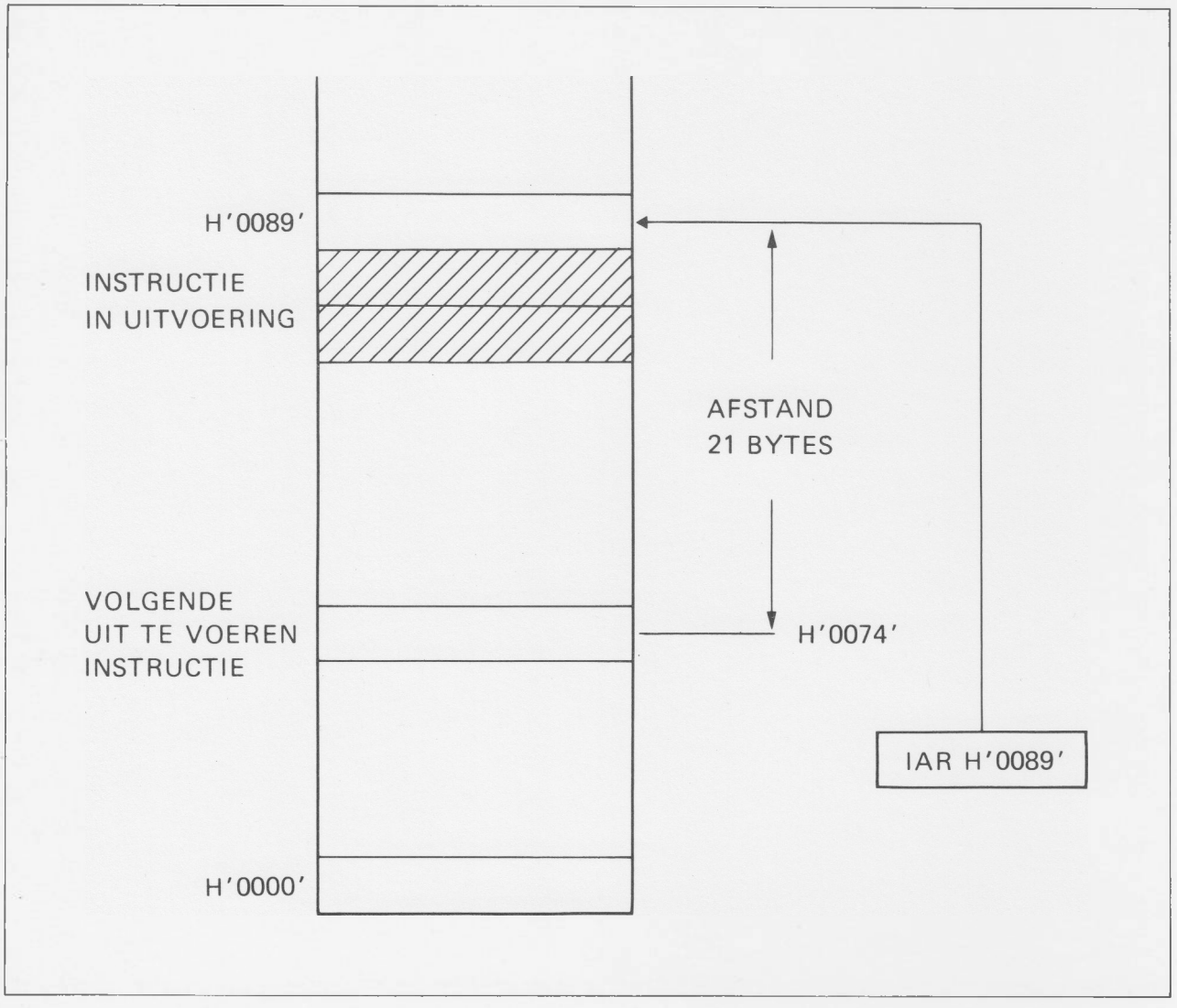


Fig. 45 Relatieve adressering.

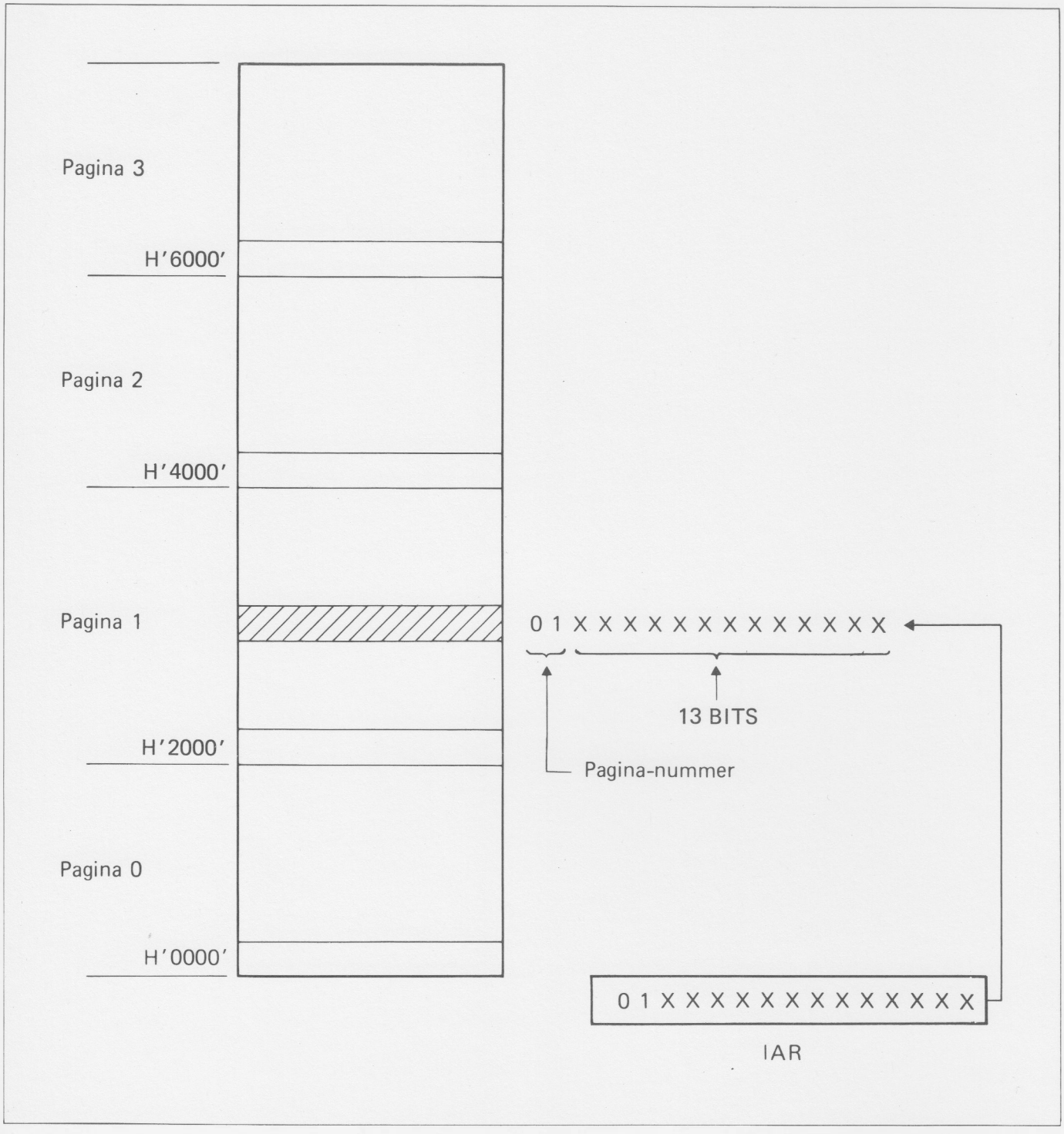


Fig. 46 Opdeling van de 2650 adresruimte in vier pagina's.

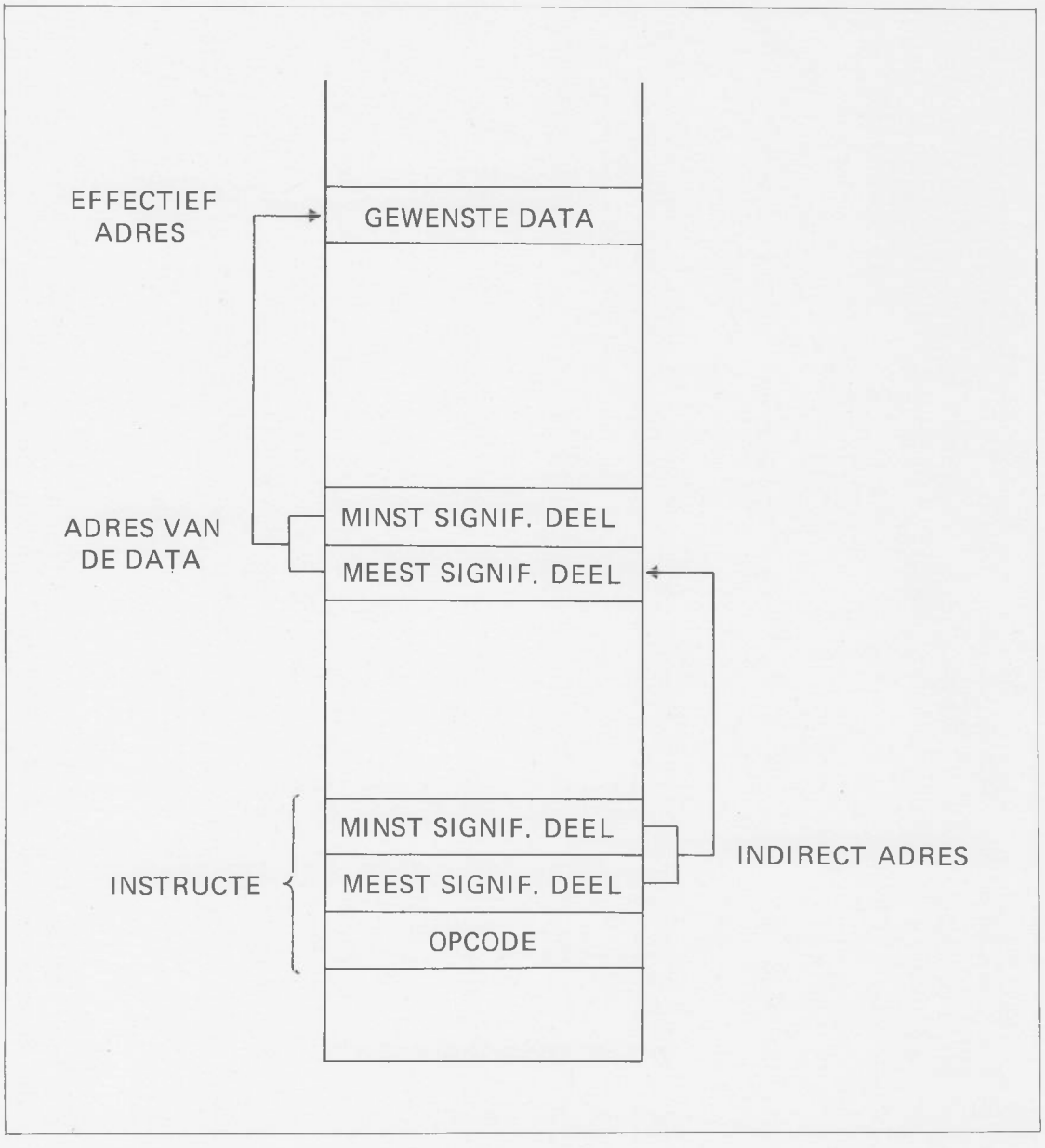


Fig. 47 Indirecte adressering.

Electronic components and materials

for professional, industrial and consumer uses

from the world-wide Philips Group of Companies

Argentina: FAPESA I.y.C., Av. Crovara 2550, Tablada, Prov. de BUENOS AIRES, Tel. 652-7438/7478.
Australia: PHILIPS INDUSTRIES HOLDINGS LTD., Elcoma Division, 67 Mars Road, LANE COVE, 2066, N.S.W., Tel. 427 08 88.
Austria: ÖSTERREICHISCHE PHILIPS BAUELEMENTE Industrie G.m.b.H., Triester Str. 64, A-1101 WIEN, Tel. 62 91 11.
Belgium: M.B.L.E., 80, rue des Deux Gares, B-1070 BRUXELLES, Tel. 523 00 00.
Brazil: IBRAPE, Caixa Postal 7383, Av. Brigadeiro Faria Lima, 1735 SAO PAULO, SP, Tel. (011) 211-2600.
Canada: PHILIPS ELECTRONICS LTD., Electron Devices Div., 601 Milner Ave., SCARBOROUGH, Ontario, M1B 1M8, Tel. 292-5161.
Chile: PHILIPS CHILENA S.A., Av. Santa Maria 0760, SANTIAGO, Tel. 39-40 01.
Colombia: SADAPE S.A., P.O. Box 9805, Calle 13, No. 51 + 39, BOGOTA D.E. 1., Tel. 600 600.
Denmark: MINIWATT A/S, Emdrupvej 115A, DK-2400 KØBENHAVN NV., Tel. (01) 69 16 22.
Finland: OY PHILIPS AB, Elcoma Division, Kaivokatu 8, SF-00100 HELSINKI 10, Tel. 1 72 71.
France: R.T.C. LA RADIOTECHNIQUE-COMPELEC, 130 Avenue Ledru Rollin, F-75540 PARIS 11, Tel. 355-44-99.
Germany: VALVO, UB Bauelemente der Philips G.m.b.H., Valvo Haus, Burchardstrasse 19, D-2 HAMBURG 1, Tel. (040) 3296-1.
Greece: PHILIPS S.A. HELLENIQUE, Elcoma Division, 52, Av. Syngrou, ATHENS, Tel. 915 311.
Hong Kong: PHILIPS HONG KONG LTD., Elcoma Div., 15/F Philips Ind. Bldg., 24-28 Kung Yip St., KWAI CHUNG, Tel. NT 24 51 21.
India: PEICO ELECTRONICS & ELECTRICALS LTD., Ramon House, 169 Backbay Reclamation, BOMBAY 400020, Tel. 295144.
Indonesia: P.T. PHILIPS-RALIN ELECTRONICS, Elcoma Division, 'Timah' Building, Jl. Jen. Gatot Subroto, P.O. Box 220, JAKARTA, Tel. 44 163.
Ireland: PHILIPS ELECTRICAL (IRELAND) LTD., Newstead, Clonskeagh, DUBLIN 14, Tel. 69 33 55.
Italy: PHILIPS S.p.A., Sezione Elcoma, Piazza IV Novembre 3, I-20124 MILANO, Tel. 2-6994.
Japan: NIHON PHILIPS CORP., Shuwa Shinagawa Bldg., 26-33 Takanawa 3-chome, Minato-ku, TOKYO (108), Tel. 448-5611.
(IC Products) SIGNETICS JAPAN, LTD, TOKYO, Tel. (03)230-1521.
Korea: PHILIPS ELECTRONICS (KOREA) LTD., Elcoma Div., Philips House, 260-199 Itaewon-dong, Yongsan-ku, C.P.O. Box 3680, SEOUL, Tel. 794-4202.
Malaysia: PHILIPS MALAYSIA SDN. BERHAD, Lot 2, Jalan 222, Section 14, Petaling Jaya, P.O.B. 2163, KUALA LUMPUR, Selangor, Tel. 77 44 11.
Mexico: ELECTRONICA S.A. de C.V., Varsovia No. 36, MEXICO 6, D.F., Tel. 533-11-80.
Netherlands: PHILIPS NEDERLAND B.V., Afd. Elonco, Boschdijk 525, 5600 PB EINDHOVEN, Tel. (040) 79 33 33.
New Zealand: PHILIPS ELECTRICAL IND. LTD., Elcoma Division, 2 Wagener Place, St. Lukes, AUCKLAND, Tel. 867 119.
Norway: NORSK A/S PHILIPS, Electronica, Sørkedalsveien 6, OSLO 3, Tel. 46 38 90.
Peru: CADESA, Rocca de Vergallo 247, LIMA 17, Tel. 62 85 99.
Philippines: PHILIPS INDUSTRIAL DEV. INC., 2246 Pasong Tamo, P.O. Box 911, Makati Comm. Centre, MAKATI-RIZAL 3116, Tel. 86-89-51 to 59.
Portugal: PHILIPS PORTUGESA S.A.R.L., Av. Eng. Duharte Pacheco 6, LISBOA 1, Tel. 68 31 21.
Singapore: PHILIPS PROJECT DEV. (Singapore) PTE LTD., Elcoma Div., P.O.B. 340, Toa Payoh CPO, Lorong 1, Toa Payoh, SINGAPORE 12, Tel. 53 88 11.
South Africa: EDAC (Pty.) Ltd., 3rd Floor Rainer House, Upper Railway Rd. & Ove St., New Doornfontein, JOHANNESBURG 2001, Tel. 614-2362/9.
Spain: COPRESA S.A., Balmes 22, BARCELONA 7, Tel. 301 63 12.
Sweden: A.B. ELCOMA, Lidingövägen 50, S-115 84 STOCKHOLM 27, Tel. 08/67 97 80.
Switzerland: PHILIPS A.G., Elcoma Dept., Allmendstrasse 140-142, CH-8027 ZÜRICH, Tel. 01/43 22 11.
Taiwan: PHILIPS TAIWAN LTD., 3rd Fl., San Min Building, 57-1, Chung Shan N. Rd, Section 2, P.O. Box 22978, TAIPEI, Tel. 5513101-5.
Thailand: PHILIPS ELECTRICAL CO. OF THAILAND LTD., 283 Silom Road, P.O. Box 961, BANGKOK, Tel. 233-6330-9.
Turkey: TÜRK PHILIPS TICARET A.S., EMET Department, Inonu Cad. No. 78-80, ISTANBUL, Tel. 43 59 10.
United Kingdom: MULLARD LTD., Mullard House, Torrington Place, LONDON WC1E 7HD, Tel. 01-580 6633.
United States: (Active devices & Materials) AMPEREX SALES CORP., Providence Pike, SLATERSVILLE, R.I. 02876, Tel. (401) 762-9000.
(Passive devices) MEPCO/ELECTRA INC., Columbia Rd., MORRISTOWN, N.J. 07960, Tel. (201) 539-2000.
(IC Products) SIGNETICS CORPORATION, 811 East Arques Avenue, SUNNYVALE, California 94086, Tel. (408) 739-7700.
Uruguay: LUZILECTRON S.A., Rondeau 1567, piso 5, MONTEVIDEO, Tel. 9 43 21.
Venezuela: IND. VENEZOLANAS PHILIPS S.A., Elcoma Dept., A. Ppal de los Ruices, Edif. Centro Colgate, CARACAS, Tel. 36 05 11.

A15

Printed in The Netherlands

9398 6083 0221