I N L E I D I N G C O M P U T E R A R C H I T E C T U U R

D E E L I

Sietse van der Meulen

RUU-CS-82-2
voorjaar 1982
(herzien 1983)

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–53 1454
The Netherlands

I N L E I D I N G C O M P U T E R A R C H I T E C T U U R

D E E L I

Sietse van der Meulen

Technical Report RUU-CS-82-2
voorjaar 1982
(herzien 1983)

Department of Computer Science
University of Utrecht
P.O. Box 80.002, 3508 TA Utrecht
the Netherlands

VOORWOORD

februari 1982
januari 1983

Deze INLEIDING COMPUTER-ARCHITECTUUR DEEL I bevat, tezamen met DEEL II de tentamineerbare stof van het gelijknamige basis-informatica college.

Het in deze syllabus beschreven hypothetische FANCY computer-systeem is ontworpen om er de meest gangbare karakteristieken van de hedendaagse (middel)grote systemen aan te demonstreren, in het bijzonder op het gebied van de adressering. Teneinde een zo realistisch mogelijk beeld te krijgen, hebben we de in micro-computers veel toegepaste MOTOROLA-68000 processor als uitgangspunt genomen voor de FANCY-CPU. Langs de CPU-interface is de MC68000 vrijwel volledig "upwards compatible" met onze FANCY - de belangrijkste toegevoegde "features" op processor nivo zijn:

- de FANCY heeft "memory-indirect-addressing",
- alle FANCY "control-instructions" zijn conditievolgend,
- de FANCY heeft (een afwijkende) floating-point aritmetiek,
- de FANCY heeft een "virtual memory option".

Zonder hierover specifiek te zijn veronderstellen we verder dat de FANCY een mogelijk fors uit de kluiten gewassen mini-computersysteem is, vergelijkbaar oa. met de (Digital) VAX11's.

De FANCY-assembler-taal FANLAN (volledig beschreven in een bij deze syllabus behorende FANLAN GUIDE) realiseert een vrij ver gaande stroomlijning van zowel de "operation-keys" als de formulering van de adresseringsfaciliteiten. FANLAN is volledig consistent in het gebruik van de notationele operatoren "#" "address of" en "@" "addressed by". Voor zover we hebben kunnen nagaan is FANLAN de eerste assemblertaal die dit zo doet. Ook de FANLAN blokstruktuur bevat een aantal nieuwigheden.

Bij het bedenken van de FANCY en FANLAN heb ik veel gehad aan de stimulerende kritiek en het nauwkeurig detailwerk van Wim Böhm, Rob Gerth, Jan-Hendrik Geels en Hans Jensen. Bij het uitwerken van dit DEEL I heeft Anne Brouwer krachtig geholpen.

Sietse van der Meulen

I N L E I D I N G C O M P U T E R A R C H I T E C T U U R

D E E L I

0. I N D E L I N G

0.0 COMPUTER-ARCHITECTUUR

De computer-architectuur houdt zich bezig met de externe aspecten van (grote en kleine) computersystemen en is dus overwegend fenomenologisch van aard. De accenten vallen op de globale werking van alle componenten, op de overdracht van informatie tussen de componenten, op de codering en interpretatie van informatie, op de organisatie en hiërarchie van geheugens en hun input en output, en bovenal op de logische structuur van het geheel.

Voor een goed begrip van al deze zaken hoeft men nauwelijks iets af te weten van electronica. De zg. hardware-technologie behoort strikt genomen tot een andere discipline (electronica). De wetenschap dat alle informatie kan worden voorgesteld met behulp van nullen en enen en dat deze nullen en enen in geheugens kunnen worden opgeslagen, "over lijnen kunnen lopen" en aan diverse operaties kunnen worden onderworpen, is in feite al voldoende. Het belangrijkste is, dat men weet dat (en hoe) deze nullen en enen op verschillend nivo verschillend kunnen worden gegroepeerd en geïnterpreteerd. Er zit echter iets onbevredigends in, om helemaal niets af te weten van wat zich achter de schermen afspeelt - vooral niet omdat juist daar alles blijkt terug te voeren tot enkele zeer eenvoudige principes (die ook eenvoudig zijn te begrijpen). Het is zelfs wel zo, dat een van de charmes van het onderwerp computer-architectuur berust op de bijna mathematische elegance van de grondprincipes. We zullen ze daarom (in hoofdstuk 1) als uitgangspunt nemen, zodat we verderop iedere keer als dat zinvol is, een kijkje achter de schermen kunnen nemen.

Dat "achter de schermen" is in de computer-architectuur, evenals trouwens in het hele vakgebied der informatica, zeer relatief - de "schermen" zijn nogal verschuifbaar (zie hoofdstuk 3). In plaats van "scherm" gebruiken we de (belangrijke) vakterm interface. Het gebied "achter" een interface wordt meestal hardware genoemd, het gebied ervóór software. De begrippen hardware en software worden ook wel absoluter gehanteerd (zie 0.1), maar we zullen zien dat deze absolutere benadering strikt genomen niet in strijd is met de modernere relatieve benadering. Waar misverstand zou kunnen ontstaan, zullen we het absolute begrip "hardware" soms electronica & apparatuur noemen.

0.1 COMPUTERSYSTEEM = HARDWARE + FIRMWARE + SOFTWARE

De electronica en de apparatuur vormen de hardware (in absolute zin) van een computersysteem: alles dat we aan een uitgeschakelde (dwz. niet-werkende) computer, met een al dan niet gewapend oog, kunnen zien - althans in principe. Al het andere (de niet-hardware) heet software. Een verandering in de software van een systeem kan dan ook alleen worden geconstateerd aan een werkende computer - de software is het geheel van alle programmatuur. We zullen zien dat deze definitie te grof èn te absoluut is, al is het een heel goede eerste benadering.

Ook op andere gebieden wordt tegenwoordig het onderscheid hardware/software gehanteerd. In de geluidsweergave bijvoorbeeld wordt de hardware gevormd door het geheel van opname- en weergave-apparatuur (microfoons, versterkers, pick-up met draaitafel, band- en cassette-recorders, speakers en boxen, akoestiese ruimten etc.). De software is datgene dat wordt opgenomen en weergegeven - de muziek dus. In de filmwereld gaat dit op nog grotere schaal op. Ook daar bestaat de hardware uit de opname- en weergave-apparatuur (camera's + projectors + alle geluids-hardware) inclusief de ontwikkel-apparatuur (voor het manipuleren van kleur en klank) etc. De software is de inhoud van de films - men rekent hieronder ook vaak de "geheugens": de filmrollen, geluidsbanden, video-banden etc. Een filmproducent wordt geacht een en ander te weten van de hardware, maar zijn eigenlijke terrein is de software. Geluidstechnici en cameramensen bewegen zich in het "tussengebied" tussen hardware en software - wij zouden kunnen zeggen dat zij zich bewegen "langs de interface".

In de informatica is dat "tussengebied" veel groter en gedifferentieerder dan in boven-aangehaalde voorbeelden. Dit tussengebied - dat bovendien op allerlei nivo's kan worden bestudeerd - is dan in feite de computer-architectuur. Van de informaticus wordt verwacht dat hij van de hardware-mogelijkheden en -onmogelijkheden iets afweet en begrijpt, en dat hij zelfs zeer goed op de hoogte is van het tussengebied (de interfaces op allerlei nivo).

We benaderen dit grensgebied nu vanuit de software en we kunnen daarin (stap-voor-stap in de richting van de hardware) onderscheiden:

a) gebruikers-programmatuur

De verzameling van alle programma's die door een willekeurige gebruiker (client) van een systeem in een of andere beschikbare programmeertaal zijn geschreven. Gebruikers-programmatuur is doorgaans van voorbijgaande waarde:

zodra het namelijk, als het enkele malen naar tevredenheid heeft gewerkt, ook van belang blijkt voor anderen (dus vaker wordt gebruikt en een meer permanente status krijgt), valt het al gauw in de volgende categorie:

b) applicatie-software

De applicatie-software is de in alle opzichten meest omvangrijke verzameling. Het omvat alle probleem-gerichte programmatuur die in een of andere vorm (geschreven of gedrukte tekst, sourcetext of vertaalde machinecode, op magneetband of enig ander medium) ter beschikking staat van specifieke gebruikersgroepen (die de machine dan meestal niet anders kennen dan via deze software: hún specifieke interface!). Voorbeelden van applicatie-software zijn: programmatheken (libraries, packages) voor numeriek-wiskundig-, technisch-, statistisch-, administratief-, commercieel-, strategisch-, etc.-gebruik (numerieke libraries, SPSS, database management systems), voor medische toepassingen (patienten-bewaking in ziekenhuizen, assistentie bij diagnose, pharmaceutische indicatie), diverse vormen van simulatie-programmatuur (van "computer-aided-instruction" tot "wereldmodellen") etc. Deze hele opsomming is op geen stukken na volledig, zelfs geen aanzet tot volledigheid. Een belangrijk aspect van alle applicatie-software is de maintenance (het "onderhoud"): wie is verantwoordelijk voor het goed functioneren van dergelijke software (correctie van fouten, aanpassing aan nieuwe omstandigheden en vooral: documentatie).

c) basis-software

De basis-software bestaat in eerste instantie uit het geheel van vertalers en operating systems die op een bepaalde machine-configuratie ter beschikking staan. Ook de zg. "utilities" (faciliteiten voor eenvoudige "jobs", zoals het via de lineprinter afdrukken van een magneetband, het plotten van een grafiek, het sorteren van allerlei informatie etc.) worden tot de basis-software gerekend. De grens tussen applicatie- en basis-software is niet scherp te trekken: over het algemeen wordt alle software ten dienste van, èn noodzakelijk voor álle gebruikers(groepen), tot de basis-software gerekend. De basis-software bepaalt mede de zg. software-interface van een computersysteem: het "gezicht" van het systeem zoals dat door alle gebruikers ervan wordt ervaren. De basis-software is in principe nog uitwisselbaar: vertalers kunnen al dan niet aanwezig zijn, een bepaalde hardware-configuratie kan onder verschillende operating systems worden bedreven.

d) firmware

Omvat alle componenten van de basis-software die volledig in een computersysteem zijn geïntegreerd en zonder welke de hardware praktisch onbruikbaar zou zijn. De firmware is dat deel van de basis-software dat niet uitwisselbaar is. Het woord "firmware" is een spits gevonden neologisme: "firm" ("stevig") is zachter dan "hard", maar harder dan "soft" - "ware" dus die ergens tussen "hard" en "soft" inhangt. De firmware is het geheel van al die programmatuur die meestal permanent ergens in een of ander geheugen aanwezig moet zijn - vaak in zg. "read-only" memory ("ROM"), soms zelfs "vast-bedraad" in de hardware - en waarvoor de computerfabrikant (de "Firm") evenzeer verantwoordelijk is als voor de hardware. Vanuit de meeste gezichtspunten wordt de firmware tot de hardware gerekend: "software" waar vrijwel iedereen af moet blijven.

Een ontwikkeling van de laatste jaren (micro-electronica, chips etc.) is, dat de firmware-funktie van de computerfabrikant verschuift naar gespecialiseerde "software-houses" - dit ziet men vooral op het terrein van de micro- t/m midi-systemen (voor deze terminologie zie 1.6). In principe maakt deze verantwoordelijkheids-verschuiving van fabrikant naar andere specialisten voor de client (individu, instelling of bedrijf) weinig uit: de "Firm" is alleen diffuser geworden.

De term "firmware" is minder algemeen ingeburgerd dan "software" en "hardware", wij zullen hem wel consequent gebruiken in de boven aangegeven zin.

e) hardware

Onder hardware (in absolute zin) verstaan we het geheel van electronica en apparatuur - oppervlakkig gezegd: alles waar technici aan te pas moeten komen als daar iets mis is. De vraag is dan natuurlijk: wat voor soort technici? We zullen ons hier niet begeven in moeizaam definieerwerk ("hardware" is een zeer complex gebied), maar constateren dat de ontwikkeling van de laatste tien jaren (microelectronica en "chips") voor het begrip hardware (en firmware en zelfs basis-software) een her-definitie noodzakelijk maakt. In hoofdstuk 3 gaan we hier dieper op in. We blijven overigens het begrip hardware hanteren als al datgene dat we als vastgegeven moeten accepteren.

We volstaan hier met de vaststelling dat ook het gebied van de hardware nog weer "gelaagd" is. Algemeen onderscheidt men:

e1) logica

De basis-elementen zijn hier de zg. "logical gates" ("logische poorten") die de fundamentele logische (Boolese) funkties realiseren (zie 1.2 en 1.3) en de hieraan weer ten grondslag liggende:

e2) electronica

De basis-elementen zijn hier diodes en transistors (zie 1.1). In feite behoort de electronica niet meer tot het gebied van de informatica - de logica van de hardware nog wel, maar zijn electronische realisering (die overigens buitengewoon gevarieerd kan zijn) is een andere technologie.

Naast logica en electronica vindt men in de hardware ook:

e3) de (rand-)apparatuur

Een rijk gevarieerde verzameling apparaten: input-apparatuur (kaart- en ponsbandlezers, toetsenborden, formulieren-lezers etc), output-apparatuur (regeldrukkers, plotters, "scopes", een grote diversiteit aan "printers" etc), geheugen-apparatuur ("drums", "disk-units", "tape-units", "cassette-recorders", "RAM"&"ROM"-chips etc), allerhande typen terminals etc.etc.

Het geheel overziend, is het gerechtvaardigd het terrein van de computer-architectuur te zien als een systeem waaraan zowel typische software- als firmware- als hardware-aspecten zitten. We zullen zien dat vooral een nauwkeurige vaststelling van de scheidingslijnen (de interfaces) van groot belang is voor een consistente wetenschappelijke behandeling.

0.2 INTERFACES

Op grond van de in 0.1 gegeven indeling, kunnen we een computersysteem nu langs verschillende interfaces bekijken. Vrij algemeen onderscheidt men:

ab) de users interface

Hierbij wordt meestal gedacht aan het computersysteem zoals zich dat voordoet aan de gebruiker die via bijv. een specifiek applicatie-pakket (bijv. SPSS, CAI, CAD, DBMS, ---) of via een specifieke vertaler (ASSEMBLER, BASIC, FORTRAN, PASCAL, COBOL, ALGOL, SIMULA, LISP, ADA, ---) met het systeem communiceert. Het is de interface van het systeem als (specifiek) bruikbaar stuk gereedschap: de computer zoals deze voor de client ter beschikking staat "met alles (wat nodig is) erop en eraan".

c) de operating system interface

Omdat vertalers gemakkelijker inwisselbaar zijn dan een operating system (waaronder ze altijd werken), noemt men de "basis software" interface meestal "operating system" interface. Bedoeld wordt: de computer voorzien van alle basis-software voor zover noodzakelijk om de machine te benutten op een wijze die in overeenstemming is met het doel waarvoor hij is gebouwd. In hoeverre men een (of meer) bepaalde compiler(s) hier al dan niet bij wil rekenen, is veelal een kwestie van smaak - over het algemeen hoort tenminste één assembler (zie hfdst. 3 & 6) er altijd wel bij. De compilers rekent men er vaak niet toe, omdat de compiler-bouwer vrijwel altijd werkt langs een bepaalde operating system interface.

d) de firmware interface

Het computersysteem zoals zich dat voordoet aan de bouwer(s) van een operating system. In het algemeen: de computer hardware voorzien van alle noodzakelijke firmware (zonder welke de hardware in feite onbruikbaar zou zijn voor iedere programmeur). Merk op, dat een (althans "minimale") assembler voor het genereren van machinecode, ook langs de firmware interface onontbeerlijk is (men moet ergens in kunnen programmeren).

e) de hardware interface

De uiterste interface waarlangs nog enige programmering (van firmware) mogelijk is. De computer waarop alleen nog een soort "ultimate codegenerator" aanwezig is (via welke men de noodzakelijke firmware kan "bootstrappen"). Voor ons is het kenmerk van de hardware interface het feit dat alle informatie is gereduceerd tot nullen en enen die op elementaire wijze worden gemanipuleerd.

De hier gegeven, vrij traditionele, indeling beschouwe men als een voorlopige. Een meer wetenschappelijke benadering van het interface-fenomeen geven we in hoofdstuk 3.

De moeilijkheid is, dat een meer genuanceerde en ook realistischer indeling afhankelijk kan worden van het beschouwde systeem. Het kan met name van belang zijn of men een eenvoudige micro, of een groot universeel systeem onder de loep neemt - een micro kan trouwens heel gecompliceerd zijn, en een groot systeem uiterst eenvoudig. Ook specifieke karakteristieken kunnen een belangrijke rol spelen - bijvoorbeeld of een systeem "microprogrammeerbaar" is of niet. We moeten dus van deze specifieke karakteristieken abstraheren, en daar wachten we nog even mee. De hierboven gegeven indeling treffen we trouwens vaak in de (commerciele/technische) litteratuur aan, zij het soms in een andere terminologie (een kwestie van woordgebruik).

Het belangrijkste voor ons is momenteel het inzicht dat een computer-systeem nooit een eenvoudig uniek "gezicht" heeft: men moet altijd nauwkeurig vaststellen vanuit welk gezichtspunt men er naar kijkt. De computer-architectuur is, in overeenstemming hiermee, een "gelaagde" discipline.

0.3 DE FANCY

Een inleiding tot computer-architectuur moet zich bij voorkeur niet vastleggen op één enkel systeem van één bepaalde fabrikant - men leert zo'n machine dan misschien wel goed kennen (met een aantal irrelevante rariteiten), maar raakt overeenkomstig gedesorienteerd (vaak zelfs bevooroordeeld) in andere systemen. Het alternatief - een overzicht van alle gangbare configuraties - is nog veel onaantrekkelijker: de onvermijdelijke encyclopedische opzet daarvan maakt de essenties onzichtbaar. Een heel belangrijke essentie is trouwens dat de meeste machines, ondanks vaak zeer grote hardware- en firmware-verschillen, steeds meer op elkaar gaan lijken naarmate men ze langs een hogere interface bekijkt - dat is uiteraard ook de bedoeling van interface structurering. Wij zullen min of meer het midden houden, door een soort "kleinste gemene veelvoud" te beschrijven van de interessantste low-level aspecten van gangbare machines.

Uitgangspunt hierbij is een bestaande moderne microcomputer: de MOTOROLA-68000, die we met een aantal "features" uitbreiden tot een zeer wel denkbaar (en ook vrij gemakkelijk realiseerbaar) computersysteem van de grote middenklasse. Voor de randapparatuur (transput en achtergrondgeheugens) hebben we onszelf geen restricties opgelegd: alles kan in principe worden aangesloten.

De aldus ontstaande computer-configuratie noemen we FANCY, om het fantasiekarakter ervan niet te verloochenen. Men doet er echter goed aan, er voortdurend vanuit te gaan dat het hart van dit systeem (de FANCY-CPU) zeer getrouw gebaseerd is op een moderne bestaande chip:

$$\text{MOTOROLA68000} \subset \text{FANCY}$$

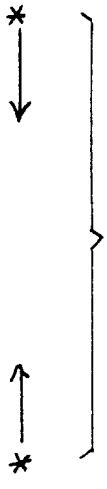
Van deze FANCY bespreken we (soms vrij gedetailleerd) een aantal interfaces, in volgorde van "hard" naar "zacht":

de electronica interface (hoofdstuk1)
de hardware interface (hoofdstukken 1, 2, 4 en DEEL II)
de firmware interface (DEEL II)
de assembly language interface (DEEL II)

De operating system interface en users interfaces komen "en passant" aan de orde in DEEL II.

1. H A R D E I N L E I D I N G

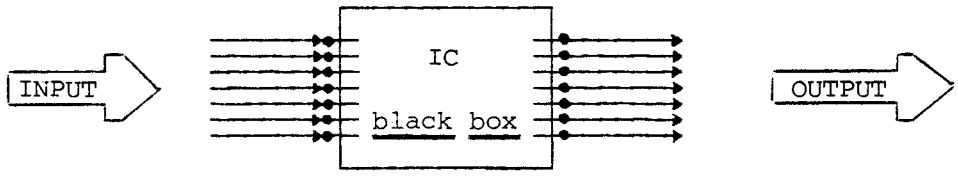
Dit hoofdstuk beoogt niet meer dan een schets van wat zich achter de "hardware-interface" van een moderne computer bevindt. Hoewel op college summier behandeld, zijn gedeelten van dit hoofdstuk alleen bedoeld om een keer door te lezen - deze onderdelen worden dus niet op het tentamen gevraagd. Ze zijn alsvolgt aangegeven:



is geen tentamen-stof

1.0 INTEGRATED CIRCUITS

De hedendaagse hardware-technologie berust geheel op de fascinerende mogelijkheden van IC's (Integrated Circuits of geïntegreerde schakelingen). Op één substraat, in omvang vaak nog kleiner dan een postzegel, zijn een groot aantal schakel-elementen samengevoegd tot één, soms zeer gecompliceerde logische operator van input-signalen naar output-signalen. We noemen zo'n IC wel een "black box", omdat we vaak helemaal niet hoeven te weten hoe hij werkt. Het is veelal voldoende te weten welke transformatie van input- naar output-signalen hij maakt (en in ongeveer hoeveel tijd):



Men onderscheidt SSI (Small Scale Integration) waarin enkele tot een stuk of tien elementaire schakelelementen tot één operator zijn samengevoegd, MSI (Medium Scale Integration) waarin globaal tien tot enkele honderden elementen zijn geïntegreerd en LSI (Large Scale Integration) waarin honderden tot zelfs tienduizenden (en meer) schakelaars tot één bijzonder machtige operator (soms zelfs een volledige processor) als het ware zijn samengesmolten. Er is alle reden te veronderstellen, dat de LSI-technologie nog maar pas is begonnen. We volstaan in dit hoofdstuk met een zeer summiere behandeling van enkele basisprincipes.

Een ook en vooral voor ons zeer belangwekkend aspect van de jongste ontwikkelingen in de LSI-technologie is, dat men (als individu, als groep of als instelling) in de niet meer zo ver verwijderde toekomst, een computer zal kunnen aanschaffen als een soort LEGO-bouwdoos. In feite kiest men dan niet meer een (kant en klare) computer, maar een aantal benodigde componenten die men zelf samenvoegt tot het gewenste systeem. Uiteraard zal men dan zo'n systeem zeer gemakkelijk kunnen uitbreiden en/of hergroeperen, al naar behoefte. Men zal, binnen zekere grenzen, de architect van zijn eigen computersysteem kunnen zijn (met mogelijk alle software-rampen van dien).

1.1 DE ELECTRONICA BASIS

De hele computer-technologie berust op zwakstroom-electronica. De gangbare spanningen liggen in de grootte-orde van 1 tot 10 V. Stroomsterkten worden gemeten in mA. De gebruikte weerstanden variëren meestal van enkele honderden Ω tot ettelijke MΩ. Capaciteiten en zelfinducties treden alleen parasitair op (geen functioneel gebruik dus).

In tegenstelling tot de audio-visuele electronica, spelen in principe slechts twee soorten stroomsterkten ("wèl of niet", eventueel "heen of weer") en twee spanningen (+ of 0, 0 of -, + of -) een rol. We noemen het ene signaal flip (aangeduid door 1) en het andere flop (aangeduid door 0).

Wanneer de hogere spanning (+ versus 0, of 0 versus -, of + versus -) is geassociëerd met flip (en de lagere dus met flop), spreekt men van positieve logica; in het andere geval van negatieve logica. Het hoge signaal wordt ook wel hi (high) en het lage signaal lo (low) genoemd. In de positieve logica is dus flip = hi en flop = lo; in de negatieve logica net andersom. In één en hetzelfde computersysteem kunnen verschillende IC's (in verschillende componenten) verschillende logica's hebben. De positieve logica is echter predominant en wordt in onze voorbeelden steeds gehanteerd.

In programmateksten zullen we flip altijd vereenzelvigen met *true* en flop met *false* (we vatten een signaal dus op als een *bool*-variable). We zullen zien dat de flip-flop logica sterk "duaal" van karakter is (d.w.z. systematische verwisseling van 1 en 0 is gemakkelijk in alle formules te hanteren). Onze keuze 1 = flip = *true* = hi en 0 = flop = *false* = lo is dus niet essentiëel, mits men eventueel maar consequent verwisselt. De conventie flip = 1 en flop = 0 is wèl algemeen aanvaard.

Het fundament van alle computer-hardware is het aan- en uitzetten van signalen. Een signaal flipt als het 1, en flopt als het 0 wordt. De basiscomponenten waarmee signalen worden geflipt of geflopt, heten schakelaars of schakel-elementen - ze worden gestuurd door (andere) flip- en flopsignalen.

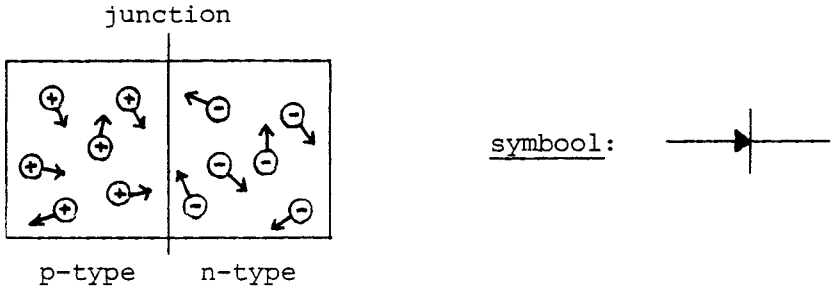
De allereerste schakelaars waren de mechanische relais, nog steeds in gebruik in ouderwetse telefooncentrales. In de veertiger jaren kwamen de eerste electronische schakelaars: vacuümbuizen. In de vijftiger jaren ging men over op transistor-schakelaars. In het eind van de vijftiger jaren werden de eerste SSIC's gemaakt: de technologie van de vroege zestiger jaren. In het eind van de zestiger jaren begon de MSI-technologie. In de afgelopen

zes jaar is de grote vlucht der LSIC's ("chips") begonnen. Het eind daarvan is nog niet in zicht.

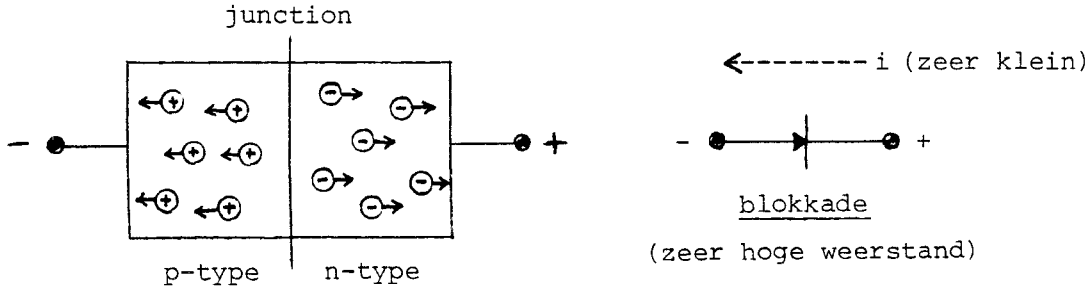
De totale moderne hardware-technologie berust op een a.h.w. uit de hemel gevallen (resp. in de hel uitgekookte) bijzondere eigenschap van Si-kristallen (het verhaal gaat ook op voor bijv. Ge=germanium). Silicium is 4-waardig: in een zuiver Si-kristal delen alle Si-kernen hun buitenste schil-electronen zodat een zeer stabiele configuratie met 8 gemeenschappelijke electronen per kern ontstaat (volledig bezette schil dus). Er zijn dan geen vrije electronen: een zuiver Si-kristal is een zeer slechte geleider.

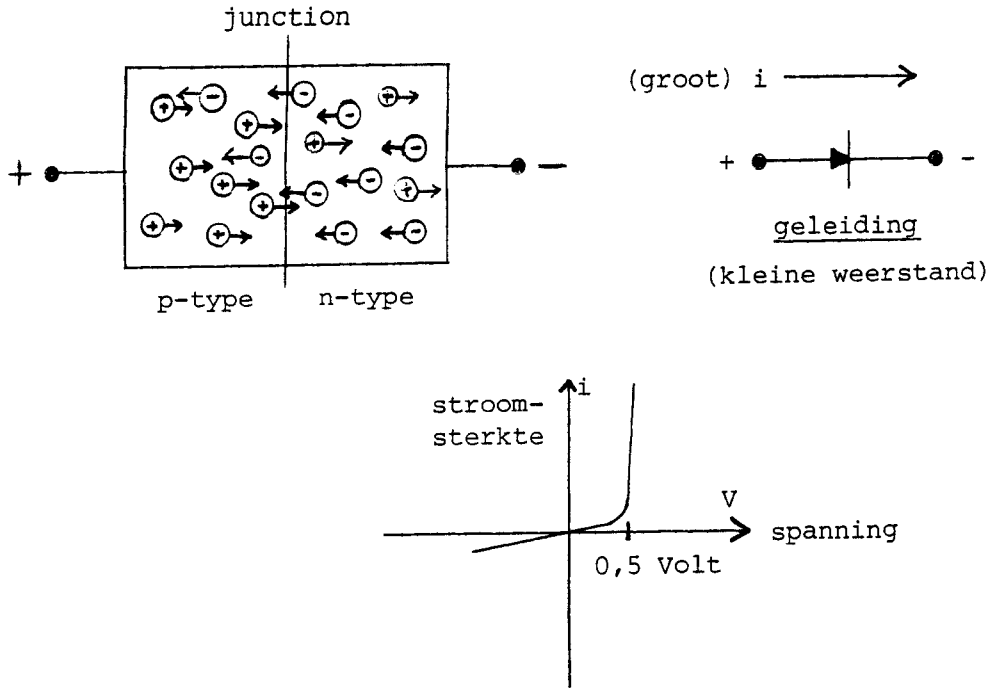
Zodra echter Si wordt "verontreinigd" ("doping" is de technische term) met een 5-waardig element (bijv. Ph=fosfor) of een 3-waardig element (bijv. B=boron), ontstaat een kristal met plekken waar resp. teveel (Ph-doping) of te weinig (B-doping) electronen zijn. Zowel de plekken met een electronen-surplus ("vrije electronen") als de plekken met een electronen-deficit ("positieve gaten") zijn vrij-bewegelijk, zodat een zg. halfgeleider (semiconductor) ontstaat.

Een halfgeleider met surplus-plekken (negatieve ladingsdragers) heet n-type semiconductor. Interessant is nu het scheidingsvlak (de junction) van een p-type en een n-type halfgeleider. We bekijken het eerst, zonder dat er een spanning wordt aangelegd:



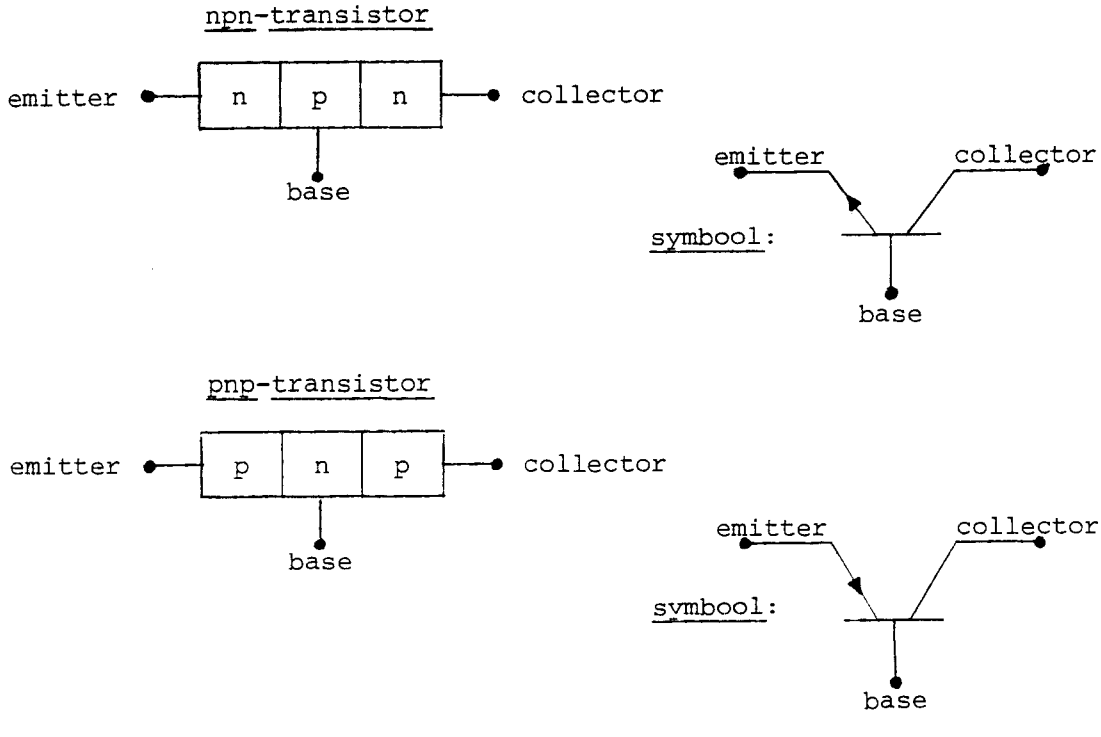
Hoewel er enige diffusie door de junction kan optreden, bewegen de positieve- resp. negatieve ladingsdragers zich "at random", elk in hun eigen omgeving. We kunnen nu op twee manieren een spanning aanleggen:





Een semiconductor-junction fungeert dus als een diode: blokkeert in de ene, geleidt in de andere richting.

Bijzonder belangrijk is de junction-transistor: een constructie met twee junctions en dus drie polen. Men kan zo'n transistor zien als twee "ruggelings" (resp. "frontaal") tegen elkaar geplaatste diodes. Er zijn dan ook twee soorten:

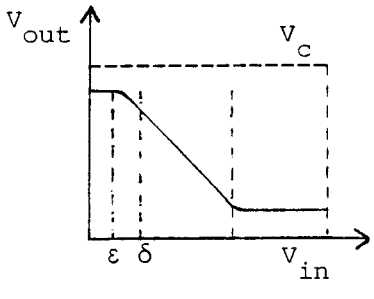


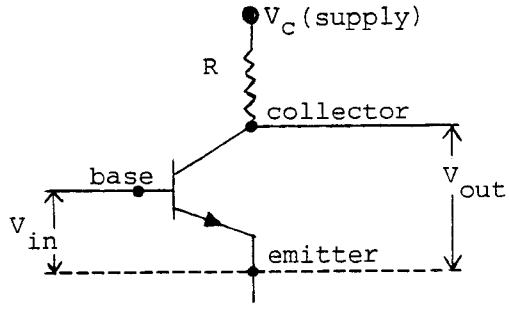
Een transistor gedraagt zich echter heel anders dan men in eerste instantie van twee tegen elkaar in geplaatste diodes zou verwachten. We bekijken een npn-transistor, aarden de emitter (V_e=0) en leggen een positieve spanning aan de collector (V_c=bijvoorbeeld 4.5 V). We kijken wat er gebeurt bij verschillende base-spanningen.

Zolang $V_b \leq V_e + \varepsilon$ ($\varepsilon \approx 0.5$ V), is de base/emitter diode geblokkeerd evenals de base/collector diode: er gebeurt dus niets (de schakelaar is uit). Zodra echter $V_b > V_e + \varepsilon$ gaat de base/emitter diode geleiden en dit heeft een verrassend effect op de base/collector junction. De ontstane stroom van base naar emitter (i_b) bestaat uit zowel positieve gaten naar de emitter, als vrije (negatieve) electronen naar de base. Zodra deze laatste in het base-substraat zijn aangeland, treffen ze aldaar een electrisch veld waarin de collector-pool veel sterker trekt dan de base-pool, immers: $V_c > V_b$. Er komt dus een electronen-stroom op gang van de emitter naar de collector -·conventioneel dus een stroom i_c van collector naar emitter (de schakelaar is aan).

In eerste benadering is $i_c = \beta i_b$; β wordt de stroomversterking genoemd. De meeste transistoren zijn zo gebouwd dat in het actieve gebied $V_e + \delta < V_b < V_c - \delta$ (met $\delta > \varepsilon$) de versterkingsfactor $\beta > 1$ is en constant (lineaire versterking).

Waar het actieve gebied van primair belang is in de audio-visuele versterkingstechniek, interesseren ons alleen de twee toestanden waarin een transistor niet-geleidend (uit) of wèl-geleidend (aan) is: de schakelaar-functie dus. Wanneer we $V_b > V_e + \varepsilon$ steeds laten toenemen, komt een moment waarop i_c verzadigd raakt: door de toenemende geleiding van collector naar emitter breekt de (aanvankelijk zeer grote) collector-emitter weerstand vrijwel geheel af, waardoor de collector-emitter spanning overeenkomstig daalt:





We zien dus:

als $V_b = V_{in} \approx V_e = 0$ dan is $V_{out} = V_c$

als $V_b = V_{in} \approx V_c$ dan is $V_{out} \approx V_e = 0$

In de positieve logica is nu V_e (= 0 V) = flop = 0 en V_c (= 4.5 V) = flip = 1. Het gegeven schakel-element werkt dus als een inverter:

$V_{in} = 0 \Rightarrow V_{out} = 1$	geen stroom:	schakelaar uit
$V_{in} = 1 \Rightarrow V_{out} = 0$	wel stroom:	schakelaar aan

In het vervolg bepalen we ons tot uitsluitend kwalitatieve beschouwingen en duiden we spanningen alleen nog aan met 0 resp. 1. De (kwantitatieve) waarden van spanningen, stroomsterkte en weerstanden laten we over aan hardware-technologen. Wij onderscheiden alleen nog flip- en flop-signalen en schakelaars die aan of uit staan.

De hier geschetste junction-transistor (ook wel bipolaire transistor) is slechts één type (dat ook nog weer in twee soorten, npn en pnp, bestaat). In de digitale schakel-technologie is over het algemeen alleen de schakelfunctie (verzadigings-gebied) van belang en niet de lineariteit van de versterking. Daardoor kunnen diverse andere typen transistors worden gebruikt. Een voorbeeld daarvan is de MOSFET (zie 1.5).

Waar het op aankomt is primair de zekerheid en de snelheid van de schakelfunctie. Voor specifieke toepassingen (bijv. pocket-calculators en random-access geheugens) kan men de snelheidseis wel wat afzwakken. Voor andere toepassingen (bijv. het rekenorgaan in een groot, voor multiprogrammering gebouwd, computersysteem) is juist de schakelsnelheid zeer belangrijk. Op het punt van zekerheid kan men eigenlijk nooit een compromis sluiten (helaas vaak het "geheim" van de àl te goedkope pocket-calculators).

1.2 DE LOGISCHE BASIS

De beschreven inverter is een voorbeeld van een zg. poort ("gate"): één of meer input-signalen definiëren één output-signaal. Omdat de signalen kunnen worden opgevat als variabelen A,B,C,--,X,Y,Z over de verzameling B = {flop,flip} = {0,1}, is een poort dus een functie g: $B^n \rightarrow B$ waarin n = 1,2,... Evenals de functies in algebra en analyse gedefiniëerd kunnen worden in termen van rekenkundige operaties (optellen, aftrekken, vermenigvuldigen en delen), zo kunnen de poorten worden gedefiniëerd in termen van operaties in een Boolese algebra:

X	$\bar{X}$
0	1
1	0

X	Y	X + Y	XY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Voor XY wordt ook wel X.Y of zelfs X×Y geschreven. Stilzwijgend wordt aangenomen dat de prioriteit van het Boolese product hoger is dan die van de Boolese som.

Zonder moeite herkennen we hierin de logische ALGOL-operatoren *not*, *or* en *and* (*not* is ¯, *or* is + en *and* is × of .):

```
op not = (bool x) bool: if x then false else true fi,
op or  = (bool x,y)bool: if x then true else y fi,
op and = (bool x,y)bool: if x then y else false fi;
```

Het loont de moeite de overeenkomst met de verzamelings-operaties complementering (′), vereniging (∪) en doorsnee (∩) na te gaan. Hiermee samenhangend wordt voor *not* ook wel ¬ of ~ geschreven, en voor *or* ∨ en voor *and* ∧. Men moet geen moeite hebben met de dialectvorming in de litteratuur, waarin allerlei notaties (soms ook raar door elkaar) worden gebruikt:

NOT , *not* , ¯ , ′ , ¬ , ~

OR , *or* , + , ∨ , ∪ , OG (= Or Gate)

AND , *and* , . , ∧ , ∩ , × , AG (= And Gate)

In de Boolese algebra zijn de volgende (gemakkelijk verifiëerbare en memoriseerbare) identiteiten van belang voor het begrijpen en vereenvoudigen van hardware-schakelingen:

commutativiteit:	$X+Y = Y+X$	$XY = YX$
associativiteit:	$X+(Y+Z) = (X+Y)+Z$	$X(YZ) = (XY)Z$
distributiviteit:	$X+YZ = (X+Y)(X+Z)$	$X(Y+Z) = XY+XZ$
flip en flop:	$0+X = X$ $\quad 1+X = 1$	$0.X = 0$ $\quad 1.X = X$
idempotentie:	$X+X = X$	$XX = X$
complementering:	$X+\bar{X} = 1$	$X\bar{X} = 0$
involutie:	$\bar{\bar{X}} = X$	
absorptie:	$X+XY = X$	$X(X+Y) = X$
	$X+\bar{X}Y = X+Y$	$X(\bar{X}+Y) = XY$
dualiteit: (de Morgan)	$\overline{(X+Y)} = \bar{X}.\bar{Y}$	$\overline{(X.Y)} = \bar{X}+\bar{Y}$

Het dualiteits-principe van de Morgan brengt tot uitdrukking dat bij complementering van de signalen (bijv. overgang van positieve- op negatieve logica of vice versa) de *and* en *or* functie van rol verwisselen. Ook het in duplo optreden van alle regels hangt hiermee samen; bijv. de "ongewone" distributiviteit $X+YZ = (X+Y)(X+Z)$ is de duale vorm van de "gewone" distributiviteit $X(Y+Z) = XY+XZ$, evenzo volgt $X(X+Y) = X$ uit $X+XY = X$.

Ten gevolge van de fundamentele dualiteit van de Boolese algebra zijn *and*- en *or*-poorten waarin het output-signaal is geïnverteerd, vaak nog belangrijker dan de "zuivere" *and*- en *or*-poorten:

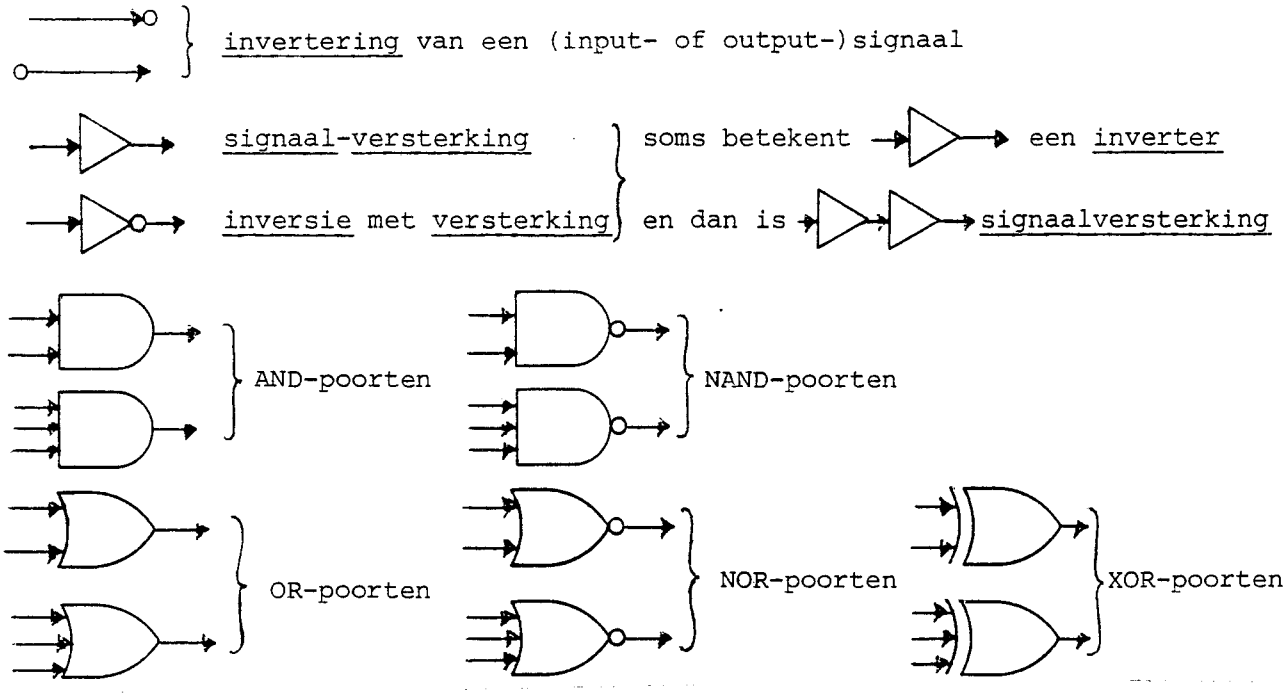
```
op nand = (bool x,y)bool: not(x and y),
op nor  = (bool x,y)bool: not(x or y);
```

Ook de zg. exclusive -or (of xor-) poort speelt in sommige schakelingen nogal eens een rol: een "or" dus die niet toelaat dat beide operanden 1 zijn:

```
op xor = (bool x,y)bool: (x and not y) or (not x and y);
```

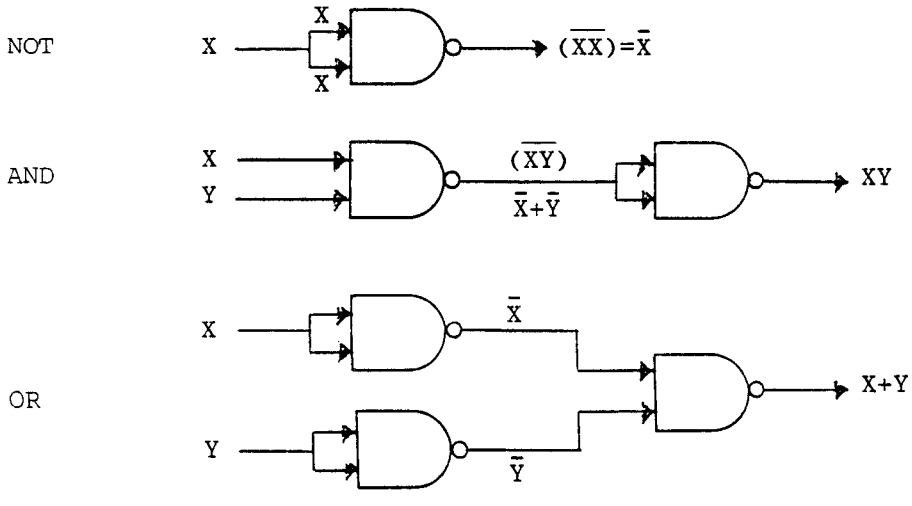
1.3 ELECTRONICA EN LOGICAL INTERFACE

In schakelschema's wordt algemeen het volgende symbolisme gebruikt (er zijn kleine locale varianten, vooral in de vormgeving):

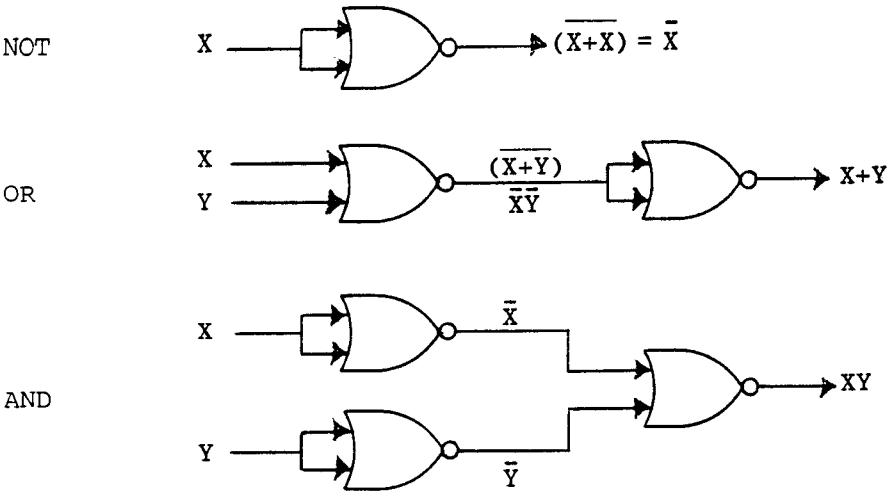


Het is op eenvoudige wijze in te zien hoe met alleen NAND-poorten, resp. NOR-poorten alle andere functies kunnen worden samengesteld:

NAND-logica:



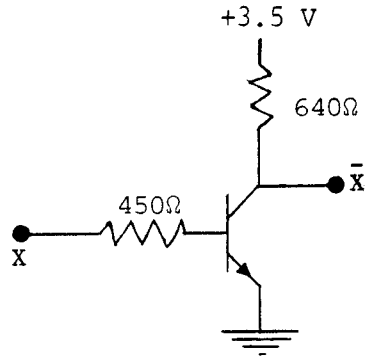
NOR-logica:

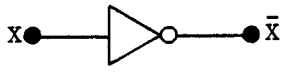


Merk in bovenstaande schakelschema's de dualiteit op, en ook hoe de OR uit NANDs en de AND uit NORs ontstaat (de Morgan!):

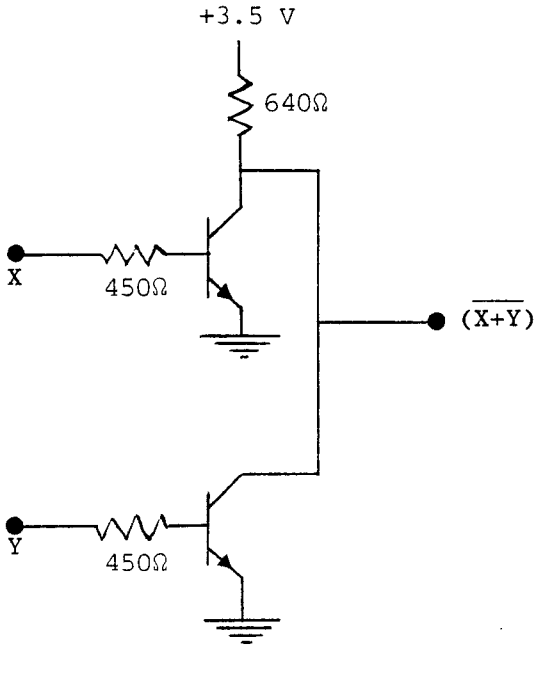
Er bestaat een zeer grote variëteit van meer of minder geïntegreerde schakelingen (met weerstanden, diodes en transistors) ter realisering van allerlei poorten. De bepalende factoren zijn snelheid, ontvankelijkheid voor ruis, mate van integratie (microminiaturisatie) en uiteraard de productiekosten. Men onderscheidt o.a. RTL (Resistor-Transistor-Logic), DTL (Diode-Transistor-Logic) en TTL (Transistor-Transistor-Logic). Ruw gezegd komt het hierop neer dat, naarmate de IC's meer schakelaars bevatten, de weerstanden moeilijker te realiseren zijn en bij voorkeur worden vervangen door diodes (die immers een grote weerstand hebben in één richting). Bij nog verdere integratie nemen transistors weer de functie over van diodes. Dit is echter maar één kant van het verhaal. Zorgvuldige afweging van de factoren snelheid, betrouwbaarheid en economie bepalen de andere kant. Bij wijze van voorbeeld geven we enkele veel gebruikte schakelingen:

een RTL inverter:

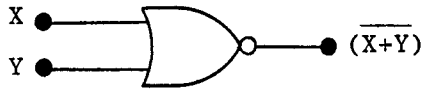




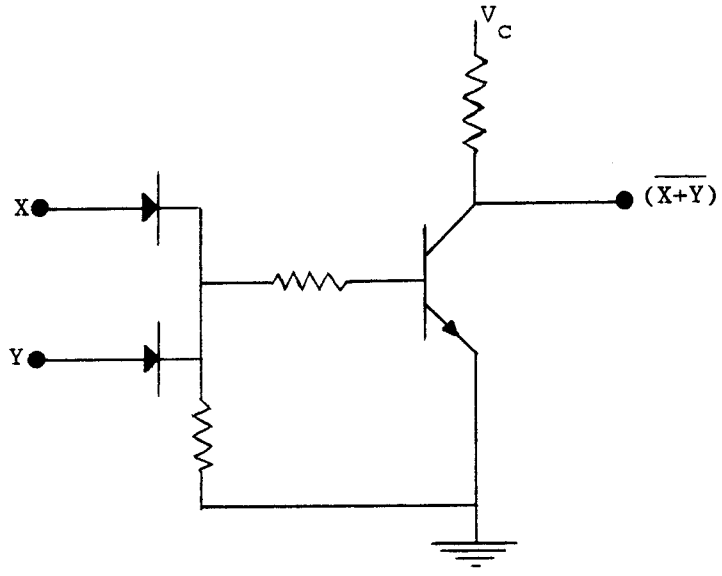
een RTL NOR-poort:

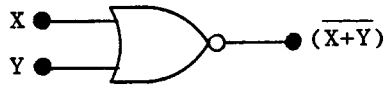


twee RTL-inverters in parallel vormen een NOR-poort



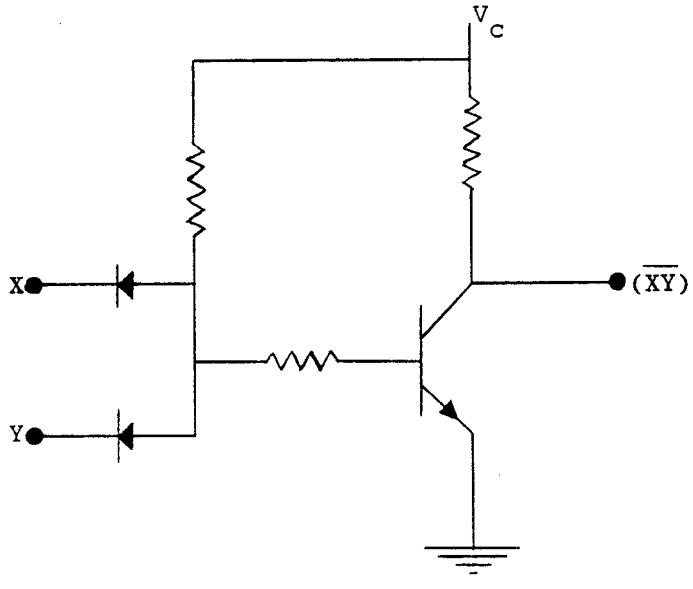
een DTL NOR-poort:

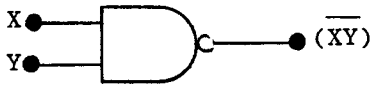




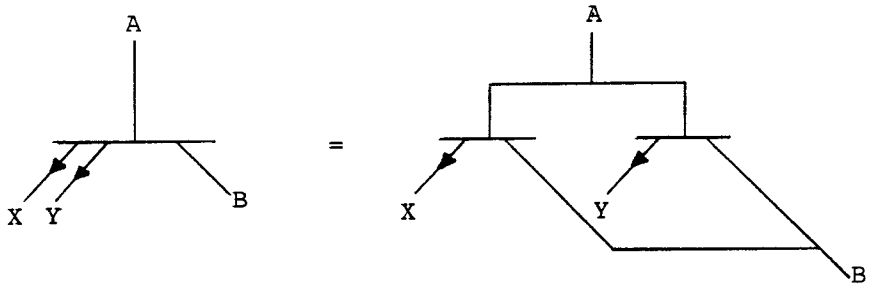
een DTL NAND-poort:

Merk de symmetrie op in deze DTL NOR- en NAND circuits.

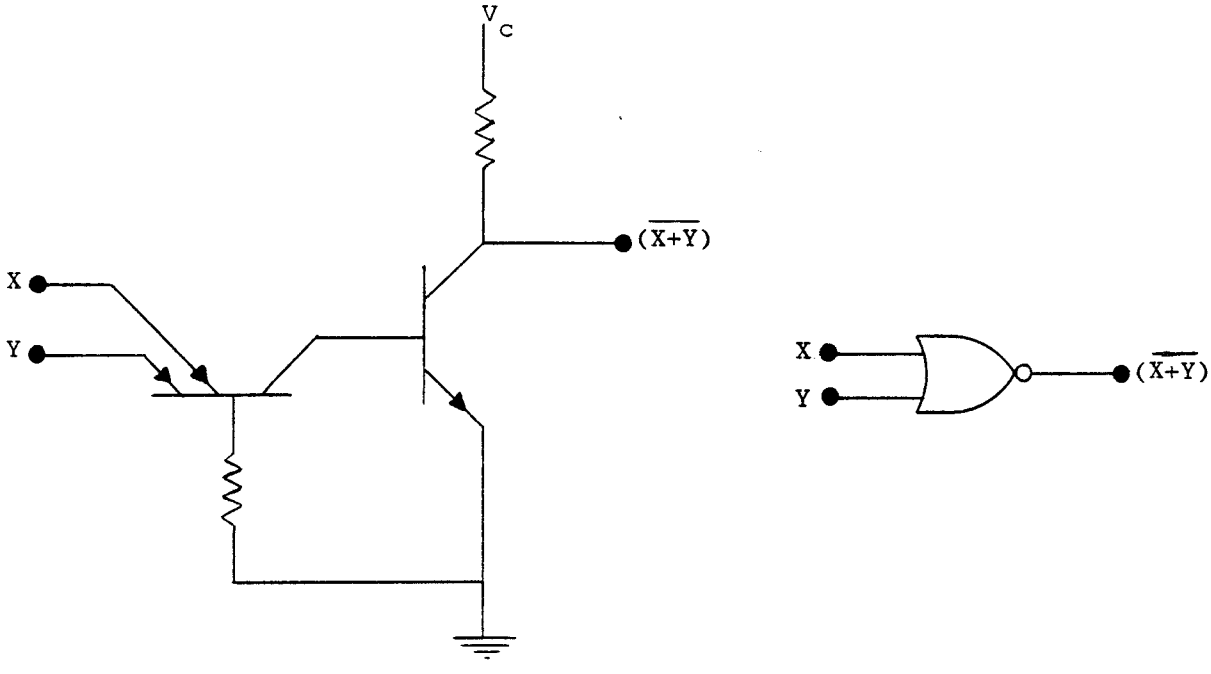




Een nadeel van DTL is de relatief grote schakeltijd (de relatieve langzaamheid dus) van diode-transistor connecties. Dit kan worden opgelost door de diode-functies te laten overnemen door een transistor met meervoudige emitters:



een TTL NOR-poort:



een TTL NAND-poort:

Merk ook hier de symmetrie op in de TTL NOR- en NAND circuits.

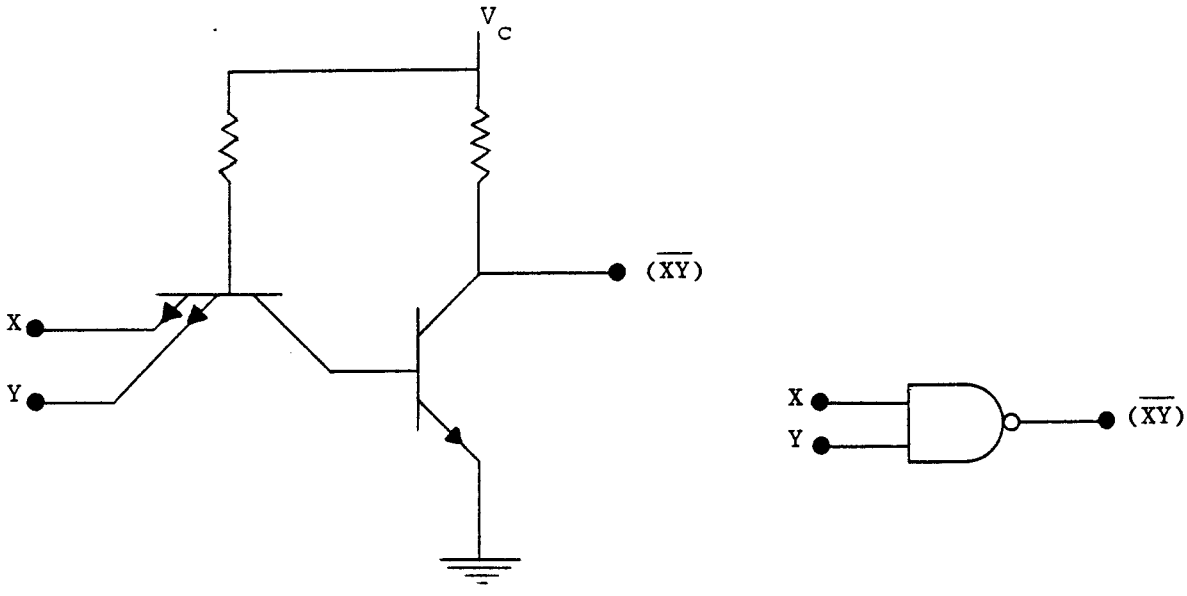


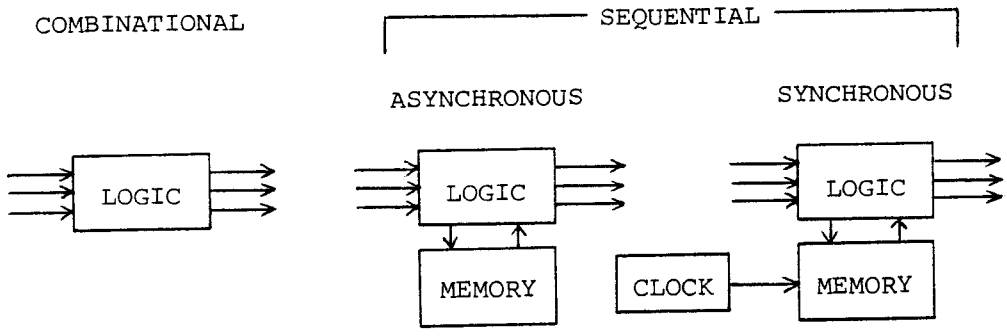
fig. 1/18

Vaak treft men allerlei combinaties van DTL en TTL aan: afweging van snelheid en productiekosten. De problematiek van RTL, DTL en TTL (er zijn ook nog vele andere schakelingen, óók met andere typen transistors) valt buiten het kader van de computer-architectuur. Voor ons vertegenwoordigen de elementaire poortfuncties de eigenlijke hardware. De electronica die daar achter zit behoort tot een ander soort technologie.

Toch is het wel nuttig (ongeveer) te weten hoe diverse poorten kunnen zijn opgebouwd uit transistors en eventueel diodes, en hoe een bepaalde hardware-logic (NOR, of NAND) eraan ten grondslag ligt. Vaak kan men bijvoorbeeld denken dat een bepaalde constructie "eenvoudiger" kan: de rechtvaardiging ligt dan meestal in de onderliggende electronica, waarin met name één transistor ook nog een meervoudige functie kan hebben (d.w.z. door verschillende poorten kan worden gedeeld).

1.4 SYNCHRONE LOGICA EN FLIPFLOPS

We onderscheiden drie soorten (hardware-)logica:



In de combinational-logic hangen de output-signalen alleen af van de input-signalen. De eventuele voorgeschiedenis speelt hier dus geen enkele rol. Poorten zijn typische combinational-logic componenten.

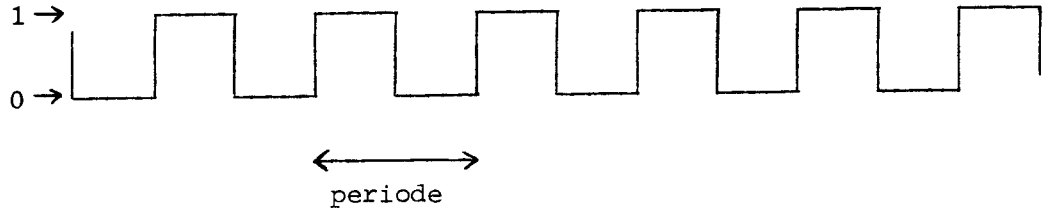
In sequential-logic spelen naast poorten ook geheugen-elementen een rol. De voorgeschiedenis is hier mede bepalend voor de output-signalen.

Wanneer in sequentiële logica geheugen-elementen op in principe ieder tijdstip hun inhoud kunnen veranderen, hebben we te maken met asynchronous sequential-logic. De volgorde der gebeurtenissen binnen een asynchrone IC (resp. in een keten van aan elkaar gekoppelde asynchrone IC's) is ongedefiniëerd.

In synchronous sequential-logic staat de volgorde der gebeurtenissen strikt onder regie van een klok die nauwkeurig bepaalt op welk moment welk geheugen-element wordt "beschreven" (een nieuwe inhoud krijgt), dan wel wordt "uitgelezen" (zijn bijdrage levert tot de gang van zaken).

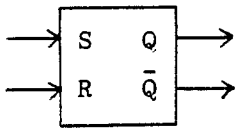
In combinational- en asynchronous sequential-logic is de tijd die verloopt tussen input en output van een IC (resp. keten van IC's) uitsluitend bepaald door de schakelsnelheden van de betrokken diodes en transistors (en mogelijk zelfs nog door de lengte van de geleidingen). In de synchrone sequentiële logica is deze tijd altijd nauwkeurig een veelvoud van de klokperiode. Synchrone logica is predominant, hoewel asynchrone componenten vaak wel worden toegepast. Wij bepalen ons verder tot de synchrone logica.

De CLOCK (meestal gerealiseerd door een kwartskristal) is een uiterst regelmatige blokspanning:



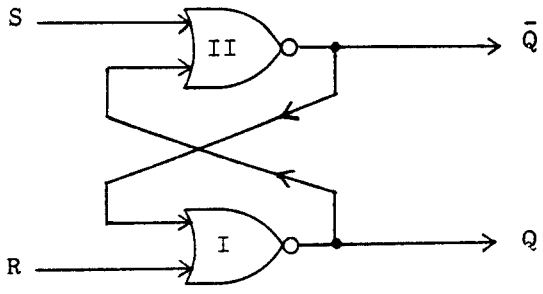
In feite is een CLOCK niets anders dan een hoogfrequente wisselspanning. Sterk afhankelijk van het soort hardware en de toepassing ervan, ligt de frequentie tegenwoordig meestal ergens tussen 1 en 200 MHz, corresponderend met een periode van 1 µsec tot 5 nsec. De allersnelste transistors (die bedreven worden beneden de verzadigingsstroom: "non-saturating-logic") schakelen in ongeveer 1 nsec en verdragen dus een klok tot 1000 MHz). In die tijd is de stroomweg in een halfgeleider in de orde van grootte van 1 dm. De nanoseconde en de corresponderende klokperiode van 1000 MHz markeren ongeveer de grens van het physisch mogelijke (de marge kan nauwelijks groter zijn dan nog eens een factor 10).

Het elementaire geheugen-element bewaart een flip of een flop (een 1 of een 0 dus). Wanneer zo'n geheugencel in transistors is uitgevoerd, en dus een onderdeel kan zijn van een IC, is de gebruikelijke naam: "flipflop". Het basissymbool voor de flipflop is:



Een flipflop heeft slechts schijnbaar twee outputs: Q en $\bar{Q}$ zijn elkaars inverse. De output is òf flip: $Q = 1$, $\bar{Q} = 0$, òf flop: $Q = 0$, $\bar{Q} = 1$.

Het loont ruimschoots de moeite om de werking van een flipflop een keer goed te analyseren. Eerst kijken we hoe een (zeer eenvoudige) flipflop uit twee teruggekoppelde NOR-gates kan zijn opgebouwd:



Ieder vak heeft z'n rariteiten: in het basis symbool voor de flipflop zet men de Q altijd boven en de $\bar{Q}$ onder: hier moet het net andersom. Ga na waarom!

Neem nu S = R = 0 en stel dat (ten gevolge van de voorgeschiedenis) Q = 1. Deze Q uit I is teruggekoppeld op de ingang van II, zodat dus (!) de output van NOR-gate II een 0 is. Deze is teruggekoppeld op I, waarvan beide inputs 0 zijn en dus de output (*0 nor 0 = 1*) blijft 1. Anders gezegd: de toestand Q = 1, $\bar{Q}$ = 0 is stabiel. We zeggen: de flipflop "bewaart" een 1. Uit symmetrie-overwegingen is direct duidelijk dat óók de toestand Q = 0, $\bar{Q}$ = 1 stabiel is, m.a.w.: de flipflop kan ook een 0 bewaren.

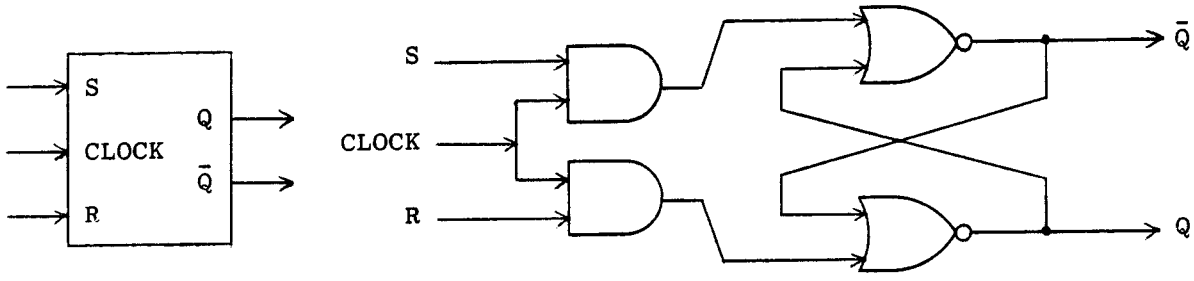
Een flipflop is dus een geheugen-element voor het onthouden van een 1 of een 0. Men noemt een "1 of 0" een bit (binary digit). Een goed Nederlands woord zou zijn "twijfer" (tweetallig cijfer).

Omdat een flipflop in twee stabiele toestanden kan verkeren, wordt hij ook wel een "bistable" genoemd. Het woord "bistable" wordt ook gebruikt voor geheugen-elementen die op andere wijze zijn gerealiseerd, zoals ferrietkernen (zie verderop).

Het is gemakkelijk in te zien dat een signaal S = 1 (R = 0) de flipflop op 1 zet (Q = 1, $\bar{Q}$ = 0), en dat een signaal R = 1 (S = 0) de flipflop op 0 zet. S betekent "Set", R betekent "Reset". Merk op dat het effect van S = 1 èn R = 1 ongedefiniëerd is. De IC waarin zo'n flipflop is opgenomen, moet dit dus uitsluiten. Meestal zijn de S en R van de ene flipflop de Q en $\bar{Q}$ van andere (niet noodzakelijk beide van dezelfde flipflop).

De hier behandelde eenvoudigste flipflop (zie boven) schakelt in principe zodra een 1 op S of R arriveert. Uiteraard verloopt er enige tijd tussen het "arriveren" van een 1 op de input en het "verschijnen" van een 1 op de output. Deze tijd heet schakeltijd van de flipflop.

In de synchrone logica is het ongewenst dat een flipflop schakelt zodra een 1 op S of R arriveert: hier wil men het schakelmoment precies bepalen op de klok. Daarvoor hebben we de zg. geklokte flipflop:



Een geklokte flipflop wordt altijd zó gebouwd dat hij op de "edge" van het klok-signaal schakelt, d.w.z. òf als de CLOCK van 0 naar 1 gaat ("flipt"), òf als hij van 1 naar 0 gaat ("flopt"). De bovenaangeduide schakeling is een flipflop die op een "flippende" klok schakelt. Het schakelen gebeurt uiteraard alleen als S = 1 òf R = 1. Merk op dat de flipflop zijn laatste waarde behoudt, ongeacht het klok-signaal, zolang S = R = 0.

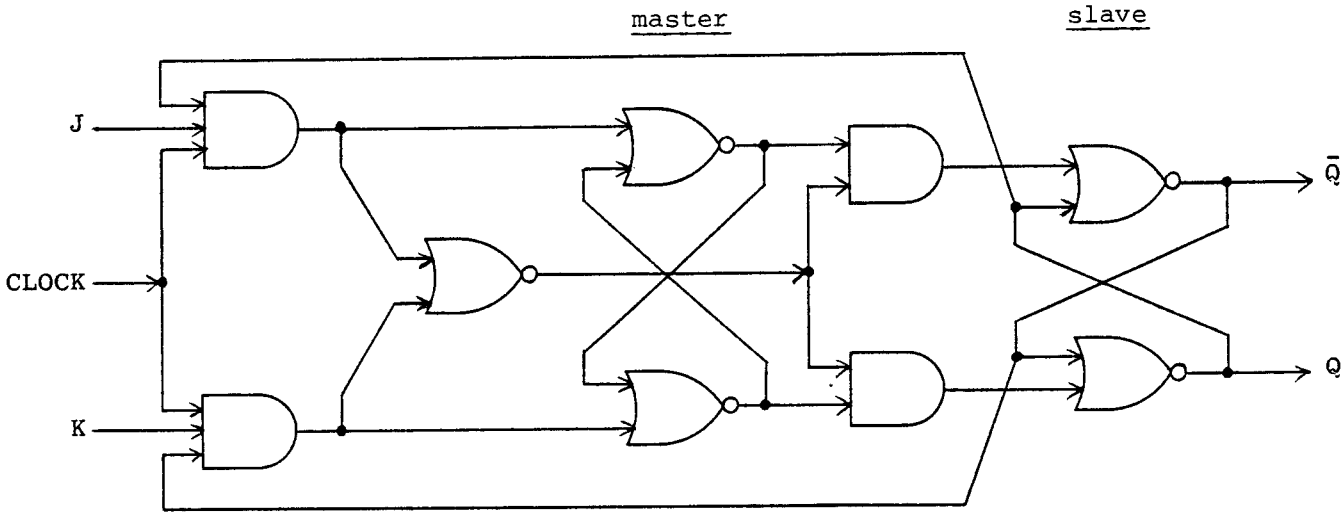
Op twee opeenvolgende klok-pulsen, ten tijde $t = t_n$ en $t = t_{n+1}$, is de flipflop-functie gegeven door onderstaande tabel, waarin q_n de flipflop-output ten tijde $t = t_n$ is, en uiteraard $\bar{q}_n = \underline{not}\ q_n$

t_n		t_{n+1}	
S	R	Q	$\bar{Q}$
0	0	q_n	$\bar{q}_n$
0	1	0	1
1	0	1	0
1	1	?	

Merk op hoe een geklokte flipflop over langere tijd (eventueel een zeer groot aantal klok perioden, tot in de milliarden) een "voorgeschiedenis" kan vasthouden.

Wanneer in de synchrone logica meerdere flipflops in serie zijn gekoppeld (en dat is nogal eens het geval), moet worden voorkomen dat een signaal op één klokpuls ongecontroleerd een eind "door de keten schiet". Dit zou immers een asynchroon proces worden. Een signaal hoort a.h.w. met-de-klokpuls-mee zich voort te planten. Een veel gebruikte methode om dit te bereiken is de zg. master-slave schakeling waarin twee gewone flipflops ("master" en

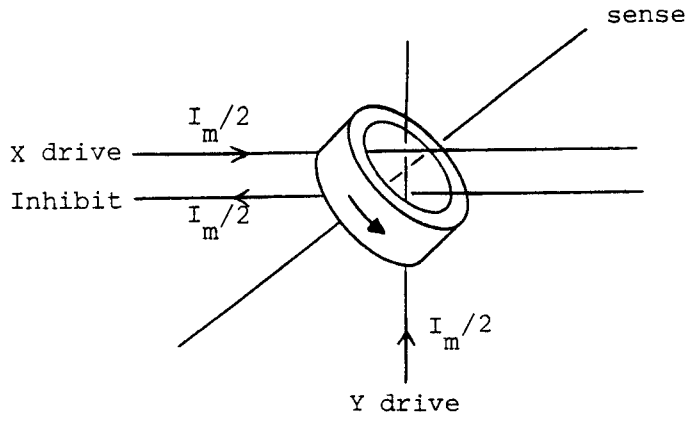
"slave") zó zijn gekoppeld dat de tweede (de "slave") pas schakelt als de klokpuls flopt; dus nádat de eerste (de "master") schakelde toen de klokpuls flipte. Wie zin heeft, kan dit nagaan in onderstaande schakeling. Om aan te geven dat men met een master/slave schakeling te maken heeft, schrijft men J voor S en K voor R:



Hoewel een elementair element, is een flipflop duidelijk een allesbehalve eenvoudige constructie. Zelfs voor de simpelste geklokte flipflop (pag terug) komt men niet met minder dan 4 transistors uit (en dat alleen bij RTL), voor een master-slave schakeling zijn er gauw een stuk of 20 (in TTL) nodig.

Tot voor kort waren flipflops dan ook kostbare geheugencellen. De LSI-technologie brengt hier nu snel verandering in. Flipflops zijn vooral aantrekkelijk door de snelheid waarmee ze schakelen (5 tot 100 νsec). Voor omvangrijke geheugens neemt men meestal (althans anno 1979) genoegen met langzamere "bistables". Een nu snel verouderende technologie is de ferrietkern: een (zeer klein) ringetje van ferriet dat "linksom" of "rechtsom" gemagnetiseerd kan zijn.

Voor het schakelen van zo'n kerntje is een zekere minimale stroomsterkte I_m nodig. Door nu twee tot vier draden (geïsoleerd) door zo'n kern te voeren kan men bepaalde kernen selecteren en "uitlezen" resp. "beschrijven". Iedere draad voert dan een stroom van (iets meer dan) $I_m/2$, op zichzelf dus onvoldoende voor een omslag van de kern:



Een (ferriet-)kerngeheugen kan bestaan uit honderden "matrices" van elk enkele duizenden kernen. De schakeltijden (lees- en schrijfcyclus) bedragen minimaal 500 νsec, meestal iets in de orde van 1 à 2 μsec.

Voor massageheugens (miljoenen en meer bits) gebruikt men gemagnetiseerde plekken op daartoe geëigend materiaal (magneetband of -schijf). De schakeltijden zijn in de orde van veelal ettelijke microseconden, maar de in magnetisme geschreven bits zijn bovendien moeilijker bereikbaar op hun drager (band of schijf): positionering en rotatiesnelheid spelen een belangrijke rol.

Voor het bewaren van vaste gegevens (constante data, maar vooral ook essentiële stukken firmware) heeft men geen bistables nodig, maar kan worden volstaan met "monostable"s - elementen die òf een flip òf een flop vasthouden, maar niet omklapbaar hoeven te zijn (bij voorkeur zelfs niet). Hiervoor bestaan diverse schakelingen en media.

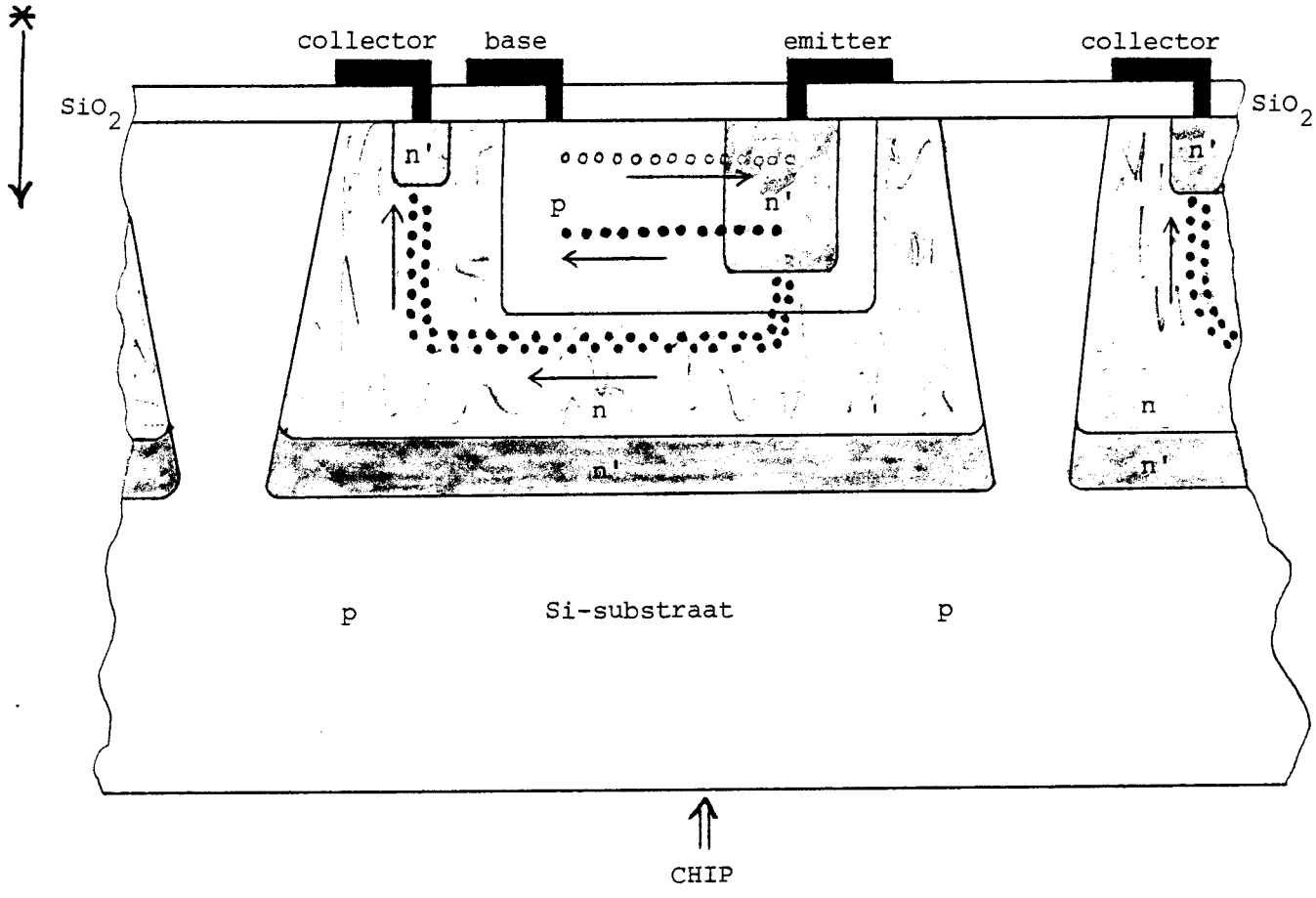
1.5 LSI EN CHIPS

In alle computersystemen vormen de twee spanningen flip en flop de objecten die gemanipuleerd worden. Weerstanden, diodes en transistors zou men kunnen beschouwen als de "subatomaire" componenten die deze objecten manipuleren in de hiërarchie van de computer hardware.

De eigenlijke atomaire componenten zijn de poorten (NOT, AND, OR, NAND, NOR, XOR etc.). De vele vormen van (geklokte) flipflops kan men dan beschouwen als basis-moleculen. Flipflops kunnen op velerlei wijze aan elkaar gekoppeld zijn. Een lineaire keten van flipflops voor het bewaren van een rij bits wordt meestal register genoemd. Andere ketens, verbonden via diverse poortschakelingen, vormen verschillende operationele eenheden (shift-register, "adder", "multiplier" etc.). Deze keten-, boom- en ringvormingen van flipflops en poorten vertoont wel enige analogie met koolstofketens in de organische chemie, al is de variëteit van vormen in de verste verte niet daarmee vergelijkbaar.

Door de MSIC- en LSIC technologie krijgen de grotere eenheden inderdaad wèl een soort moleculaire realiteit. In het bijzonder in LSI kunnen de basiscomponenten uit vele tot zeer vele flipflop-ketens (registers) en functionele schakelingen (operatoren) bestaan. De revolutionaire en in z'n gevolgen zelfs dramatische IC-technologie valt buiten het bestek van dit verhaal. We volstaan met een summiere en vooral globale schets.

Een chip (Engels voor "schijfje" of "fiche") is niets anders dan een zorgvuldig gefabriceerd substraat van p-type (of n-type) Si-kristal waarin met een zeer geraffineerde techniek "eilandjes" van meer of minder zwaar n-type (p-type) Si-kristal a.h.w. zijn "geïnjecteerd". De afbeelding hiernaast geeft een (niet meer dan schematische) indruk van een zeer klein fragment van een chip. De verhoudingen zijn uiteraard geheel zoek. Het enige dat het plaatje beoogt, is aan te geven hoe verschillende lagen en eilandjes van n- of p-type Si-kristal als transistor kunnen gaan fungeren, mits juist geschakeld.



SiO_2 (silicium-dioxide): een zeer goede isolator

p = p-type Si-kristal (lichte doping met bijv. Boron)

n = n-type Si-kristal (lichte doping met bijv. Phosfor)

n' = n-type Si-kristal (zware doping met bijv. Phosfor)

aluminium-geleider

positieve "gaten"

(negatieve) electronen

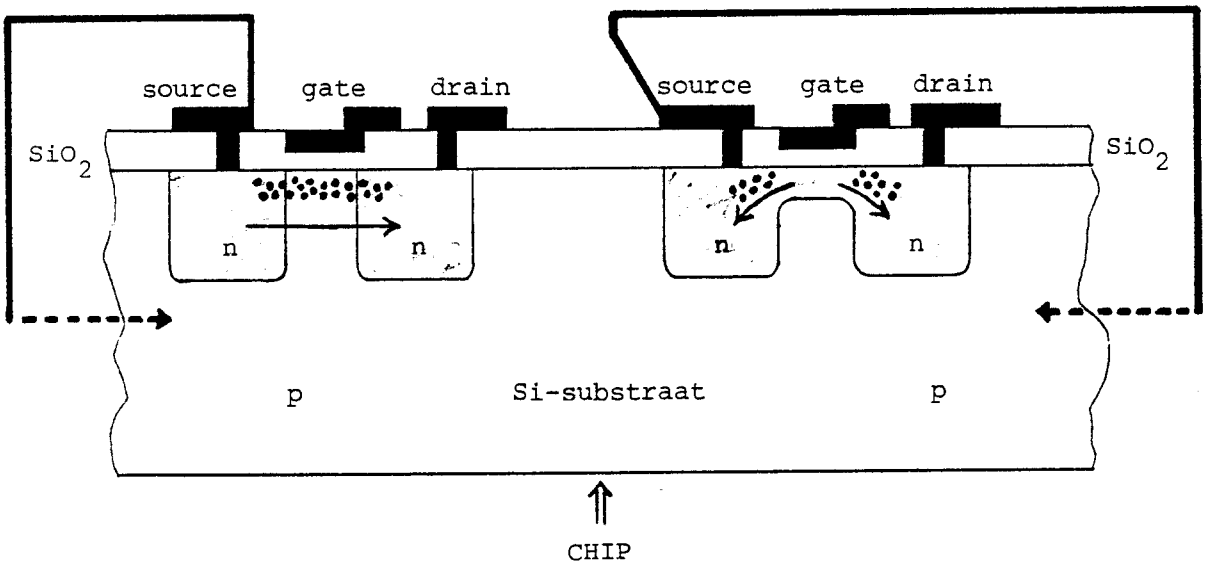
} merk op dat de collector-emitter stroom aanzienlijk groter is dan de base-emitter stroom.

Op één chip kan men tegenwoordig duizenden tot zelfs tienduizenden transistors (zowel npn and pnp) en diodes tot stand brengen. Uiteraard is de functie van zo'n chip geheel bepaald door de koppeling van deze basis-elementen.

Een chip wordt ontworpen door deze koppelingen zéér nauwkeurig in te tekenen op speciaal tekenpapier van ettelijke vierkante meters. Deze worden dan fotografisch verkleind tot de afmeting van de chip (ca. 1 vierkante centimeter). Zowel het injecteren van de eilandjes in het substraat, als het opdampen der aluminiumgeleidingen voor de onderlinge koppelingen komt

tot stand door laags-gewijze opbouw via zeer zorgvuldige chemische micro-processen.

De productie van bipolaire transistors (d.w.z. npn- of pnp-transistors) op een chip is kostbaar. Een eenvoudiger type transistor schakelt op een aan de "base" geïnduceerd electrisch veld. De base heet dan "gate" (vooral niet te verwarren met een gate- of poort-schakeling), de collector en emitter worden resp. drain en source genoemd. De source is verbonden met het substraat:



Een dergelijke transistor heet MOSFET: Metal-Oxyde-Semiconductor Field-Effect-Transistor. MOSFETs schakelen veel langzamer dan bipolaire transistors (het kan een factor 10 tot 100 schelen), maar door hun veel eenvoudiger struktuur lenen ze zich aanzienlijk gemakkelijker voor massaproductie.

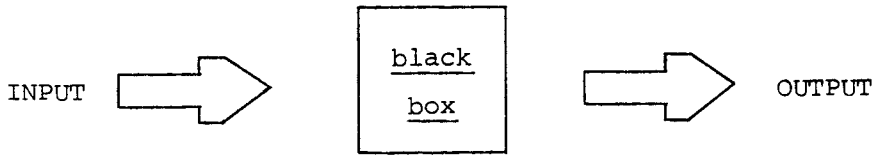
MOSFETs hebben vooral een grote vlucht genomen in chip-geheugens - de zg. MOS-geheugens (2^{14} = 16.384 bits op één chip begint normaal te worden). MOSFETs worden ook op grote schaal toegepast in (al dan niet programmeerbare) pocket-calculators.

Het is moeilijk te voorspellen op welke typen transistor de technologie van de large-scale-integration zich zal stabiliseren. Nieuwe typen transistor en integratie ("super-transistors": een gecompliceerde poortschakeling in één element) zijn volstrekt niet ondenkbaar. Momenteel ontwikkelt de

technologie zich primair in de richting van steeds grotere geheugens op één chip (MOSFET-technologie) en steeds snellere registers en operatoren (bipolaire transistors).

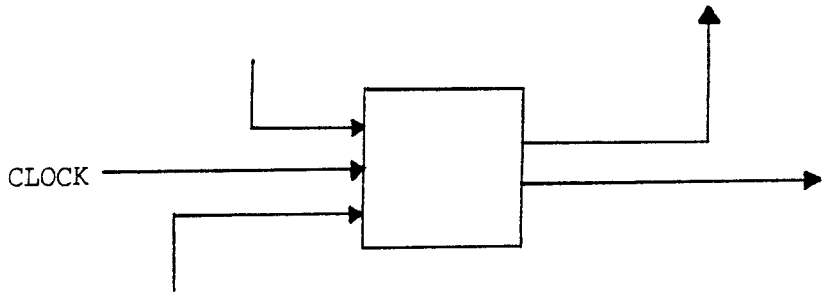
Voor ons is vooral de hiërarchie in de struktuur van moderne hardware van belang. Het is een gelukkig toeval dat de integratie van basiselementen tot grotere eenheden op verschillende nivo's (zoals we die in de computer-architectuur graag hanteren), in grote lijnen samenvalt met de integratie in de IC-technologie. Wellicht is dat ook helemaal geen toeval: er is uiteraard veel beïnvloeding over en weer. Toch doet men er verstandig aan, de hiërarchische eenheden ("black boxes") zoals die in de computer-architectuur worden gehanteerd, niet al te zeer te vereenzelvigen met hardware-ICs.

In principe onderscheiden we slechts twee soorten black boxes: zonder geheugen (combinational logic) en met geheugen (sequential logic). Uit de context is altijd wel duidelijk met welke van de twee we te doen hebben, zodat we op vrijwel elk nivo werken met het volgende beeld van een logische eenheid:



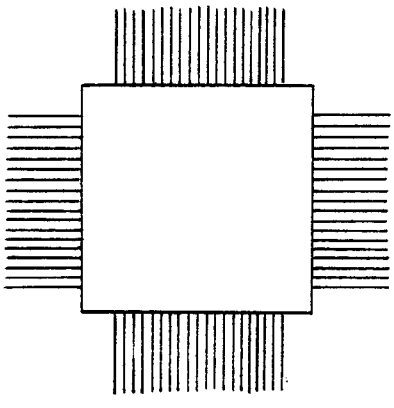
Dit is opnieuw de figuur van 1.0. Zowel de input als de output kan (en zal over het algemeen) uit vele parallelle signalen bestaan. Reeds in het simpele geval van een flipflop hebben we drie input-signalen en twee (afhankelijke) output-signalen. Bij een register zijn het er uiteraard heel wat meer. Bij een binaire operator (bijv. een opteller) kunnen allerlei combinaties van input- en output-signalen een rol spelen.

In architectuur-schema's volstaan we meestal met één of enkele lijnen om de informatie- (signaal-)overdracht van de ene box naar de andere aan te geven. Vaak wordt daarbij zelfs de richting niet eens vermeld omdat deze uit de context duidelijk is. Uiteraard wordt de richting aangegeven als dit voor een goed begrip noodzakelijk is. In een schema-onderdeel zoals:

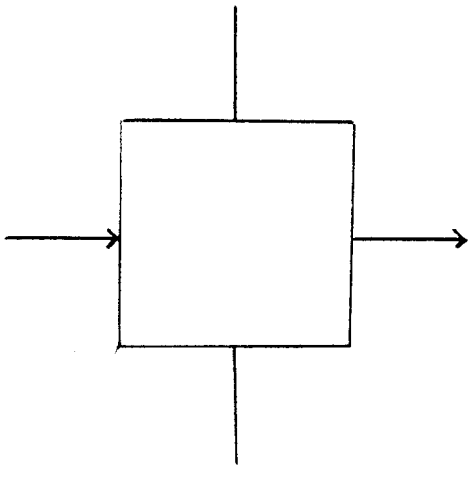


doet men er goed aan zich te realiseren dat iedere lijn op zich vele parallelle signalen kan voorstellen. Uit de vermelding CLOCK bij een van de inputs is echter duidelijk dat dit precies 1 signaal is. Bovenstaand plaatje kan nu evenzeer één enkele flipflop voorstellen, als bijv. een (1-bits of n-bits) optelcircuit met register en een transport van het resultaat naar twee andere boxen. Uit het CLOCK-input-signaal is direkt duidelijk dat we hier met een synchrone sequentiële unit te maken hebben (een box met geheugen dus).

Een LSI-schakeling (een chip) kan alsvolgt in een schema voorkomen:



Het opmerkelijke van dit figuurtje is, dat men zoiets ook werkelijk ongeveer ziet (bij voldoende vergroting) wanneer men een chip van boven bekijkt.
De (lineaire) vergroting is ca. 2 à 4.



In (globale) schema's tekent men over het algemeen slechts 1 (of enkele) verbindingslijnen, waar eigenlijk vele (parallelle) signalen in en/of uitgaan.
Een dergelijke verbinding wordt "bus" genoemd.

1.6 DE HARDWARE INTERFACE

De hardware interface is het eindprodukt van een zeer groot aantal (combinational- en sequential-logic) schakelingen (zowel asynchroon als synchroon) tot een combinatie van processor & geheugen (ipv. met flipflops op een chip, kan het geheugen ook gerealiseerd zijn met ferrietkernen). Voor het begrip processor verwijzen we naar hoofdstuk 3, voor een bepaalde processor (de FANCY-CPU) naar hfdst. 4 waar ook gedetailleerd wordt ingegaan op de relatie processor ⇔ geheugen.

De LSI-technologie maakt het mogelijk dat een complete hardware-interface op één chip wordt gerealiseerd. In de MSI-technologie betaalt men met veel meer ruimte voor meestal veel grotere schakelsnelheden (thans nog wel in de orde van 100 tot enkele malen 10000 keer zo snel).

Een globale karakteristiek van een hardware-interface (de laagste interface, zie 3.3) is - ietwat populair - alsvolgt:

> Een zeer groot "electronisch telraam" waarop,
> gegroepeerd in rijen van 8 of veelvouden daarvan (zie 2.1),
> flipflops zijn bevestigd: geheugenelementen voor een 0 of een 1.
> Dit telraam is het geheugen - een omvang van enkele miljoenen
> flipflops (mogelijk ferrietkernen) is normaal.

> Een veel kleiner, maar wel zeer veel sneller en gecompliceerder telraam,
> de processor, kan flipflopstanden ("bitrijen", zie hfdst. 2)
> overnemen (de operatie FETCH) in een veel kleiner eigen
> register-geheugen, en omgekeerd nieuw berekende flipflopstanden
> in het geheugen opbergen (de operatie STORE).

> De processor bestaat uit een Arithmetic Logic Unit (ALU) en een
> Control Unit (CU). In de ALU zijn de basis-operaties op bitrijen
> en individuele bits gerealiseerd, de CU stuurt de ALU op grond
> van in het geheugen gecodeerde instructies.

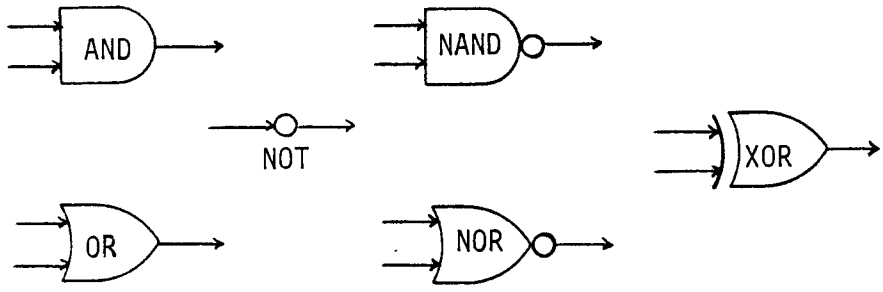
> Een CLOCK "dirigeert" de CU en daarmee de ALU en bepaalt uiteindelijk
> de basis-rekensnelheid: de "processor-cycle". Ook het geheugen-contact
> staat onder regie van de CLOCK: de "memory-cycle" is normaal een veel-
> voud van de processor-cycle. De processor-cycle wordt tegenwoordig
> meestal in νsec (1 nanoseconde = 10^{-9} sec) gegeven, de memory-cycle
> (nu nog) in μsec (1 microseconde = 10^{-6} sec).

De in dit hoofdstuk geschetste interfaces hebben de volgende hierarchie:

operaties op bit-rijen

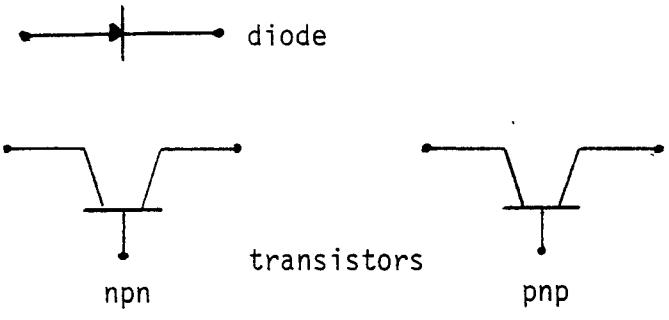
▪▪ hardware interface

logical gates:



▪▪ logical interface

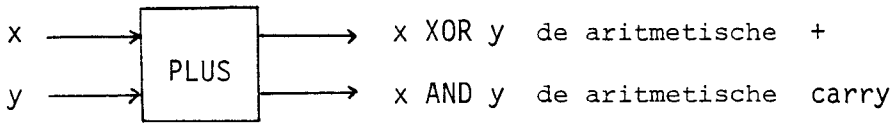
diodes & transistors:



▪▪ electronica interface

laws of the electrodynamic field
notably the properties of semiconductors

De operaties op bitrijen boven de hardware interface worden gedetailleerd behandeld in de hoofdstukken 2 en 4. Het zijn in essentie de logische- en aritmetische operaties van de ALU. Een belangrijke elementaire operatie van de ALU is de aritmetische PLUS die twee ingangsignalen omzet in twee uitgangsignalen:



Het loont de moeite een keer na te gaan wat de optimale (NAND- of NOR-) logische poortschakeling is voor de aritmetische PLUS.

2. B I N A I R E D A T A R E P R E S E N T A T I E

2.0 HET BINAIRE STELSEL

De basisinformatieeenheid in de computer-hardware is de bit (binary digit). Van deze tweewaardige eenheid wordt de waardenverzameling meestal aangegeven als $\{0,1\}$.

k	2^k	k	2^k	k	2^k
1	2	11	2048	24	16777216
2	4	12	4096	32	4294967296
3	8	13	8192		
4	16	14	16384		
5	32	15	32768		
6	64	16	65536		
7	128	17	131072		
8	256	18	262144		
9	512	19	524288		
10	$1024 = K \approx 10^3$	20	$1048576 = M \approx 10^6$		

In de computer-vakliteratuur worden de voorvoegsels K(=kilo) en M(=Mega) vaak in een ietwat afwijkende (binair gemotiveerde) betekenis gebruikt: K=1024 ($\approx 10^3$) en M=1048576 ($\approx 10^6$). Een 16K byte geheugen is dus een geheugen van 16.2^{10} bytes (zie 2.1), een 1M woord geheugen is een geheugen van 2^{20} machinewoorden (zie 2.1).
Een 16MHz kwartskristal is echter een kristal met een frequentie van 16.10^6 Hertz.

Net als cijferrijen in het decimale getalstelsel kunnen we bitrijen interpreteren in het positioneel binair getalstelsel:

$$\underline{\text{decimaal}} \quad 2504 = 2\times10^3 + 5\times10^2 + 0\times10^1 + 4\times10^0$$
$$= 4\times10^0 + 0\times10^1 + 5\times10^2 + 2\times10^3$$
$$\underline{\text{binair}} \quad 1101101 = 1\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 \ (=109)$$
$$= 1\times2^0 + 0\times2^1 + 1\times2^2 + 1\times2^3 + 0\times2^4 + 1\times2^5 + 1\times2^6$$

Merk op dat we (in beide stelsels!) eigenlijk "van rechts naar links" lezen; een erfenis van de Arabische cultuur.

Voor het snel kunnen omrekenen van binair naar decimaal, en vice versa, is het nuttig onderstaande bitrijen en hun decimale getalwaarden vlot uit het hoofd te kennen:

	eventueel ook:
000 = 0	1000 = 8
001 = 1	1001 = 9
010 = 2	1010 = 10
011 = 3	1011 = 11
100 = 4	1100 = 12
101 = 5	1101 = 13
110 = 6	1110 = 14
111 = 7	1111 = 15

Merk op:

"een nul erachter"
betekent:
"twee keer zoveel".

bijv. 11 = 3 | 101 = 5
110 = 6 | 1010 = 10
1100 = 12

enz.

Een (heel belangrijke) eigenschap van het binaire getalstelsel is dat er geen tafels van vermenigvuldiging nodig zijn: in de hardware geldt heel letterlijk dat vermenigvuldigen = herhaald optellen.
Voor negatieve - en niet-gehele getallen zie 2.3 en 2.4.

De basis-operatie voor de binaire aritmetiek is de (aritmetische) +

```
 0     0     1     1
 0     1     0     1
 +     +     +     +
00    01    01    10
 ↑     ↑     ↑     ↑
 └─────┴─────┴─────┘
       carry  (zie 1.6)
```

In een (uiteraard altijd eindige) bitrij kan de carry aanleiding geven tot OverFlow (OF): de carry valt buiten de bitrij die de grootte van het getal representeert. In behoorlijke hardware zal een OF-bit (een speciale "sturende" flipflop) altijd overflow signaleren (er ontstaat dan immers een fout antwoord):

```
 1111111
       1
       +
1←0000000
↑
└──────────  carry geeft overflow
```

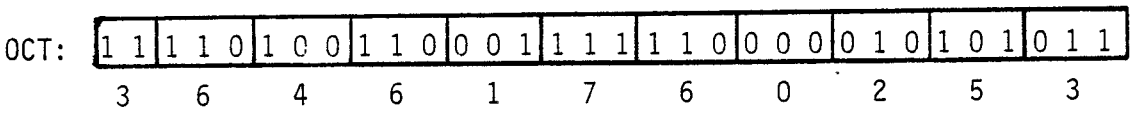
2.1 GROEPERINGEN VAN BITS

2.1.0 Octaal en hexadecimaal

Bitrijen zijn poly-interpretabel - dit is een essentiële eigenschap van alle computer hardware. Men kan niet zonder additionele informatie zomaar "zien" wat een gegeven bitrij nu betekent. Niettemin is het vaak nodig hem op een menselijker wijze te kunnen benoemen. Hiervoor zijn twee systemen (naast en vaak door elkaar) in gebruik:

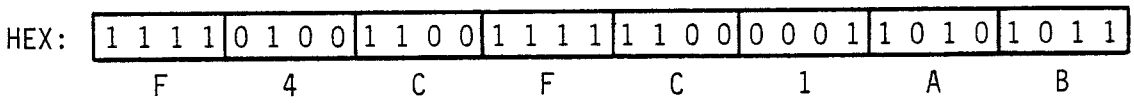
octaal (bijv. 8r 36461760253)

De bitrij wordt (van rechts naar links!) in groepjes van 3 bits ingedeeld en dan met de cijfers 0 t/m 7 benoemd:



hexadecimaal (bijv. 16r F4CFC1AB)

De bitrij wordt in groepjes van vier bits (ook wel nibbles genoemd) ingedeeld en met de cijfers 0 t/m 9, A(=10), B(=11), C(=12), D(=13), E(=14) en F(=15) benoemd. Merk op dat A, B, C, D, E en F hier als cijfers worden gehanteerd:



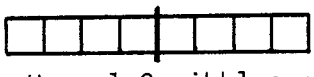
Het is van belang dat men zich goed realiseert dat "octaal" en "hexadecimaal" uitsluitend het benoemen van een ongeïnterpreteerde bitrij is. De bitrij kan nog van alles betekenen.

De octale- en vooral de hexadecimale benoeming van specifieke bitrijen heeft nog een groot praktisch nut:

Op een "kale" machine (de machine zonder firmware) moet minimaal een soort "ultimate codegenerator" (ook wel "inleesprogramma" genoemd) aanwezig zijn, anders zou men niet beschikken over een middel om bitrijen te definieren in (het geheugen van) zo'n kale machine. Meestal accepteert zo'n ultimate codegenerator hexadecimale (event. octale) codes; in het alleruiterste geval kunnen ze (via een diepverborgen schakelaarspaneel) "met de hand" worden ingebracht (zie 0.2 sub e).

2.1.1 Bits, nibbles, bytes en kbytes (machinewoorden)

Afhankelijk van de omvang en de "power" van een computersysteem wordt gewerkt met betrekkelijk "korte" tot betrekkelijk "lange" bitrijen. Bitrijen van verschillende lengte spelen in ieder systeem een rol. We onderscheiden in eerste instantie: *

- de bit 
  1 bit dus. De betekenis is altijd een ja/nee beslissing (true/false). Enkelvoudige bits worden voor zeer uiteenlopende doelen gebruikt, hun rol wordt verderop vanzelf wel duidelijk.
- de nibble 
  4 bits dus. Een nibble kan 16 ($=2^4$) verschillende waarden aannemen. We hebben al gezien hoe deze waarden hexadecimaal kunnen worden benoemd. Nibbles spelen ihb. een rol in de BCD-aritmetiek (zie 2.3.0).
- de byte 
  8 bits dus. Hoewel 2 nibbles een byte vormen, moet men een byte vooral zien als een zelfstandige bitsgroepering die 256 ($=2^8$) verschillende waarden kan hebben. De byte wordt gebruikt voor de codering van CHARs (zie 2.2) en voor (zeer) kleine INT-getalsystemen (bijv. van 0 t/m 255, of van -128 t/m +127).

De byte (vermoedelijk een opzettelijke misspelling van "bite"="beet" of "snack") is op vrijwel alle micro's en op de meeste moderne machines tevens de kleinste (adresseerbare) geheugeneenheid: een geheugencel is (minimaal) 1 byte.

Behalve op de hele kleine micro's wordt (bovendien) gerekend met 2bytes (een 16bits eenheid dus), 3bytes (weinig gangbaar, zie echter 2.4), 4bytes (de meeste grote moderne systemen), 6bytes (enkele grote systemen (ihb. voor LONG REAL aritmetiek) en 8bytes (enkele zeer moderne "super"-computers).

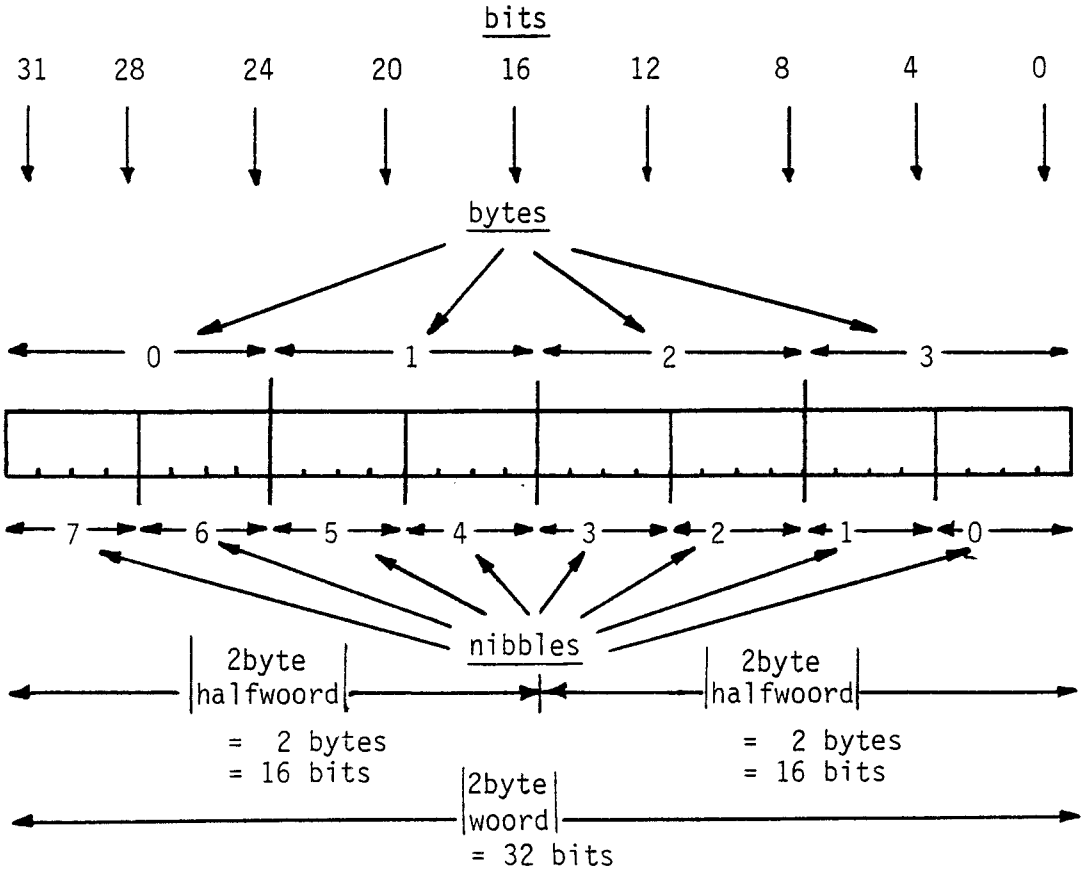
De meeste grotere systemen hebben geheugenselectie op zowel 1bytes, als 2bytes, als 4bytes (die dan vaak "woorden" worden genoemd).

* In de datatransmissie (datacommunicatie) onderscheidt men ook nog de dibit, een eenheid van 2 bits dus.

De byte is de standaard (in wording) van de moderne hardware. In het geheugen van de FANCY (en vooral in de FANCY-processor, zie hfdst. 4) onderscheiden we:

- de 2byte (half machinewoord)
  Dit is o.a. de basis-eenheid voor de codering van de FANCY-instructies en tevens voor de korte INT-aritmetiek en de adresberekening.
- de 4byte (het machinewoord)
  De registers in de FANCY-processor zijn allemaal 4byte-registers; ze bestaan dus uit 32 bits = het "machinewoord" van de FANCY. De 4byte is tevens de eenheid voor de normale INT-aritmetiek en de floating-point - aritmetiek.
- de 8byte (het dubbele machinewoord)
  Twee machinewoorden kunnen (in de software!) worden gekoppeld tot een 8byte-eenheid, bijv. voor "dubbellengte"-aritmetiek, voor complexe getallen, etc.

Overzicht van 32-bits woord-indeling.



Let in voorgaand overzicht op de (afwijkende) nummering van de bytes (van links naar rechts) tov. de nummering van nibbles en bits (van rechts naar links), dit in verband met de conventies van de positionele getal-systemen.

Hieronder een overzicht van gangbare bitgroeperingen van bekende datatypen.

data-type	representatie
BOOL	bit { 0 = TRUE/FALSE, 1 = FALSE/TRUE } komt allebei voor
<digit>	nibble (alleen voor BCD)
CHAR	byte (hardware afhankelijk 6, 7, 8 of 9 bits)
INT (FIXED)	halfwoord - SHORT INT woord - INT dubbel woord - LONG INT
REAL (FLOAT)	woord - REAL dubbel woord - LONG REAL
REF	halfwoord - address woord - long address

Dat de woordlengte op verschillende computers niet altijd dezelfde is mag blijken uit de volgende tabel. In het vervolg gaan wij echter steeds uit van een woordlengte van 32 bits (FANCY), omdat deze vermoedelijk standaard gaat worden.

8	(byte)	micro's
16	(in feite "halfwoord")	micro/mini
24	(met 4 6bits of 3 8bits bytes)	ICL
32	(met 4 normale bytes)	IBM
36	(met 4 9bits bytes)	UNIVAC
60	(met 10 6bits bytes)	CDC (maar geen byte-adressering)
64	(met 2 4bytes-halfwoorden)	STAR, CRAY-1, nieuwe CYBERs(?)

2.1.2 Pariteits-bit en fout-detectie

Binaire informatie kan in verschillende typen geheugens (zie DEEL II) en op verschillende media (ferrietkernen, magneetband, ponsband, etc.), tijdelijk of meer permanent, worden bewaard. Binaire informatie wordt in iedere configuratie dan ook veelvuldig van het ene medium naar het andere getransporteerd. Zowel bij het bewaren als bij het transporteren (communicatielijnen, vooral op de grotere afstanden, en ook draadloos) kunnen storingen optreden. Het is uiteraard van belang dergelijke storingen tijdig te kunnen signaleren.

Een algemeen gebruikte methode is de zg. pariteits-bit. Aan de eigenlijke informatiebits wordt nog een extra bit toegevoegd die ervoor zorgt dat het aantal bits in een groepering altijd even is (even pariteit), of oneven (oneven pariteit). Dergelijke pariteitsbits verdwijnen meestal al bij de laagste interfaces achter de schermen (kunnen dus worden beschouwd als een pure electronica-hardware aangelegenheid). Een uitzondering is de byte als drager van CHARs (zie 2.2.0).

Het manipuleren (en interpreteren) van teksten (bijv. in de communicatie met terminals) is in elk geval een basis-software aangelegenheid, en dat impliceert soms ook de reactie op communicatiestoringen. De byte is een "ruim pak" voor een CHAR: in 7 bits codeert men het toetsenbord van een moderne schrijfmachine (zie 2.2.0). Het achtste bit kan dan als (software-toegankelijk) pariteitsbit worden gebruikt.

Bij het opslaan van binaire informatie (bytes) op magneetband kunnen behalve pariteitsbits (binnen de byte, of nog weer toegevoegd aan de byte) ook hele bytes met louter pariteitsbits (op elke 7 bytes bijv. een pariteits-byte (een zg. "dwars-pariteits controle")) worden toegevoegd.

Naast fout-detectie is soms ook fout-herstel ("error-recovery") noodzakelijk (op zwaar gestoorde communicatielijnen). Hiervoor zijn speciale ("redundant") codes ontwikkeld, die echter buiten het bestek van deze syllabus vallen.

2.2 CODERING VAN TEKSTEN (datatype [ ]CHAR)

2.2.0 Character-code

Een tekst is (per definitie) een rij CHARs. Welke CHARs en in welke binaire codering gemanipuleerd kunnen worden, is in sterke mate afhankelijk van het gegeven computer-systeem. De codering is weliswaar een software (firmware)-aangelegenheid, maar niettemin sterk gebonden aan hardware-beperkingen van de I/O-apparatuur. Zo heeft CDC een nogal beperkte eigen codering (op basis van een 6-bits "byte"), terwijl IBM met de zg. EBCDIC ook een weinig gelukkige greep heeft gedaan.

De toekomstige standaard is waarschijnlijk de door de meeste moderne hardware (ihb. micro's en mini's) gebruikte ASCII-code, die wij ook voor de FANCY hebben aanvaard:

American Standard Code for Information Interchange (ASCII).

		$b_6b_5b_4$ (column)							
	row	000	001	010	011	100	101	110	111
$b_3b_2b_1b_0$	(hex)	0	1	2	3	4	5	6	7
0000	0	NUL	DLE	SP	0	@	P		p
0001	1	SOH	DC1	!	1	A	Q	a	q
0010	2	STX	DC2	"	2	B	R	b	r
0011	3	ETX	DC3	#	3	C	S	c	s
0100	4	EOT	DC4	$	4	D	T	d	t
0101	5	ENQ	NAK	%	5	E	U	e	u
0110	6	ACK	SYN	&	6	F	V	f	v
0111	7	BEL	ETB	'	7	G	W	g	w
1000	8	BS	CAN	(	8	H	X	h	x
1001	9	HT	EM	)	9	I	Y	i	y
1010	A	LF	SUB	*	:	J	Z	j	z
1011	B	VT	ESC	+	;	K	[	k	{
1100	C	FF	FS	,	<	L	\	l	|
1101	D	CR	GS	-	=	M	]	m	}
1110	E	SO	RS	.	>	N	^	n	~
1111	F	SI	US	/	?	O	_	o	DEL

Control Codes

NUL	Null	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronize
BEL	Bell	ETB	End transmitted block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete or rubout

In de ASCII-code wordt onderscheid gemaakt tussen "control characters" (NUL t/m US) en "printable characters" (van SP tot DEL, 95 dus). De control characters worden gebruikt voor de sturing van de I/O-apparatuur (electronica/ hardware dus), waarbij de characters NUL en DEL geen andere betekenis hebben dan "negeer mij". SP (spatie) is echter een printable character.

Merk op dat in de ASCII-code het achtste bit (b_7) is vrijgehouden als pariteitsbit (zie 2.1.2).

2.2.1 Strings

Een CHAR-array ([]CHAR) van in principe variabele lengte wordt string genoemd. De binaire codering van een string bestaat uit essentieel twee delen:

1. De []CHAR, gecodeerd als een (aaneengesloten) rij printable ASCII-bytes in het geheugen. Soms is het noodzakelijk de bytes-rij tot halfwoorden of hele woorden aan te vullen (achterin) met DELs; dat is precies waarvoor DELs dienen.
2. Meta-informatie betreffende de lengte van de string en zijn plaats ("adres") in het geheugen (eventueel ook nog additionele meta-informatie). Deze meta-informatie wordt op de FANCY in 1 machinewoord (een 4byte) gepakt: de zg. string-descriptor.

Soms is het adres van de string apriori bekend (bijv. in vaste buffers, of op specifieke plaatsen in een sourcetext, zie 3.1.4); zijn lengte kan dan ook volgen uit het voorkomen van een control-character in de string zelf.

2'. Een string wordt in de FANCY-firmware altijd afgesloten met NUL, en/of LF. Het verschil tussen NUL en LF is dat NUL altijd het einde van een (deel)string aangeeft en LF bovendien (!) het einde van een regel (line) aankondigt. NULs en DELs hebben nooit effect op de lay-out van een tekst (het zijn beide "delete"-characters), LF heeft dat (althans in potentie) wèl.

Zie verder DEEL II.

2.3 INTEGER ARITMETIEK (datatype INT)

2.3.0 Binary Coded Decimal (BCD)

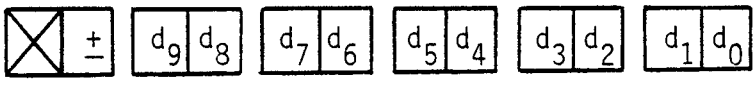
Er is een alleen door vrij omvangrijke software (firmware) overbrugbare discrepantie tussen het decimale-, en het binaire getalstelsel (zie ook 2.5). De kloof zou heel wat gemakkelijker te overbruggen zijn als de mens bijv. 4 (of, nog beter, 8) vingers aan elke hand (gehad) zou hebben, en dus op "natuurlijke wijze" octaal- of hexadecimaal zou zijn gaan rekenen.

We hebben gezien dat de electronica essentieel binair is (toekomstige ontwikkelingen maken het weinig waarschijnlijk dat dit ooit anders zal worden). Niettemin is het niet zo moeilijk de electronica decimaal te laten rekenen: voor elk decimaal cijfer gebruikt men 1 nibble, die dan echter voor slechts 5/8 wordt gebruikt. Behalve voor een (toch wel) duurdere hardware, betaalt men dus ook voor een minder efficiënt geheugengebruik (37.5 % verlies).

Er zijn twee redenen voor het gebruik van een dergelijk Binary Coded Decimal (BCD) systeem:

1. De noodzaak van conversie en deconversie (zie 2.5) vervalt. Dit is vooral van belang wanneer registerinhouden direct zichtbaar gemaakt moeten worden op LCD's en LED's (pocket-calculators!).
2. De financiële rekenkunde eist een bepaald soort precisie. Dit is waarschijnlijk een drogreden. Er zijn inderdaad enkele afrondingsperikelen, maar deze kunnen door software worden overbrugd.

In BCD worden steeds 2 nibbles in 1 byte gepakt (twee decimals dus per byte). Een kop-byte kan worden gebruikt voor een voorteken:



Voor de binaire codering van de decimalen (d_i) en het voorteken bestaan geen vaste conventies: de hardware kan hier specifieke eisen stellen (bijv. voor electronische koppeling aan LCD's en LED's), die van systeem tot systeem kunnen verschillen. BCD-aritmetiek valt verder buiten het bestek van deze syllabus.

2.3.1 INT-aritmetiek

De zuiver binaire aritmetiek voor gehele getallen kent slechts één probleem: de representatie van negatieve getallen. Hiervoor zijn echter maar even 4 systemen in zwang, die hieronder (zeer summier) worden behandeld.

We gaan er van uit dat het datatype INT in een 32-bits machinewoord wordt gerepresenteerd. Op de FANCY worden echter ook 16-bits halfwoorden (2bytes) gebruikt (SHORT INT) en er zijn zelfs INT-instructies voor 1bytes (SHORT SHORT INT). De conventies hiervoor zijn onderling compatibel.

Voor LONG INT (bijv. een 8byte) kan (zg. "dubbellengte") aritmetiek worden gemaakt in software

Hoewel de binaire representatie op het firmware-level reeds achter de schermen is verdwenen, loont het toch de moeite om de essentie van de vier hieronder geschetste systemen te doorgronden. De "machine-programmeur" moet zéker enig inzicht hebben in enkele details van de binaire aritmetiek.

Voor het vastleggen van INT-data dienen we onderscheid te maken in unsigned en signed integers.
Van de unsigned integers (de natuurlijke getallen) kunnen de getallen 0 t/m 2^{32}-1 in een 32-bits woord worden vastgelegd. Op de FANCY wordt de unsigned INT-aritmetiek (in halve woorden) gebruikt voor de (enkelvoudige) adresberekeningen (zie hfdst. 4 en DEEL II).

Voor vastleggen van gehele getallen (positief en negatief) kennen we de volgende systemen:

- signed magnitude
- 1's complement
- 2's complement
- excess 2^{m-1}

In al deze systemen is het meest linkse bit de tekenbit.

→ Signed magnitude

Bit 31 tekenbit: 0 = positief, 1 = negatief.
Bits 30 - 0 bevatten de absolute waarde van het getal.

→ 1's complement

Bit 31 tekenbit: 0 = positief, 1 = negatief.
Om een getal te inverteren (additief) dienen alle bits (incl. tekenbit) omgekeerd te worden.

→ 2's complement

Bit 31 tekenbit: 0 = positief, 1 = negatief.
Inverteren van een getal: eerst alle bits (incl. tekenbit) omkeren en er vervolgens 1 bij optellen.

→ Excess 2^{m-1}

In dit systeem wordt een getal opgeslagen als de som van het getal zelf en 2^{m-1} (waarin m = het aantal bits in de representatie). Voorbeeld: voor 8-bits getallen, m=8, wordt dit excess 128 genoemd en een getal wordt opgeslagen als som van 128 en het getal zelf.
Dus -3 wordt -3 + 128 = 125 en in 8 bits is de representatie van -3 dan: 01111101.
Bij nadere beschouwing blijkt dit systeem identiek te zijn aan het 2's complement met geïnverteerd tekenbit.

Zowel de signed magnitude als de 1's complement methode hebben 2 representaties voor het getal 0.

+0	0 000.....000	-0	1 000.....000	signed magnitude
		-0	1 111.....111	1's complement

De 2's complement en excess 2^{m-1} methode hebben 1 representatie voor het getal 0.

	2's complement	excess 2^{m-1}
0	0 000.....000	1 000.....000

Signed magnitude en 1's complement zijn echter symmetrisch tov. 0, de beide andere methoden zijn dit helaas niet.

symmetrisch	asymmetrisch
$1-2^m \leq n < 2^m-1$	$-2^m \leq n \leq 2^m-1$

De reden voor deze verschillen ligt voor de hand. Idealiter zouden we een coderingssyteem met 2 eigenschappen willen hebben:

- slechts 1 representatie voor nul,
- precies evenveel positieve als negatieve getallen.

De laatste eigenschap geeft een verzameling van een even aantal coderingen, terwijl m bits altijd een even aantal verschillende bitrijen kunnen bevatten. Er is dan altijd óf 1 bitrij teveel óf 1 bitrij te weinig. Dit is ietwat dubbelzinnig: we krijgen óf een "nul" teveel (1's complement) óf een negatief getal teveel (2's complement) - deze kan men gebruiken voor "undefined".

De optelling van twee binaire cijfers is in onderstaande tabel gegeven:

0	0	1	1	
0	1	0	1	
+	+	+	+	
0	1	1	0	som
0	0	0	1	carry

Twee binaire getallen worden opgeteld door de corresponderende bits van de getallen op te tellen, te beginnen bij het meest rechtse bit. Als er een carry ontstaat, dan wordt deze één plaats links ervan mee opgeteld; net als bij de decimale aritmetiek.

Bij de 1's complement aritmetiek wordt een carry, ontstaan door optelling van de meest linkse bits, opgeteld bij het meest rechtse bit van de som. Dit proces heet end-around-carry.

Bij de 2's complement aritmetiek wordt een carry, ontstaan door de optelling van de meest linkse bits, weggegooid.

Voorbeeld:

decimaal	1's complement	2's complement
10	00001010	00001010
-3	11111100	11111101
+	+	+
7	1 00000110	1 00000111
	carry 1	↓
	+	weggegooid
	00000111	

In alle systemen kan overflow ontstaan: het resultaat van een optelling past niet meer in de gegeven #byte. Bij overflow is het resultaat zonder meer incorrect. In alle (goede) hardware wordt overflow uiteraard (zodra het ontstaat) gesignaleerd. De overflowdetectie is in de 4 systemen verschillend.

In de binaire hardware zijn alle rekenkundige operaties direct (ook electronisch) afgeleid van de optelling.

aftrekken:	a - b wordt geëffectueerd als a + (-b)
vermenigvuldigen:	wordt geëffectueerd als herhaald optellen
delen:	wordt geëffectueerd als herhaald aftrekken.

In de computer-litteratuur wordt de INT-aritmetiek meestal FIXED-POINT aritmetiek genoemd. Men gaat ervan uit, dat de programmeur de INTs interpreteert als getallen die van een (decimale- of binaire-) punt zijn voorzien. Deze "punt" (die het gehele deel scheidt van het breukdeel) komt dan vaak pas tevoorschijn bij het afdrukken van de INT. Men rekent bijv. in millimeters, milliseconden, milligrammen of tienden van centen en drukt dan uiteindelijk af in resp. meters, seconden (mogelijk uren), grammen (mogelijk kilogrammen) en guldens. Uiteraard moet men bij de input de getallen overeenkomstig "schalen". Dit is de "klassieke" manier van het hanteren van niet-gehele getallen.

In een minder rigide behandeling van niet-gehele getallen kan men de schalingsfactor (nog steeds naar keuze decimaal of binair) bijhouden in een aparte (SHORT)INT. Het rekenen met een dergelijke expliciete schalingsfactor heeft uiteindelijk geleid tot de zg. floating-point representatie (zie 2.4), die nu (in een reeds sterk gestandaardiseerde representatie) op vrijwel alle computersystemen (vanaf de grote micro's) gerealiseerd is. Daarin is de schalingsfactor altijd de plaats van de binaire punt.

De basis van een dergelijke ("ingebouwde") floating-point aritmetiek is uiteraard de (altijd aanwezige) INT-aritmetiek.

2.4 FLOATING POINT ARITMETIEK (datatype REAL)

2.4.0 Floating-point representatie

De floating-point representatie dient om REALs met een constante relatieve precisie (in een constant aantal decimalen, en dus een constant aantal bits) over een zo groot mogelijk bereik (een zover mogelijk verschuivende - "drijvende" - decimale/binaire punt) vast te leggen. Het is geen grote mathematische toer om in te zien dat dit wordt bereikt mbv. twee INTs m (mantisse) en e (exponent):

$r = m \times b^e$

b = 2 (het meest natuurlijke)
= 4 } worden ook wel gebruikt
= 16 }

b = 10 is volstrekt in onbruik geraakt

Met b = 2 zijn er voor m nog vele mogelijkheden, bijv.

- m is een INT (de binaire punt staat direct rechts van de meest rechtse bit)
- $\frac{1}{2} \leq |m| < 1$ (de binaire punt links)

In het laatste geval (de zg. genormeerde mantisse) is de meest significante bit altijd verschillend van de tekenbit. Merkwaardig genoeg heeft men betrekkelijk laat ingezien dat men dus(!) deze bit ook niet hoeft op te slaan (1 bij positieve mantisse en 0 bij negatieve mantisse): deze verzwegen bit wordt wel "hidden bit" genoemd.

De waarde van b (de "floating-point base") is een systeem-constante en impliciet aan de gekozen floating-point hardware. Het floating-point getal r wordt dus gerepresenteerd door 2 INTS m en e:

r = (m , e)

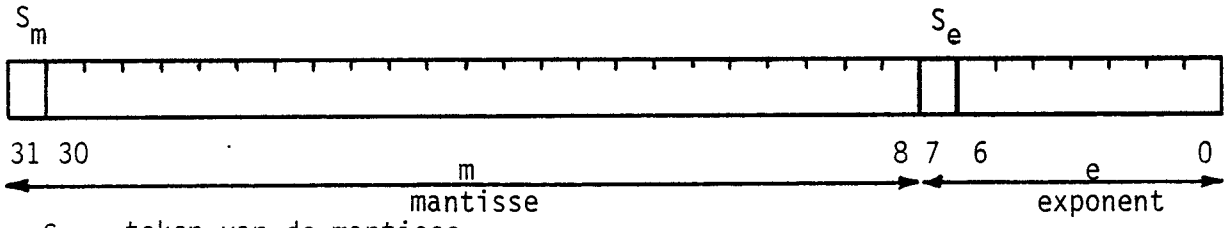
Men is nog vrij in de wijze waarop m en e in een *k*byte gepakt worden, het ligt voor de hand ze in één 4byte (of 6byte of 8byte) te pakken en ze dus als één bitrij in het geheugen op te slaan, volstrekt noodzakelijk is dat niet. In de wijze waarop men ze pakt is men ook geheel vrij (dwz. de hardware is hier niet gebonden aan vaste conventies), evenals in het aantal bits dat men ter beschikking stelt voor m en e.

Niet zonder opzet kiezen we voor de FANCY een REAL-representatie die op geen enkel ander systeem (ons bekend) wordt gebruikt, maar die goed aansluit bij de INT-representatie van de FANCY - we doen dit omdat vrijwel elk processor systeem er toch ook zijn eigen floating-point representatie opnahoudt.

We kiezen voor m een 3byte en voor e een 1byte, de waarde van b = 2. We werken met een hidden bit; merk op dat we daarmee een relatieve precisie van ruim 7 decimalen halen (met opslag van de hidden bit gaat dat net niet):

$$b = 2 \quad \text{en} \quad \tfrac{1}{2} \leq |m| < 1 \quad \text{met hidden bit .}$$

Bijzondere gevallen zijn de representatie van 0.0 (waarbij zowel e als m beide de waarde 0 hebben), en dientengevolge(!) de representaties voor +0.5 en -0.5, waarvoor de dichtstbijliggende representaties werden gekozen. In een FANCY-4byte wordt een floating-point getal alsvolgt opgeslagen:



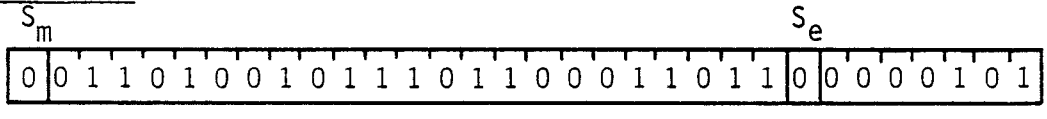
S_m = teken van de mantisse

S_e = teken van de exponent

De exponent wordt dmv. 2's complement vastgelegd en de mantisse mbv. 2's complement met hidden bit.

m is dan goed voor vastlegging van 7 decimalen en $-2^7 \leq e \leq 2^7-1$ en op deze wijze is een REAL-bereik van $2^{\pm 128} \approx 10^{\pm 38}$ mogelijk.

Voorbeeld:



Betekenis:

de hidden bit

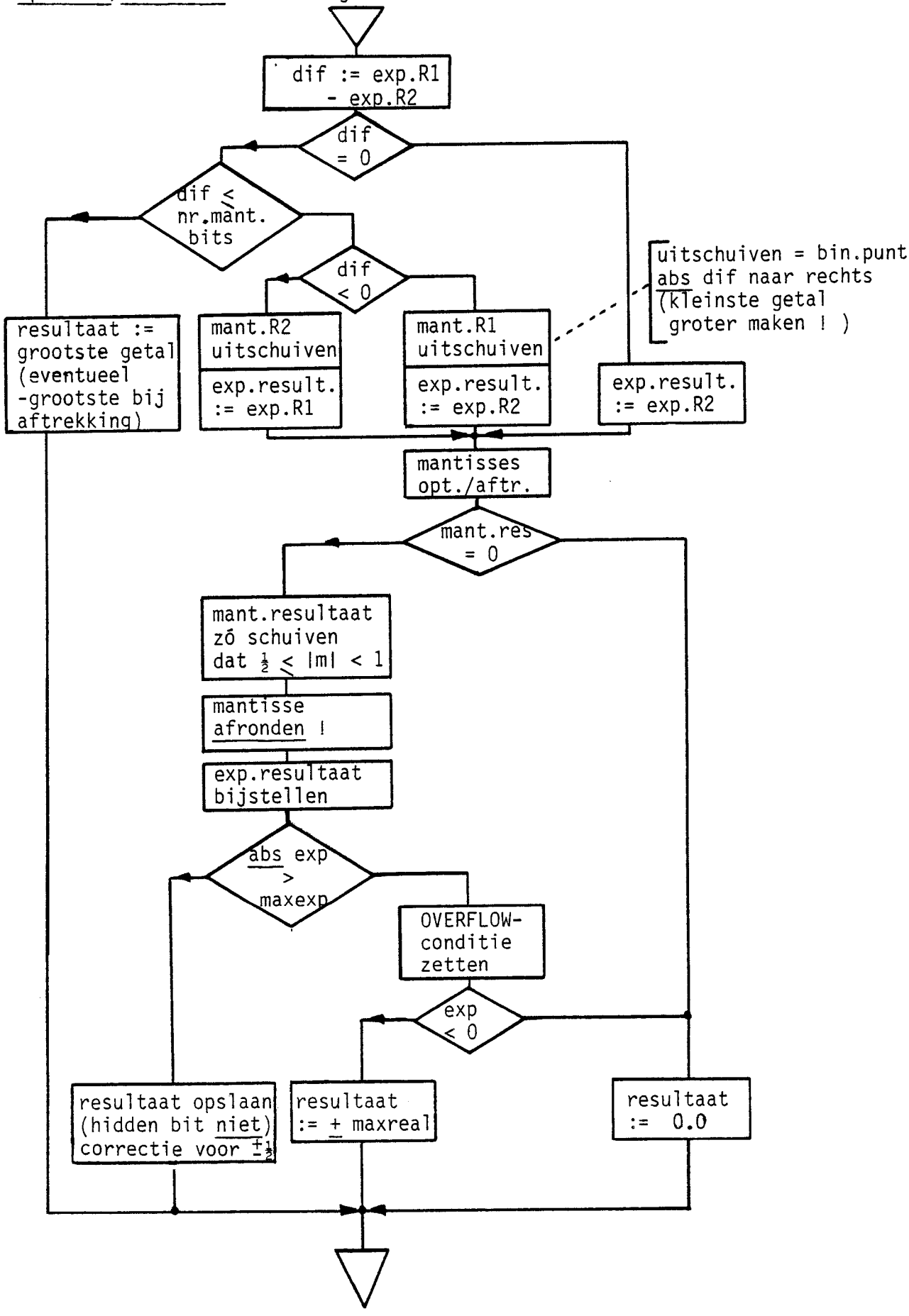
$+ \; . \; 10110100101110110001101 1 \times 2^5 =$

$+ \; 10110.100101110110001101 1 = + 22.59136$

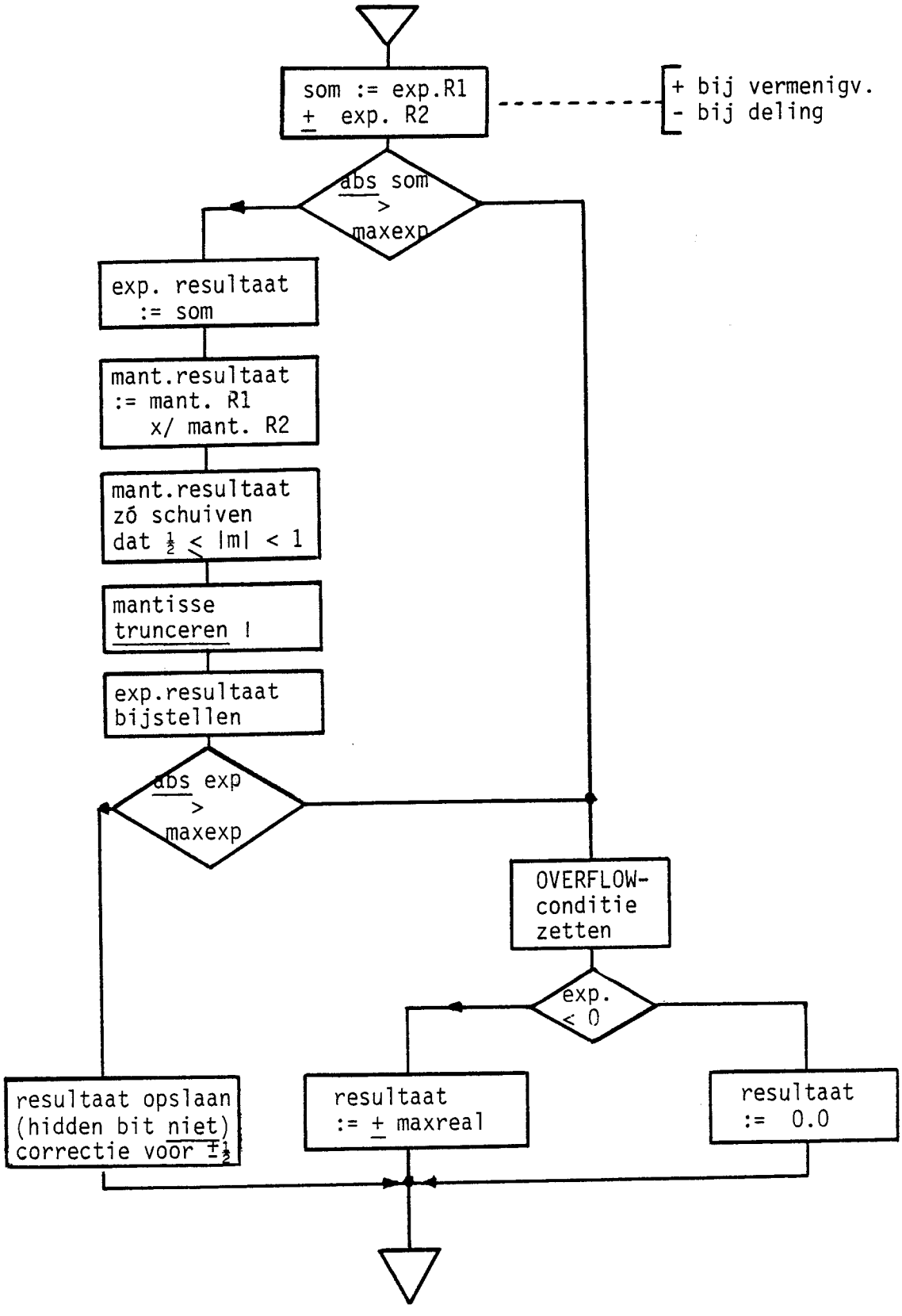
2.4.1 Schets van de floating-point operaties

In alle floating-point operaties worden mantisse en exponent direkt gescheiden en aan afzonderlijke operaties (resp. 1byte- en 4byte-aritmetiek) onderworpen. Na de berekeningen wordt het resultaat (mantisse & exponent) na de nodige afronding, normering en hidden-bit correctie (voor 0.0 en ± 0.5) in het voorgeschreven format gepakt. Globaal verlopen de basisoperaties als geschetst in de volgende blokschema's:

Optellen/aftrekken van REAL-getallen R1 en R2.



Vermenigvuldigen/delen van REAL-getallen R1 en R2.



Voorbeeld van een floating-point optelling (resp. aftrekking, als men de mantisse van y opvat als de geïnverteerde van -y) $z = x + y$:

m_x = (+) 0.00101100010101100100000 e_x = (-) 11111001 (e_x = -7)
m_y = (-) 1.11111010010110101101101 e_y = (-) 11111101 (e_y = -3)

De "." symboliseert zowel de hidden bit als de binaire punt.

Omdat $e_x = e_y - 4$ moeten we m_x over 4 posities naar rechts uitschuiven om de binaire punten van x en y gelijk gepositioneerd te krijgen:

$m_{x'}$ = (+) 0000010010110001010110010 $e_{x'}$ = (-) 11111101 ($e_{x'}$ = -3)
m_y = (-) 10111110100101101011011 01 e_y = (-) 11111101 (e_y = -3)
$m_{z'}$ = (-) 11000011010010000000011111 + $e_{z'}$ = (-) 11111101 ($e_{z'}$ = -3)

Normeren en hidden bit verdisconteren in resultaat:

m_z = (-) 1.00011010010000000111110 e_z = (-) 11111100 (e_z = -4)

Het loont de moeite dit een keer nauwkeurig na te gaan (en eventuele fouten eruit te halen) - men moet daarbij 7 decimalen meenemen bij binair-decimaal conversie.

De floating-point aritmetiek kent:

- OVERFLOW de (positieve) exponent "kookt over"
- UNDERFLOW de (negatieve) exponent "kookt over(onder)"

De FANCY-acties bij OVERFLOW | OF-flipflop zetten,
"maxreal" afleveren (met het goede teken)

De FANCY-acties bij UNDERFLOW | OF-flipflop zetten,
0.0 afleveren

Inspectie van de resultaat-mantisse beslist dan altijd welke van de drie gevallen (positieve of negatieve overflow, dan wel underflow) zich heeft voorgedaan.

De FANCY-oplossing van het probleem met de hidden-bit representaties van ±0.5 ivm. met conflict met de hidden-bit representatie voor 0.0 geeft de volgende "dichtstbijgelegen" representaties:

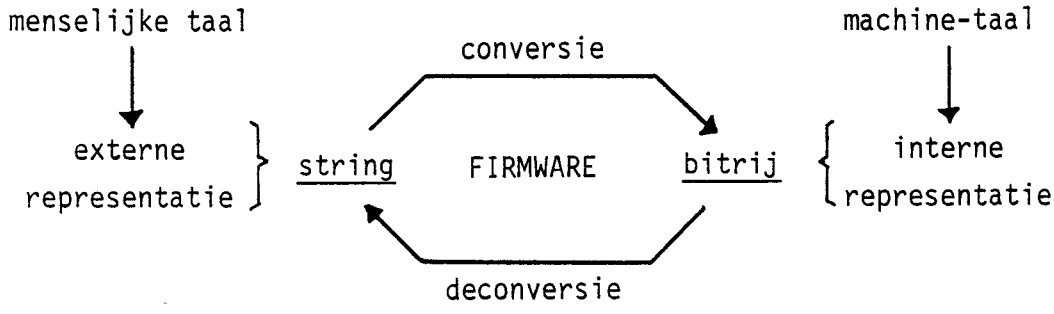
+0.5 = 0.11111111111111111111111 11111111
-0.5 = 1.00000000000000000000000 11111111

2.5 CONVERSIE EN DECONVERSIE

De binaire representatie van INTs en REALs als zodanig is voor de mens niet eenvoudig te produceren (te coderen) of te reproduceren (te decoderen). Uiteraard verwacht men dit wel van de computer-firmware.

We onderscheiden:

- conversie: het coderen van een tekst (opgebouwd uit cijfers, eventueel een decimale punt, misschien ook nog een letter "e" voor decimaal-floating-point, een voorteken "+" of "-", eventuele spaties voor leesbaarheid)
  naar binaire "fixed-" of "floating-point" representatie.
- deconversie: het terugvertalen van interne binaire representatie naar "tekst" in een gewenste "lay-out".



Een specifiek probleem bij conversie èn deconversie is, dat zeer veel verschillende strings dezelfde (fixed- of floating-point) bitrij kunnen definieren. Bij conversie betekent dit, dat men bedacht moet zijn op een grote varieteit van mogelijkheden; bij deconversie is nogal wat additionele informatie nodig om de juiste string te kunnen produceren. Dit "format"-probleem maakt het schrijven van (de)conversie-firmware tot een moeizaam en vaak zeer frustrerend probleem - ook allerlei afrond-haarkloverijen maken de zaak tricky. Het valt (gelukkig) buiten het bestek van deze syllabus - en terecht, want vrijwel alles wat hier te bedenken valt is goed gedocumenteerd, en hoeft nooit meer opnieuw bedacht te worden (en is ook niet meer voor verbetering vatbaar).

37 , 037 , +37 , +037 , + 00 0037 , 000 000 037 , enzovoort

3.141593 , 0.3141593e+1 , 3141593e-6 , +00314.1593e-2 ,
31415930000e-010 , +0.000003141593e6 , 3.1415 93 e0 , enzovoort

3. I N T E R F A C E A R C H I T E C T U U R

3.0 DEFINITIES

De in 0.1 gegeven definities zijn redelijk scherp, mits ze goed worden geïnterpreteerd:

hardware = alles aan een computersysteem dat ook zichtbaar is - althans in principe - als het systeem niet werkt:
de electronica, de geheugens,
de randapparatuur etc.

software = alles in een computersysteem dat zich pas kan manifesteren als het systeem werkt:
de basis-programmatuur (firmware, basis-software),
de databases, de packages en programmatheken,
de gebruikers-programmatuur etc.

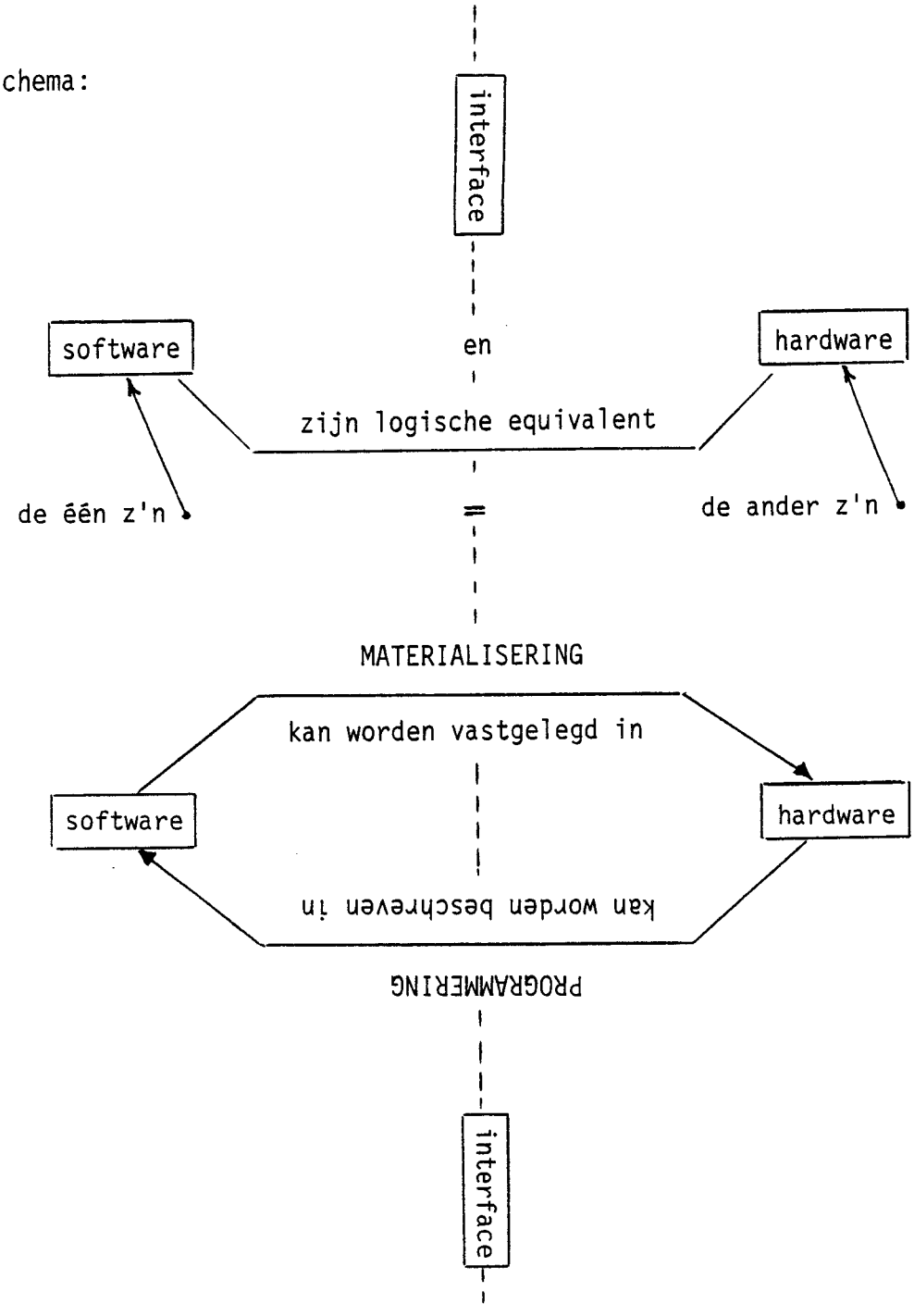
Er is echter een interpretatie-probleem: de betekenis van de definities hangt af van het gezichtspunt dat men inneemt, door de manier waarop men "kijkt". Een hardnekkige hardware-adept kan volhouden dat elk stukje "software" ook in een stilstaande machine zichtbaar gemaakt kan worden (bijv. individuele bits met een magnetische loep of zo). Een even hardnekkige software-adept kan tegenwerpen dat elke hardwarecomponent, hoe elementair of geïntegreerd ook, pas functioneel betekenis krijgt als het werkt. De tegenstelling hardware-software vertoont enige overeenkomst met die van materie en geest.

Het is dan ook duidelijk dat de definities pas hanteerbaar worden als men het eens is geworden over een standpunt. Anders gezegd: de scheidingslijn tussen hardware en software is flexibel. Nog wat pregnanter: de een z'n hardware is de ander z'n software - hardware en software zijn logisch equivalent en zelfs op een bepaalde manier uitwisselbaar. Van praktisch belang is dan ook alleen de scheidingslijn.

Men stelt ahw. bij conventie de interface vast (iets dat overigens zeer nauwkeurig moet gebeuren en veelal gedetailleerde documentatie vereist). Wat nu "vóór" de interface ligt, noemt men software (alles dat men zelf kan programmeren), wat er "achter" ligt heet hardware (alles dat gegeven is, onveranderbaar is vastgelegd, op een of andere wijze is gematerialiseerd).

Hardware kan geheel worden beschreven als software (dit is dan uiteraard een functionele beschrijving), omgekeerd kan software worden vastgelegd ("bevroren") in hardware.

In schema:



Het black-box model is hier gemakkelijk in terug te vinden: alles dat wij als informaticus aan een machine, vanuit een bepaald standpunt, (kunnen/willen) begrijpen, noemen wij software - alles dat wij als vast gegeven (kunnen /willen) aanvaarden, zit "in" de black-box en noemen we hardware. De interface in dit model is de "buitenkant" van de black-box: z'n gebruiksaanwijzing (documentatie!), datgene dat we nog net (kunnen/willen) begrijpen.

Een belangrijke informatica-stap is: het verschuiven van een interface:

A) | Het openen van een black box om te kijken wat er in zit, en vooral hoe het werkt. We stuiten dan bijna altijd op weer een nieuwe black box, een nieuwe interface dus. De uiterste interface - waarachter het ophoudt informatica te zijn - is de electronica.

S) | Het afsluiten (afschermen) van een samenhangend stuk software, zodat een nieuwe interface ontstaat (een nieuwe black box die de oude omvat).

Een A-stap is analyse van een gegeven systeem; een S-stap is syntese van een nieuw systeem. Merk op, dat het beoordelen van een gegeven software-systeem en het verifieren van een gegeven programma evenzeer A-stappen zijn als het analyseren van de documentatie van een of ander stuk hardware. Omgekeerd is het bouwen van een compiler of een operating system evenzeer een S-stap als bijvoorbeeld het ontwerpen van een chip. Merk vooral op dat een compiler (assembler) tot taak heeft een gegeven programma te vertalen naar een interface van een lager nivo: vertalers reduceren van een hogere interface naar een lagere - en besparen ons in feite het zelf volbrengen van een A-stap.

Een andere benadering van de relatie software-hardware volgt uit onderstaande definities van nog enkele belangrijke grondbegrippen:

(sequentieel) proces = | de opeenvolging (in de tijd) van welgedefinieerde, discrete data-manipulaties.

Karakteristieke eigenschappen van (informatica-)processen:

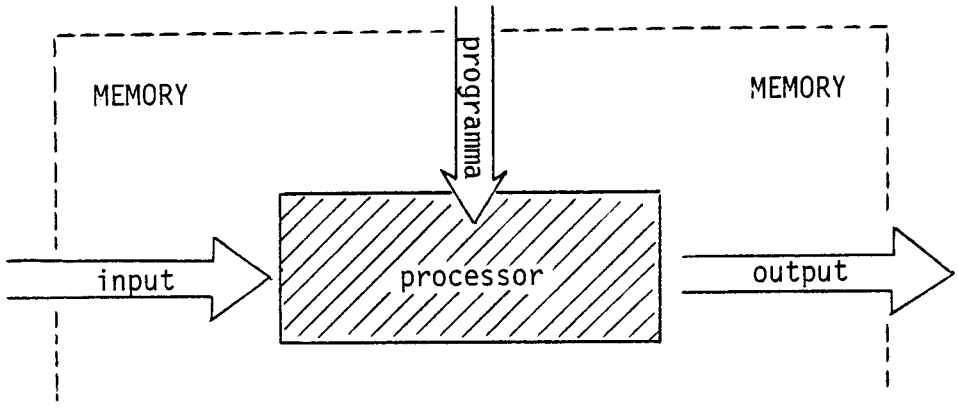
- | het effect ervan is onafhankelijk van de executietijd (het kan uiteraard snel of langzaam gaan, maar dat rekenen we niet mee als we het hebben over het "effect van een proces"),

- | het effect is geheel bepaald door de gemanipuleerde data (uiteindelijk de input dus) - anders gezegd: verandering van input impliceert in het algemeen verandering van output (de output is in feite het "effect"),

- | de data-manipulaties bestaan uit discrete stappen (in de tijd continue veranderingen van de data horen in een informatica-proces niet thuis).

De onafhankelijkheid van de executietijd moet men goed begrijpen: iedere informaticus is uiteraard geïnteresseerd in de mogelijkheid van het versnellen van processen, maar het is dan strikt genomen niet het proces dat wordt veranderd, dan wel z'n specificatie (programma) of de hardware waarop het werkt.

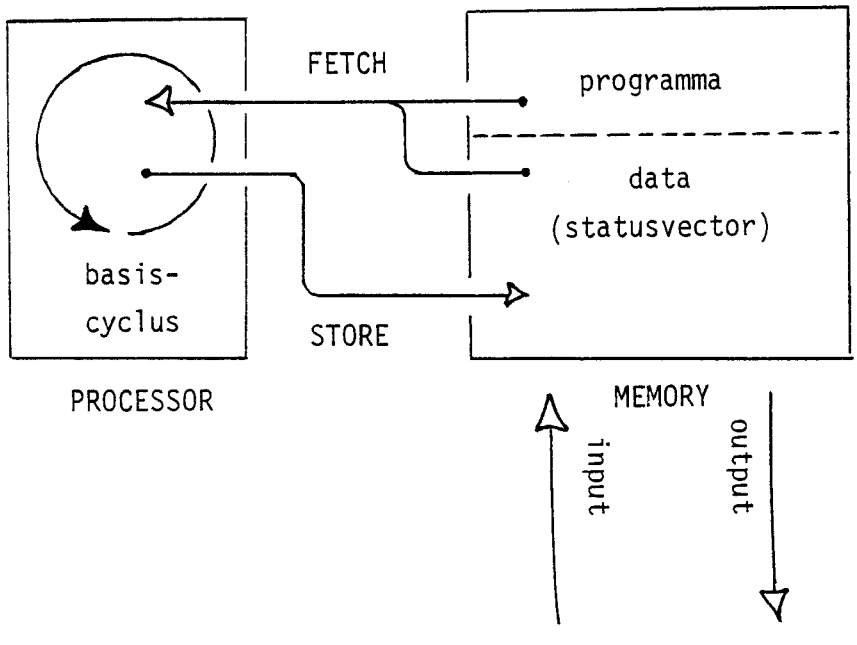
Een proces { wordt gedefinieerd door een programma (software)
wordt gerealiseerd door een processor (hardware)



Dit is opnieuw het black-box model: een processor (hardware die we niet hoeven te begrijpen) doorloopt een programma (software die we goed moeten begrijpen).

Een programma is in principe (eigenlijk per definitie) invariant: de veranderlijke bestanddelen van een proces (de gemanipuleerde data) noemt men wel de statusvector.

Programma + statusvector resideren in een geheugen ("memory"). Essentieel is dan ook dat een processor communiceert met een geheugen. We hebben het geheugen-gebied hierboven schematisch aangegeven - of men het geheugen opvat als een component van de processor of niet, is veelal een kwestie van smaak; het is echter zinvol ze te scheiden (wat wij ook altijd zullen doen). Aldus komen we tot een iets meer gedetailleerd schema:



De basis-acties voor communicatie tussen processor en geheugen:

FETCH ("lezen") en
STORE ("schrijven")
van een gegeven uit/in het geheugen.

De werking van een processor is vastgelegd in zijn basis-cyclus: een cyclisch proces(!) dat zijn opdrachten èn data uit het geheugen leest (FETCH) en data in het geheugen kan schrijven (STORE). De basis-cyclus zelf (de hardware van het systeem) kan eventueel worden beschreven door een programma, dat dan nog net tot de interface van het systeem gerekend kan worden.

Het opmerkelijke van dit black-box model is, dat het geldt voor iedere interface-level:

- langs de electronica interface zijn programma èn de data èn de basis-cyclus die volgens het programma op de data opereert, toestanden ("states") van flipflops (in processor en geheugen),
- langs de hardware interface zijn het programma en de data vastgelegd in bits, nibbles, bytes, (halve, hele en dubbele) machinewoorden, en de basis-cyclus kan (in principe) worden beschreven als een altijd aanwezig ("vast-bedraad") programma dat het gegeven programma executeert,
- langs elke hogere interface zijn programma en data vastgelegd door een programmeertaal, waarin (althans in principe) ook de basis-cyclus kan worden beschreven.

Vooral dit laatste is van belang. In feite impliceert dit dat iedere interface "boven de hardware" is bepaald door de gehanteerde programmeertaal:

programmeertaal en interface zijn in essentie identiek

Deze observatie stelt ons in staat interfaces vanuit één concept te behandelen: we leggen een interface vast door zijn programmeertaal vast te leggen. De programmeertaal van de hardware-interface wordt wel de "machinetaal" genoemd - wij zullen meestal een nivo hoger gaan zitten, dwz. van een iets hanteerbaardere programmeertaal uitgaan: de assembler-taal die de assembly-language interface bepaalt.

Nauwkeurige interface-definitie is een van de meest aktuele problemen in de computer-architektuur, en in meer algemene zin in de informatica. Interfaces zijn over het algemeen complexer dan "alleen maar een programmeertaal" (nauwkeurige definitie van een programmeertaal is trouwens ook allesbehalve triviaal). Een modern woord voor "interface" op de gebruikers-nivo's is "programming environment".

3.1 INTERFACE LEVELS

3.1.0 Een voorbeeld

Nu we hebben gezien dat interface en programmeertaal in feite identiek zijn, zullen we demonstreren hoe een programmeertaal (interface) kan worden beschreven in (gereduceerd tot) een andere, simpeler, taal. De taal (interface) waarnaar wordt gereduceerd noemen we de "onderliggende" taal - het lagere nivo dus; de taal (interface) die gereduceerd wordt, is dan het hogere nivo.

We illustreren een en ander aan een concreet voorbeeld: een algoritme die we op vier taal-(interface-)nivo's bekijken. Bij elke (hypothetische) processor geven we de basis-cyclus in een enigszins verwant jargon (als we hier volstrekt precies zouden willen zijn, zouden we zeer veel papier nodig hebben). De vier voorbeeld-talen zijn:

ALGOL68 $\supset$ SMALGOL $\supset$ FANGOL en FANLAN

"SMALGOL" en "FANGOL" zijn simpeler vormen van ALGOL68, terwijl FANLAN de assemblertaal is van de FANCY. FANLAN wordt behandeld in hoofdstuk 6, men neme de in 3.1.4 gegeven tekst voor wat het is: illustratie (de puntjes op de i komen later vanzelf).

We beginnen met het programma te illustreren in menselijke (ietwat mathematische) taal:

De input is een rij reële getallen $x_i \neq 0.0$, afgesloten door het getal 0.0 Het programma berekent en drukt af:

output1: het aantal input-data $\neq 0.0$ (de hoogste waarde van i dus),

output2: hun gemiddelde:

$$\overline{x} = \frac{\Sigma\, x_i}{n}$$

output3: de standaard-deviatie:

$$\sigma = \sqrt{\frac{\Sigma\, x_i^2}{n} - \overline{x}^2}$$

De statistische merites van dit programma zijn irrelevant voor wat we willen demonstreren (maar het schijnt best in orde te zijn).

3.1.1 de ALGOL68-interface

In ALGOL zijn de data geklassificeerd naar disjuncte verzamelingen, meestal "types" genoemd - in ons voorbeeld zijn alleen de types INT en REAL van belang. De data-types definieren de betekenis van operaties zoals +, -, *, /, read, print etc. De basis-componenten van het programma zijn <phrase>s. De data-typering en de grote expressiekracht van de <phrase>s zijn karakteristiek voor een hogere programmeertaal.

```
begin loc int  numb:=0 ;
      loc real  sum:=0.0, squm:=0.0,
                mean, sqean, standev, next ;
      while  read(next); next/=0.0
         do  sum := sum + next ;
             squm :=squm + next * next ;
             numb :=numb + 1
         od ;
         if numb /= 0
      then mean := sum/numb ;
           sqean :=squm/numb ;
           standev := sqrt(sqean-mean * mean) ;
           print((numb,mean,standev))
      else print("empty input")
         fi
end
```

We kunnen de basis-cyclus van de ALGOL68-machine in principe ook in ALGOL68 beschrijven. Een volledige beschrijving zou betekenen dat we een complete ALGOL68-compiler of -interpreter in ALGOL68 zouden moeten schrijven, wat kan, maar géén kleine opgave is. We volstaan met het meest globale schema van een dergelijke compiler/interpreter. We zullen zien dat - naar mate we de interface verder reduceren - de basis-cyclus in deze globale gedaante steeds nauwkeuriger wordt.

De basis-cyclus van de ALGOL68 processor:

```
while  algol68 processor busy
   do  fetch phrase to be elaborated ;
       define next phrase ;
       fetch operands required ;
       elaborate phrase
   od
```

De vijf <identifier>s in bovenstaande <loop-clause> zijn uiteraard aanroepen van procedures (die elders gedeclareerd zijn: ahw. de "hardware" achter de interface).

We hebben de <phrase>s in de programmatekst in kaders gezet. Er zijn dus in eertse instantie slechts vier <phrase>s. Echter(!) de ALGOL68-processor gaat in de opdracht 'elaborate phrase' in recursie en recurreert zo vaak als nodig is, om de basis-ALGOL <phrase>s te bereiken. In de SMALGOL- en FANGOL-versies van het programma zullen we iets te zien krijgen van deze basis-ALGOL <phrase>s.

3.1.2 de SMALGOL-interface

In SMALGOL worden de stuurconstructies <loop-clause> en <choice-clause> gereduceerd tot de veel primitievere <goto-statement>s (die conditioneel kunnen zijn). De <phrase>s worden hiermee gereduceerd tot enkelvoudige <statement>s. SMALGOL heeft ongeveer de expressiekracht van een taal als FORTRAN. Merk op, dat we de aritmetische expressies nog niet reduceren: we gaan er dus nog van uit dat de SMALGOL-processor een behoorlijk krachtig reken-orgaan heeft, dat met name ook in staat is verschillende datatypes te onderscheiden en te manipuleren.

In de SMALGOL-basiscyclus zien we dat een nogal vage opdracht als 'define next phrase' nu veel nauwkeuriger gedefinieerd is. Het programma bestaat uit "lines" en iedere "line" bevat precies één <statement>. Dit is een eerste stap in de richting van de werkelijkheid langs een lagere interface.

Het leven zal nog veel primitiever worden.

Merk op, dat de declaraties in SMALGOL nu niet meer als <statement>s kunnen worden gezien: het zijn aanwijzingen voor geheugen-reservering en data-types. Het initialiseren van deze grootheden moet nu via expliciete <statement>s geschieden.

```
          begin  loc int  numb,
                 loc real sum, squm, mean, sqean, standev, next ;
line 1:   init:  numb:=0 ;
line 2:           sum:=0.0 ;
line 3:          squm:=0.0 ;
line 4:   again: read(next);
line 5:          if next=0.0 then goto ready fi ;
line 6:           sum := sum + next ;
line 7:          squm :=squm + next * next ;
line 8:          numb := numb + 1 ;
line 9:          goto again ;
line10:   ready: if numb=0 then goto empty fi ;
line11:          mean := sum/numb ;
line12:          sqean:=sqean/numb ;
line13:          standev := sqrt(sqean - mean * mean) ;
line14:          print(numb) ;
line15:          print(mean) ;
line16:          print(standev) ;
line17:          goto finish ;
line18:   empty: print("empty input") ;
line19:   finish: stop
           end
```

```
while  smalgol processor busy
   do  statement := text[line number] ;
       line number := line number + 1 ;
       fetch operands required ;
       execute statement
   od
```

3.1.3 De FANGOL-interface

In FANGOL worden ook de <expression>s gereduceerd. Alle (INT- en REAL-) aritmetiek loopt over de flipflopregisters N (INT-aritmetiek) en R1, R2, R3 (REAL-aritmetiek). Aan deze registers zijn de flipflops ZERO en OVERFLOW gekoppeld, die steeds op false staan, tenzij een rekenresultaat =0(=0.0) is, resp. overflow geeft.

We geven de declaraties in het FANGOL-programma om het nog steeds zo goed mogelijk ALGOL68 te laten zijn; we veronderstellen dat het zij-effect op ZERO en OVERFLOW impliciet is met iedere ("laatste") reken-operatie (in N, R1, R2, R3 etc.). De procedures READREAL, PRINTINT, PRINTREAL en PRINT TEXT kan men beschouwen als (tot de FANGOL-"hardware" behorende) standaard-operaties, evenals := (voor FETCH en STORE) en +:=, -:=, *:=, /:= en SQUAROOT voor de gesuggereerde aritmetische operaties. HALT beschouwe men als een specifieke procedure-aanroep waarop de FANGOL-processor stopt, althans met de executie van het onderhavige programma.

De successieve <statement>s corresponderen vrijwel één-op-één met de "echte" FANCY-machine opdrachten (zie hiervoor de FANLAN-versie in 3.1.4). De programmatekst illustreert overigens heel aardig, hoe een hogere programmeertaal óók gebruikt kan worden voor "low level" programmeren. We hebben systematisch hoofdletter-identifiers gebruikt voor variabelen die via een "hardware"-register worden gespeeld. Het enige verschil tussen FANGOL en de FANCY-werkelijkheid is nog dat in FANGOL de registers "type-gevoelig" zijn, wat in FANLAN niet meer het geval is.

```
                begin  boolregister ZERO, OVERFLOW ;
                        intregister N ;
PC:                     realregister R1, R2, R3 ;
 1          init:       N:=0 ;
 2                      R1:=0.0 ;
 3                      R2:=0.0 ;
 4          again:      READREAL(R3) ;
 5                      if ZERO then goto ready fi ;
 6                      R1+:=R3 ;
 7                      R3*:=R3 ;
 8                      R2+:=R3 ;
 9                      N +:= 1 ;
10                      goto again ;
```

```
11      ready:   R1/:=N ;
12               if OVERFLOW then goto empty fi ;
13               PRINTINT(N) ;
14               PRINTREAL(R1) ;
15               R2/:=N ;
16               R1*:=R1 ;
17               R2-:=R1 ;
18               SQUAROOT(R2) ;
19               PRINTREAL(R2) ;
20               goto finish ;
21      empty:   PRINTTEXT("empty input") ;
22      finish:  HALT
          end
```

De output-statements staan op een iets andere plaats in de tekst teneinde een optimaal registergebruik mogelijk te maken; dergelijke overwegingen zijn inherent aan "low-level programming". Merk ook op, dat de FANGOL-tekst nauwelijks omvangrijker is dan de oorspronkelijke ALGOL-tekst (maar wel moeilijker te lezen).

Voor de beschrijving van de basis-cyclus van de FANCY-processor (daar zitten we nu al vlakbij) introduceren we twee "control-registers": IR (Instruction Register) en PC (Program Counter). M is het geheugen-array van de FANCY; het beschouwde FANGOL-programma wordt geacht te zijn gecodeerd in opeenvolgende M-cellen: één FANGOL-instructie per cel. Een "cel" kan ongelijk van lengte zijn (afhankelijk van het type opdracht bijv.), maar men kan er voorlopig vanuitgaan dat 1 cel = 4 bytes (waar dat niet het geval is, regelt de FANCY-hardware de oneffenheden wel achter de schermen, zie ook 3.1.4).

```
while  fancy processor busy
  do   IR  := M[PC] ;
       PC +:= 1 ;
       fetch operand if required ;
       execute IR
  od
```

3.1.4 De FANLAN-interface

In tegenstelling tot ALGOL, SMALGOL en ook nog FANGOL, is FANLAN een zg. "typeless language" - dwz: de data zijn bitrijen en als zodanig vrij interpreteerbaar. In iedere opdracht beslist de zg. operation-key over de interpretatie (over het "type") van de operand(en): bijv. ADD voor INT-optelling, ADDF voor floating-point optelling etc. Aan de operation-key kan nog een size-cijfer worden toegevoegd, dat aangeeft hoeveel bytes bij de opdracht betrokken zijn. Deze size kan 1, 2 of 4 zijn; het ontbreken van een expliciet gegeven size betekent vrijwel altijd dat de operand een 4byte is ("default"-option 4) - dat is de standaard-woordlengte van de FANCY.

De FANCY heeft acht data-registers (D0, D1, D2, D3, D4, D5, D6 en D7). We gebruiken in ons voorbeeld D0 als "INT"-register (N in FANGOL) en D1, D2 en D3 als "REAL"-registers (R1, R2 en R3 in FANGOL). De telling van de input-data (in N = D0) doen we in een 2byte, de REAL-aritmetiek uiteraard in 4bytes.

De procedure (ALGOL, SMALGOL, FANGOL) wordt in FANLAN gerealiseerd door een subroutine-CALL, voorafgegaan (in de tijd, niet noodzakelijk tekstueel) door de overdracht van parameter(s) via de parameter-stack - dit is wat de MACRO-opdracht PUSH doet. Een eventueel door de subroutine berekend (al dan niet samengesteld) resultaat wordt eveneens op de stack gedeponeerd en kan met de MACRO-opdracht POP worden opgevraagd. Subroutinenamen en MACROnamen in hoofdletters maken deel uit van de standaard-firmware van de FANCY - de assembler maakt overigens geen verschil tussen hoofdletters en kleine letters, dit is dus puur een conventie (voor betere leesbaarheid): programma-identifiers schrijven we in kleine letters.

Langs de FANLAN-interface moet men de standaard-input en -outputkanalen (kaartlezer, lineprinter etc) expliciet openen; hiervoor zorgen de firmware-subroutines BUFFIN en BUFFOUT - het sluiten van de kanalen is een van de nevenacties van de MACRO-opdracht STOP.

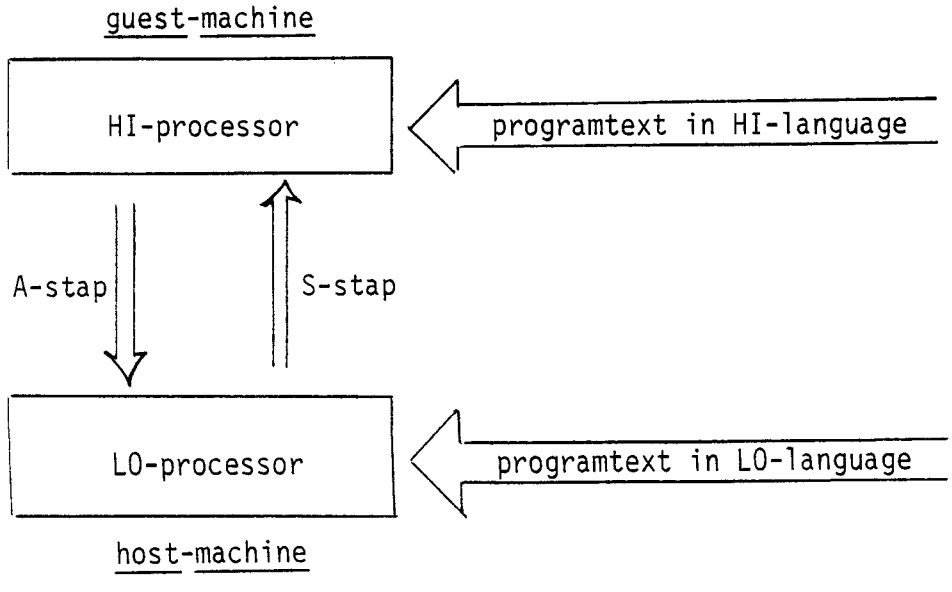
Voor verdere details en explicatie verwijzen we naar de hoofdstuk 4, en DEEL II . Men kan het FANLAN-programma bijna regel-voor-regel leggen naast het FANGOL-programma, dat men rustig als gedetailleerde "comment" kan gebruiken. Waar afwijkingen zijn, geven we dit aan in een FANLAN-comment: een willekeurige tekst achter het comment-symbol "$" (tot einde-regel). Het wordt krachtig aanbevolen de FANGOL- en de FANLAN-versie zorgvuldig te vergelijken - dat is een heel leerzame stap op weg naar het assembler-programmeren.

```
.BEGIN    $ statistics program
 init:    PUSH    CR                  $ kanaalnummer Card Reader
          CALL    BUFFIN              $ open input-kanaal
          PUSH    LP                  $ kanaalnummer Line Printer
          CALL    BUFFOUT             $ open output-kanaal

          COPY 2  D0  0
          COPY    D1  0.0
          COPY    D2  0.0

 again:   CALL    REALIN
          POP     D3
          GOTO    ready   IF ZERO
          ADDF    D1  D3
          MULF    D3  D3
          ADDF    D2  D3
          ADD  2  D0  1
          GOTO    again

 ready:   PUSH    D0                  $ N moet als INT worden geprint
          FLOAT   D0                  $ zet D0 om in floating-point
          DIVF    D1  D0
          GOTO    empty   IF OF
          CALL    INTOUT              $ print N (op stack)
          PUSH    D1
          CALL    REALOUT
          DIVF    D2  D0
          MULF    D1  D1
          SUBF    D2  D1
          PUSH    D2
          CALL    SQUAROOT
          CALL    REALOUT             $ SQUAROOT-result is op stack
          GOTO    finish

 empty:   POP     D0                  $ verwijder N van stack
          PUSH    text
          CALL    TEXTOUT

 finish:  STOP                        $ sluit alle kanalen en einde

   text:  STRING  "empty input"

 .END
```

3.2 HIERARCHISCHE STRUCTUUR

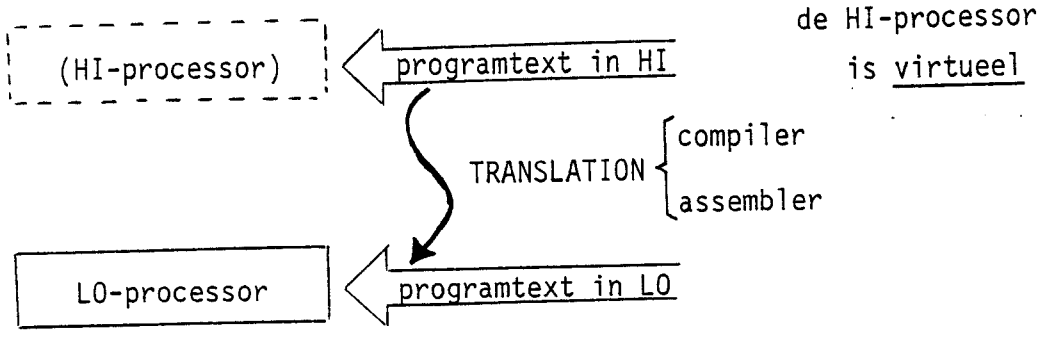
In de interface-hierarchie hebben we steeds te maken met een "high-level" processor waar - bij nadere analyse - een "lower-level" processor achter schuil blijkt te gaan. In ons voorbeeld: ALGOL68 ==> SMALGOL ==> FANGOL ==> FANLAN ==> en verder. Het belang van deze hierarchie kan niet gemakkelijk worden overschat. Niet alleen krijgen (en houden) we op deze manier greep op het vaak zeer gecompliceerde gebeuren op een computersysteem (het is een "verdeel-en-heers" strategie, de verwantschap met gestructureerde programmering is duidelijk), maar het stelt ons ook in staat om zeer verschillende systemen op elkaar af te stemmen, vanuit één gezichtspunt te bekijken, en zelfs op elkaar te simuleren. Het greep krijgen op complexiteit is echter het belangrijkste. Een aardige demonstratie hiervan zien we in de FANLAN-versie van ons voorbeeld: de transput wordt daar nog vrij gemakkelijk geregeld via de subroutine-aanroepen BUFFIN, REALIN,REALOUT, TEXTOOUT, enz. De firmware die hierachter schuil gaat (buffer-administratie, conversie en deconversie, synchronisatie) is echter van een aanzienlijke omvang en gecompliceerdheid (zie hiervoor DEEL II).

In de directe relatie van twee interfaces hebben we steeds te maken met twee processoren die we hier HI en LO zullen noemen: LO is de black-box die zichtbaar wordt als we de black-box HI openen en analyseren (een A-stap doen van HI naar LO). Men noemt de HI-processor wel de "guest-machine" en de LO-processor de "host-machine": een terminologie die duidelijk is ontleend aan de simulatie van de ene machine op de andere, maar ook heel algemeen gebruikt kan worden:



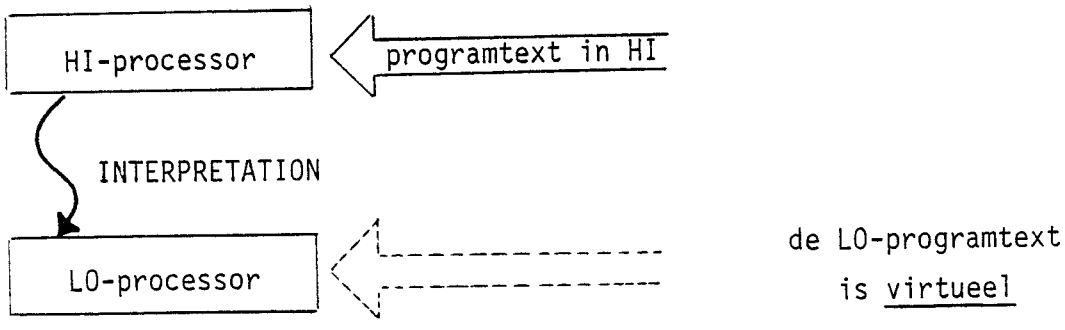
Van belang is nu dat er twee manieren zijn waarop de relatie tussen HI en LO geregeld kan zijn:

1) de vertaal-structuur:



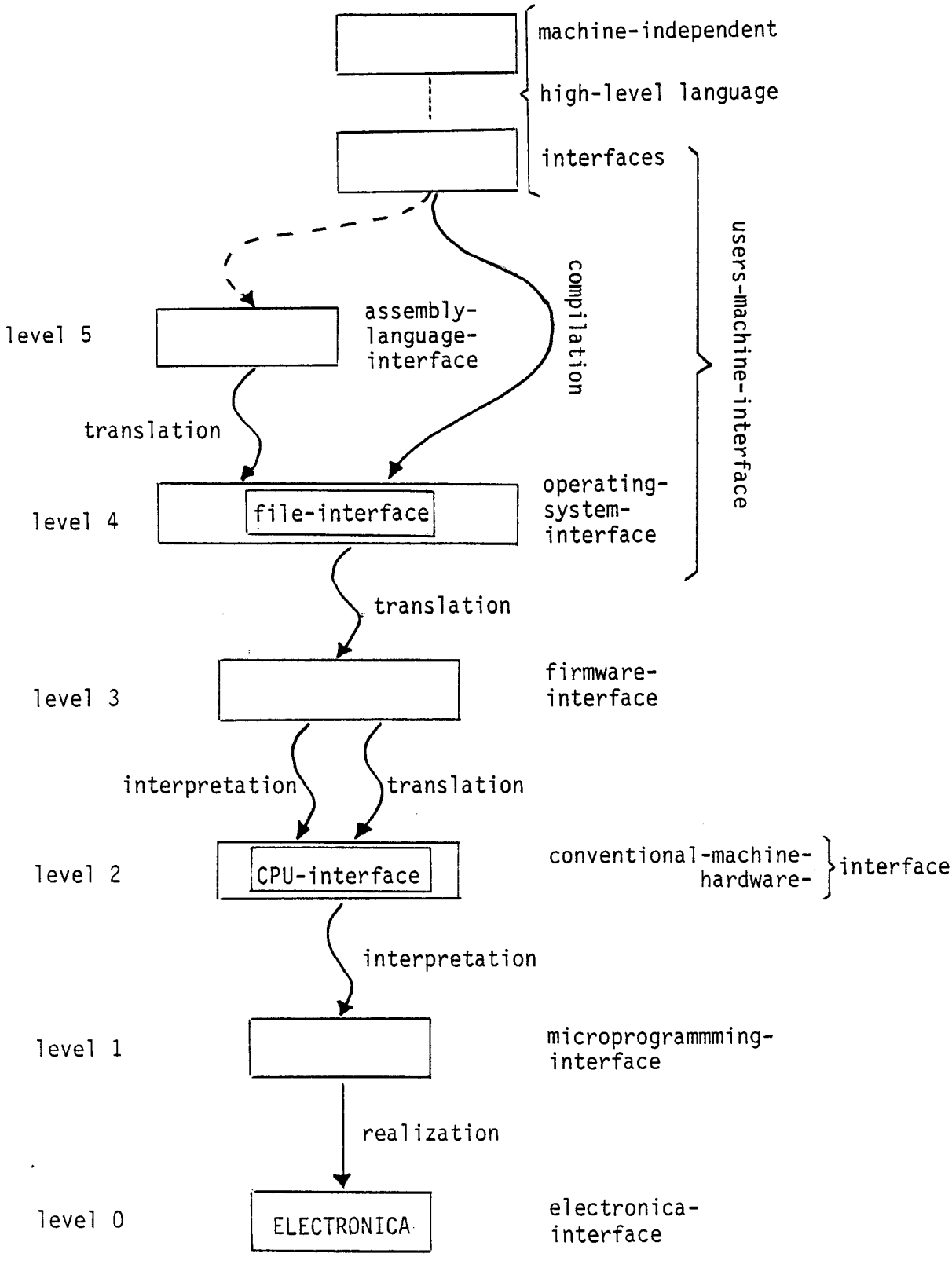
- Het te verwerken programma wordt in een afzonderlijk proces door een compiler of assembler omgezet in een LO-level programmatekst die geschikt is voor de beschikbare LO-processor
- Een HI-processor is niet in concreto aanwezig. De vertaler is uiteraard een LO-program.

2) de interpretatie-structuur:



- De HI-processor is als een LO-level program aanwezig
- De LO-processor executeert de HI-processor
- De HI-processor op zijn beurt executeert de aangeboden programtext in HI.

De in 0.3 genoemde interfaces (waarlangs we de FANCY-configuratie zullen bekijken) kunnen we nu alsvolgt in kaart brengen:



De microprogramming-interface (in het schema tussen level0 en level2) behoeft nog enige toelichting. Over het algemeen kan men de "conventional-machine" (hardware-)interface nog verklaren (analyseren) in termen van enkele zeer elementaire bitmanipulaties: we interpreteren de hardware-interface als een HI-processor die werkt op een LO-processor (de microprogramming interface). Deze interpretator is op de meeste machines als een in de electronica verborgen (misschien op read-only geheugen vastgelegd) micro-programma aanwezig. Op sommige machines (bijv. de BURROUGHS1700) zijn de microprogramma's nog uitwisselbaar en zelfs tot op zekere hoogte programmeerbaar - anders gezegd: het kan zijn dat er nog een volledig processor-level is tussen de CPU-interface en de electronica. We hebben dit voor de FANCY-architectuur niet aangenomen, zodat men daar de microprogramming-interface kan beschouwen als het "vast-bedrade gezicht-naar-boven" van de electronica (zie ook 4.0).

Men kan nog twee soorten interfaces onderscheiden:

1) gesloten interfaces ("opaque interfaces")

   Bij het werken (dwz. programmeren) langs een gesloten interface kan men op geen enkele wijze direct bij de LO-processor komen. Er zitten geen "vensters" in de interface. Een (goede) high-level programmeertaal is gesloten (een ALGOL-programmeur programmeert alleen in ALGOL - dwz. hij kan geen ALGOL-<phrase>s larderen met bijv. FANLAN-phrases)

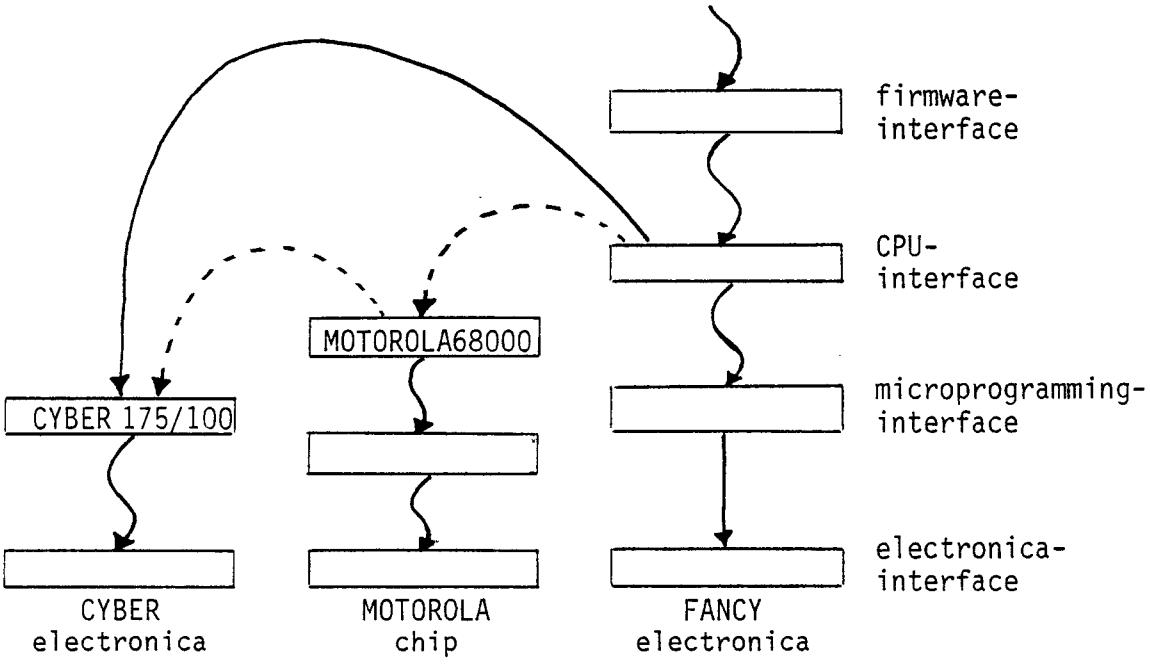
2) open interfaces ("transparent interfaces")

   De interface heeft ahw. een aantal "vensters" op de LO-processor - anders gezegd: bij het programmeren van de HI-processor kan men (meestal met mate) ook direct beschikken over de LO-processor. In het algemeen creëert een subroutine- (procedure-) library (een "package") een vrijwel geheel transparante interface: de gebruiker ervan programmeert in dezelfde programmeertaal als waarin de routines geschreven zijn.

Bij transparante interfaces kan men een (ietwat filosofisch) probleem krijgen met de hiërarchie - wat is nu "HI" en wat "LO"? Zowel de firmware als de operating system software kunnen (en zullen over het algemeen) in FANLAN geschreven zijn, anderzijds zal iedere FANLAN-programmeur over operating system- en firmware-routines kunnen beschikken. Wij beschouwen daarom de FANLAN-interface èn de operating-system-interface èn de firmware-interface als transparant: langs alle drie interfaces "ziet" men de CPU-interface. De hiërarchie is bepaald door de overweging dat er zonder firmware geen operating system kan bestaan, en zonder operating system nauwelijks assembler-programmering

mogelijk is. Bovendien zullen zowel de firmware- als de operating-system programmeur vaak gebruik maken van specifieke ("privileged") instructions die de (gewone) FANLAN-programmeur niet ten dienste staan. Vandaar onze levels 2-3-4-5: door 3, 4 en 5 is level2 echter zéér zichtbaar!

In onderstaand schema zien we hoe de FANCY als een guest-machine op de MOTOROLA68000-chip gerealiseerd zou kunnen worden, en deze op zijn beurt weer op de CYBER175/100. Efficienter is uiteraard direct van de FANCY naar de CYBER. Dergelijke simulatoren worden natuurlijk via de hogere interfaces van de host-machine geprogrammeerd, vandaar (in onderstaand schema) de van boven ontspringende pijlen:



3.3 MACHINE CONFIGURATIES

3.3.0 De FANCY-configuratie

Hiernaast een plattegrond van de FANCY-machine-configuratie. Dit is het beeld dat men ongeveer krijgt langs de operating-system interface. De voornaamste funktie van een operating system is het creëren van een interface waarlangs alle peripherals ("randapparatuur": de I/O apparaten, achtergrondgeheugens etc) zoveel mogelijk op elkaar lijken en ook op vergelijkbare wijze worden "aangesproken". De individuele verschillen (die zeer groot kunnen zijn) worden door het operating system als het ware weggemasseerd, vandaar dat de plattegrond een zeer symmetrische aanblik geeft - de "werkelijkheid" langs een lagere interface ziet er heel anders uit.

Elk blok stelt een apparaat voor. De verbindingslijnen zijn data-lines - kabels waarover bits getransporteerd kunnen worden. De meeste data-lines zijn "bussen", kabels waarin een groot aantal parallelle draden zijn gebundeld: in de central site is het normaal dat kbytes in hun geheel (de individuele bits dus parallel) worden getransporteerd. Pas achter de modems kan de communicatie serieel (bit-na-bit) worden.

Het Random Access Memory (RAM) kan bestaan uit twee delen:
- Main Memory: altijd aanwezig (maximaal 16 Mbytes)
- extended RAM: optional (goedkoper, maar langzamer: maximaal (4K-16)Mbytes).

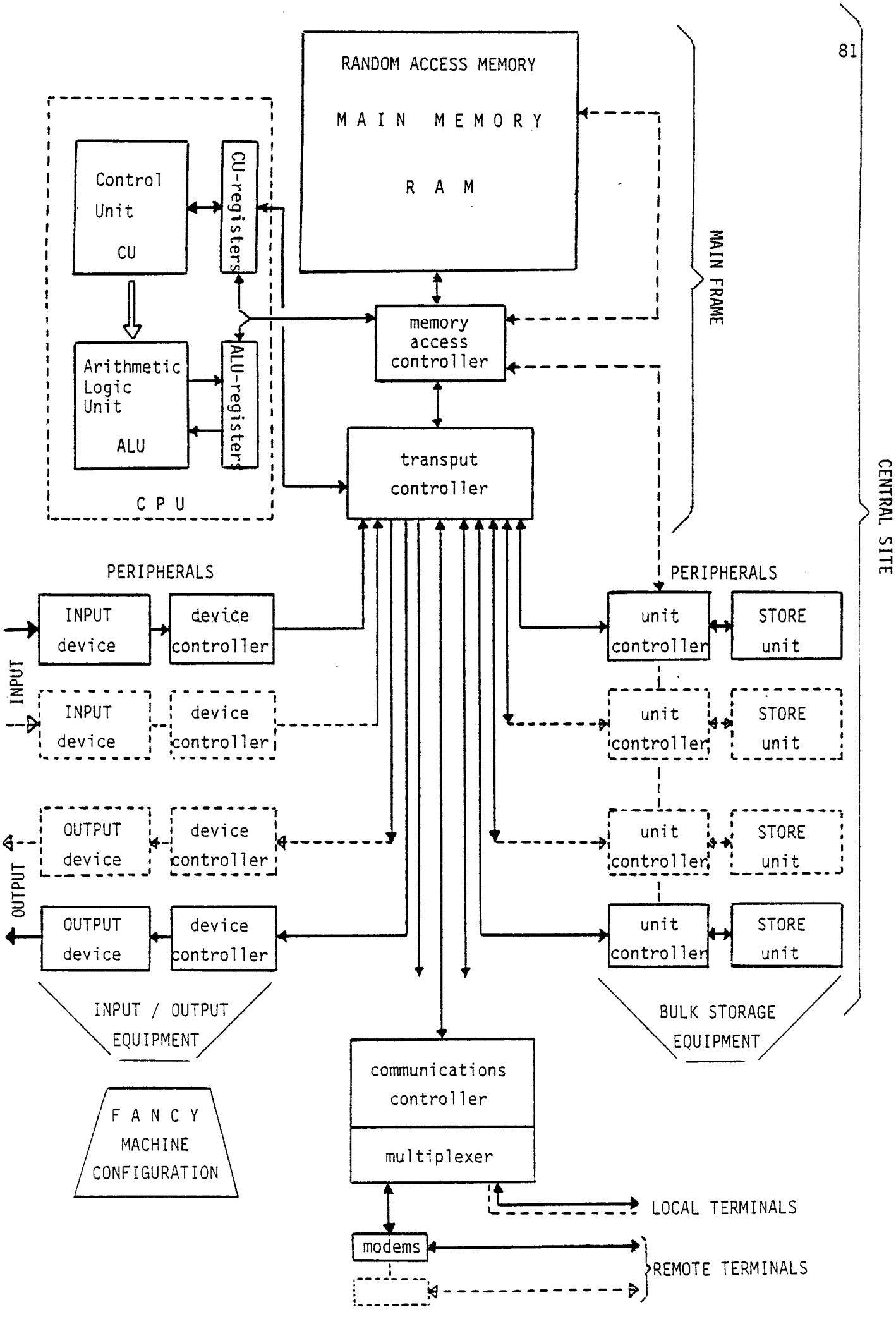
De FETCHes en STOREs van/naar RAM worden geregeld door de memory access controller (ihb. prioriteitsregeling). Zie hiervoor DEEL II.

De CPU (Central Processing Unit) bestaat uit een CU (Control Unit) die de ALU (Arithmetic Logic Unit) stuurt op grond van de opdrachten in het programma. Zie verder hfdst.4.

De transput controller coördineert (onder globale regie van de CPU) alle verkeer tussen het Random Access Memory en de peripherals.

De peripherals bestaan in twee categorieën:
- Input/Output equipment (kaart- en ponsbandlezers, regeldrukkers, plotters etc)
- Bulk Storage equipment (magneetband- en schijven-eenheden, trommelgeheugens etc)

De communications controller coördineert (onder globale regie van de transput-controller) alle verkeer van en naar (locale en op afstand geplaatste) terminals. De modems (modulator/demodulator) past de signalen aan voor lange-afstand-communicatie (telefoonlijn, hoogwaardige communicatielijnen, draadloos etc.).


RANDOM ACCESS MEMORY
MAIN MEMORY
RAM
Control Unit
CU
CU-registers
Arithmetic Logic Unit
ALU
ALU-registers
CPU
memory access controller
transput controller
MAIN FRAME
CENTRAL SITE
PERIPHERALS
INPUT device
device controller
INPUT
OUTPUT device
OUTPUT
INPUT / OUTPUT EQUIPMENT
PERIPHERALS
unit controller
STORE unit
BULK STORAGE EQUIPMENT
communications controller
multiplexer
LOCAL TERMINALS
modems
REMOTE TERMINALS
FANCY MACHINE CONFIGURATION

3.3.1 Peripherals en terminals

Uit architectuur-oogpunt zijn de technische (hardware-)aspekten van de randapparatuur nauwelijks interessant - waar het om gaat is, wat je ermee kunt doen en wat dat betreft zorgt het operating system (ondersteund door een groot aantal specifieke firmware-packages) voor een verregaand geuniformiseerde interface, waarlangs we in feite alleen nog maar input-kanalen, output-kanalen en geheugen-kanalen onderscheiden. De enige individuele gegevens die nog wel van belang kunnen zijn, hebben betrekking op snelheden van informatie-overdracht en hoeveelheid informatie. We volstaan met wat globale en gemiddelde getallen - er is uiteraard een sterke prijs/prestatie spreiding.

INPUT/OUTPUT apparatuur

Afgezien van speciale input-devices zoals optische en magnetische lezers, gaat alle input nog steeds via ponskaarten en ponsbanden of direct (zonder tussen-medium) vanaf het toetsenbord van een terminal. Output geschiedt via regeldrukkers, eventueel ponskaarten en ponsbanden, grafische plotters en verder de "scopes" van terminals. Afgezien van grafische output, is alle transput in principe character-transput.

- CR Card Reader
  leest 600-1200 kaarten/min. = 48000-96000 char/min.

- CP Card Punch
  ponst ca 250 kaarten/min. = 2000 char/min

- PTR Paper Tape Reader
  leest 250-1000 char/min.

- PTP Paper Tape Punch
  ponst ca 150 char/min.

Ponsband wordt niet veel meer gebruikt voor character-transput. De belangrijkste toepassingen liggen op het gebied van automatische registratie van meetgegevens en het sturen van machines en gereedschap (draaibanken, frazers etc).

- LP Line Printer
  De snelheid is sterk afhankelijk van het soort characters dat wordt afgedrukt, bijv. alleen cijfers en enkele getal-tekens, één of twee lettertypen, het wel of niet gebruikmaken van een "overprinting" mogelijkheid (samengestelde tekens).

  300-1500 regels/min. = (bij 120 char/regel) 36000-180000 char/min.
  (bedenk dat regels zelden "vol" zullen zijn, en dat de formulierbreedte ook sterk kan uiteenlopen op verschillende lineprinters).

- PL PLotter

  Grafische plotter: tekent in 8 richtingen met stapjes van 0.1-0.14 mm.

  200-800 st/min.

STORE-equipment ("massa-geheugens")

Vooral de grote systemen (zie 3.3.2) hebben zeer veel geheugenruimte nodig voor het opslaan van firmware- en operating-system modulen, vertalers, bibliotheken, vaste programmatuur, databanks etc. In gebruik (maar snel in onbruik rakend) zijn nog steeds de "fast drums" ("trommel geheugens"), vooral UNIVAC werkt hier nog altijd mee. Algemeen in gebruik zijn:

- DU Disk Unit

  Een disk is een grammofoonplaatachtige metalen schijf waarop informatie magnetisch kan worden vastgelegd (en ook weer gemakkelijk kan worden uitgeveegd). De informatie is opgeslagen op tracks (concentrische cirkels), zowel boven- als onder-op de schijf. De schijven roteren snel en worden met read/write-heads uitgelezen/beschreven. Deze read/write-heads kunnen (nogal snel) van track naar track worden bewogen, de rotatie van de schijf zorgt voor de rest.

  aantal schijven per DU = 2-20 (4-8 is normaler)
  aantal tracks per oppervl = ca 200
  aantal sectoren per track = 10-100 (firmware-afhankelijk)
  aantal bytes per sector = 128-1024 (firmware-afhankelijk)

  capaciteit: 2.5 Mbyte tot 1 Gbyte ($G=KM=2^{30}$)
  mede afhankelijk van track/sector indeling

  insteltijd read/write-heads (seek-time): 5-50 msec.
  rotatiesnelheid: ca 40 omw/sec

  Een DU kan worden beschouwd als een langzaam random-access geheugen. De adressering is echter nooit woord- (laat staan byte-) gewijs, maar altijd (sector,track); het informatietransport is dus minimaal 1 sector (128-1024 bytes). Voor het lezen van en schrijven op een DU is omvangrijke firmware noodzakelijk.

- FDU Floppy Disk Unit

  Wint vooral terrein op micros en mini's (zie 3.3.2). Te beschouwen als een kleine DU.

  capaciteit: ca 2 Mbyte

- MTU Magnetic Tape Unit

  Op een MTU kan men snel tapes verwisselen, zodat de capaciteit van MTU + operator vrijwel onbeperkt is. De moderne MTU werkt met 9-track tapes (1 parity-track). De informatie op een tape is ingedeeld in records (blokken) gescheiden door interrecord-gaps. De blok-indeling is tamelijk vrij en typisch een firmware-software aangelegenheid. Een MTU is beslist niet te gebruiken als random-access geheugen. De informatie wordt blok-na-blok sequentieel gelezen of beschreven. De enige positioneringsmogelijkheden zijn (niet op alle MTU's) het terugwinden of overslaan van een gegeven aantal blokken, eventueel (ook niet op alle MTU's) het opzoeken van een bepaalde blok-header; rewind is uiteraard altijd mogelijk.
  Een MTU kan worden beschouwd als een snel input/output-apparaat, en ook als een relatief langzaam sequentieel-geheugen van grote capaciteit.

  capaciteit: sterk afhankelijk van de dichtheid waarmee de tapes worden beschreven (800-6250 bits/inch, normaal 1600 bpi) en de blok-indeling:

  32-128 Mbytes/tape

- CTU Cassette Tape Unit

  Wint vooral terrein op micro's en mini's (zie 3.3.2). Er zijn speciale digitale cassettes, maar ook gewone cassettes worden gebruikt (de digitale informatie wordt dan wel gemoduleerd). Een cassette is uiteraard een bijzonder handig input/output medium!

TERMINALS

Een terminal is een combinatie van keyboard met scope en/of printer. Het is essentieel een combinatie van input & output-device. Typerend is, dat men de input (dwz. de informatie stroom van terminal naar central site) direct (op de scope en/of de printer) te zien krijgt. Een terminal is bij uitstek geschikt voor dialoog-communicatie met een computer-systeem, maar tegelijk ook als een privé-input/output-device van een groot systeem.

Bijzondere terminals zijn graphical terminals.

Zeer belangrijk is de mogelijkheid van een terminal met ingebouwde processor (chip): een fraaie combinatie van een zelfstandige micro-computer met een zg. intelligente terminal. Graphical terminals winnen aanzienlijk aan "power" met een ingebouwde processor.

3.3.2 Klassificatie van configuraties

Sinds de chip-processor z'n intree deed, is een merkwaardige terminologie-verwarring ontstaan. Het kleine computersysteem noemde men reeds "mini-computer", het lag dus voor de hand de 1-chip computer "micro" te noemen. De razendsnelle ontwikkeling achterhaalde echter snel de schaal-suggestie van het woord. Tegenwoordig kunnen in principe alle grote systemen - voor zover het hun main-frame betreft - worden uitgevoerd op 1 chip (de VLSI = Very Large Scale Integration, definieert zelfs, als men dat wil, de complete firmware erbij op 1 chip). Met de peripherals gaat dat niet zo gemakkelijk (de controllers leveren uiteraard geen probleem, maar een LP of een MTU plak je niet zo gemakkelijk op een postzegel. Een definitie "micro = 1-chip systeem" zou dus uiterst misleidend zijn.

De grootte ("omvang", "power") van een systeem is voornamelijk bepaald door de volgende factoren:

- diversiteit van de I/O-peripherals,
- capaciteit en snelheid van het massa-geheugen,
- snelheid, raffinement en reken-precisie van de CPU('s),
- aantal onafhankelijke gebruikers (multiprogrammering, zie ook hfdst. 7).

In het algemeen geldt: hoe "groter" een systeem, hoe universeler zijn mogelijkheden en hoe meer gebruikers er tegelijkertijd gebruik van kunnen maken.

Hoewel de terminologie nog erg aan verandering onderhevig is (en voorlopig wel zal blijven: commercial advertising speelt een hartig woordje mee) houden we de volgende indeling aan:

MICRO - MINI - MIDI - MAXI - MACRO - SUPER

De termen "maxi" en "macro" zijn nog niet erg algemeen ingeburgerd, en worden ook wel in omgekeerde volgorde gebruikt. We geven hieronder een redelijk hanteerbare karakteristiek van deze 6 categorieën.

- micro-computers

Main frame op 1 of enkele chips. Alleen main RAM, mogelijk met ROM. Woordlengte 1, hoogstens 2 bytes. Alleen INT-aritmetiek. Transput-controller geheel gedelegeerd aan CPU. Peripherals: input alleen keyboard, output scope en/of eenvoudige printer, store FDU en/of CTU. Hoogstens één gebruiker (geen multiprogrammering). Geen grote rekensnelheid, of snelle memory-access.

- mini-systemen

Main frame niet noodzakelijk op chip(s); MSI gangbaar, maar maakt plaats voor LSI. Extended RAM niet gebruikelijk, maar niet uitgesloten. Woordlengte 2 bytes, dubbellengte woorden van 4 bytes mogelijk (meestal software). Soms floating aritmetiek (vaak software). Transput-controller vrijwel altijd funktie van CPU. Peripherals: input eenvoudige CR en/of PTR, output eenvoudige LP en/of PTP, store FDU en/of DU, eenvoudige MTU en/of CTU. Terminal-aansluiting(en) eerder regel dan uitzondering: een stuk of 16 is niet abnormaal. Multiprogrammering eerder regel dan uitzondering; mini-systemen zijn vaak juist gespecialiseerd in "timesharing". Redelijke rekensnelheden (processorcycle rondom de µsec. memorycycle enkele µsec.)

- midi-systemen

Main frame uitgevoerd in snelle MSI (maakt langzaam, gedeeltelijk, plaats voor LSI, maar snelheid begint belangrijk te worden). Extended RAM vaak aanwezig. Woordlengte 4 bytes normaal, evenals hardware-floating-aritmetiek. Ook dubbellengte aritmetiek (zij het vaak in software). Transput-controller meestal kleine dedicated processor, vaak met eigen geheugen. Alle peripherals in principe mogelijk en vaak in duplo of meer; redelijk omvangrijke achtergrondgeheugens, meestal enkele snelle DU en MTU. Communications controller en multiplexer voor (local en soms ook remote) terminals meestal aanwezig. Vrij grote rekensnelheden (processorcycle 100-250 ν sec, memorycycle 0.5-1 µsec.).

- maxi-systemen

Als midi-systemen, maar met zeer omvangrijke achtergrondgeheugens: 10 of meer DU's en MTU's normaal. Input/output: alle apparaten in veelvoud beschikbaar. Remote-batch stations en uitgebreide terminal-aansluitingen (local en remote). De transput-controller vaak gesplitst in verscheidene dedicated processors met eigen geheugen (de CYBER heeft 10 zg "peripheral processors" met elk 4K 16-bits woorden geheugenruimte, 1 "PP" werkt uitsluitend voor het operating system). Vaak meer dan 1 CPU of/en CPU met gemultipliceerde "function-boxes" in de ALU voor parallelle ALU-operaties.

De Utrechtse CYBER kan men beschouwen als een middelgroot maxi-systeem.

- macro-systemen

Een maxi-systeem met een of meer midi's (voor "dedicated tasks" zoals terminal-afhandeling, separate compilation, text-editing, graphical displays etc), mogelijk (bovendien) meer dan 1 maxisysteem, onder één operating system (dat is essentieel, zie verder) samengevoegd tot een zeer grote "monolithische" configuratie. De LSI-technologie heeft hier uiteraard een grote toekomst: allerlei standaard-taken kunnen nu buiten de CPU(s) om door micro's, mini's en midi's worden verricht - wat "vroeger" ("vandaag" nog) firmware- en basis-software taken zijn, worden "nu" (nabije toekomst dus) aparte processoren of complete kleine systemen. De enige (maar niet geringe) problematiek is het ontwerp van een alles-regisserend en coördinerend operating system. De LSI- en VLSI-technologie zijn hier dusdanig in beweging, dat er weinig zinnigs over valt te voorspellen: ontwikkelingen die snel een eind maken aan het hele concept van een macrosysteem zijn even denkbaar als het samensmeden van verscheidene macro's tot macro-macro-etc.

- super-computers

Bij super-computers denkt men in eerste instantie aan rekensnelheid en reken-precisie (woordlengte van 8 bytes min of meer standaard). Het begrip "megaflop" wordt hier gehanteerd als performance-maat: 1 megaflop = 10^6 floating-point operaties per seconde - dat veronderstelt een processor-cycle van hoogstens 100 νsec. Thans reeds operationele (en zelfs commercieel beschikbare) super-computers hebben een performance in de grootte-orde van 10-100 megaflops - processorcycles tot rond 1 νsec. zijn thans reeds gerealiseerd (experimenteel). Men moet zich goed realiseren, dat daarmee dan ook de grens van het fysische-mogelijke in zicht komt: in 1 νsec legt het licht een afstand af van 3 dm. en electronenstromen in transistors gaan een factor 1.5 tot 3 keer zo langzaam. De lengte van de verbindingskanalen (bussen) beginnen dus kritisch te worden, bovendien gaan allerlei quantum-mechanische effecten een rol spelen (er ontstaan essentiële onzekerheden in het wèl of niet schakelen van flipflops).

Super-computers kunnen (en zullen) het hart gaan vormen van macrosystemen, zodat de grote configuraties van de toekomst "super-macro's" zullen zijn. Op dit punt is het echter uiterst riskant zich aan al te apodictische uitspraken te wagen. Er is teveel in beweging, maar de ontwikkeling is wèl fascinerend.

Computer-netwerken

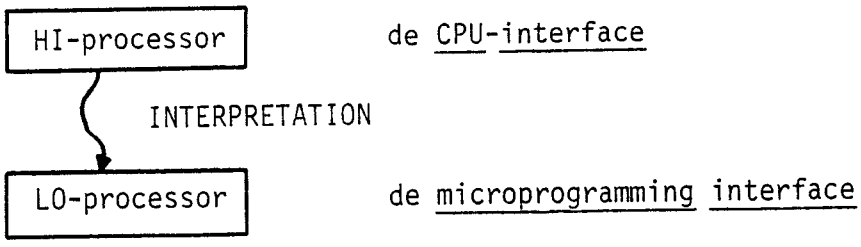
Hèt kenmerk van de hierboven geschetste systemen is, dat ze allen werken onder één operating system (mogelijk een coördinatie van enkele operating systems wat zich echter toch weer presenteert als een hogere operating-system interface). Een totaal ander concept is de koppeling van autonome systemen (die in principe alles kunnen zijn van mini- tot super-macro) met behoud van hun autonomie. Iedere computer in een computer-netwerk kan worden aan- en af-gekoppeld zonder dat dit hun operationele mogelijkheden essentieel wijzigt - anders gezegd: de koppeling is niet sterker dan die van individuen die met elkaar telefoneren (dit beeld kan men trouwens vrij letterlijk opvatten).

Een opmerkelijk aspect van computer-netwerken is de alles-overheersende rol van de interface-architectuur. Men onderscheidt (internationaal geaccepteerde ISO-standaard) thans tenminste 7 "layers" (successieve interfaces in één hiërarchie) van de fysische hardware van een verbindings-kanaal tot en met de taal waarin de computers met elkaar communiceren: deze interfaces worden ook wel protocollen genoemd.

4. D E F A N C Y P R O C E S S O R

4.0 DE PROCESSOR (CPU)

In dit hoofdstuk beschrijven we (vrijwel volledig gedetailleerd) de FANCY-CPU-interface (het hoofdbestanddeel van de hardware- of conventional machine-interface). We doen dit door (hier en daar) een kijkje achter de schermen te nemen. Men kan zich daarbij het volgende beeld vormen (zie 3.2):



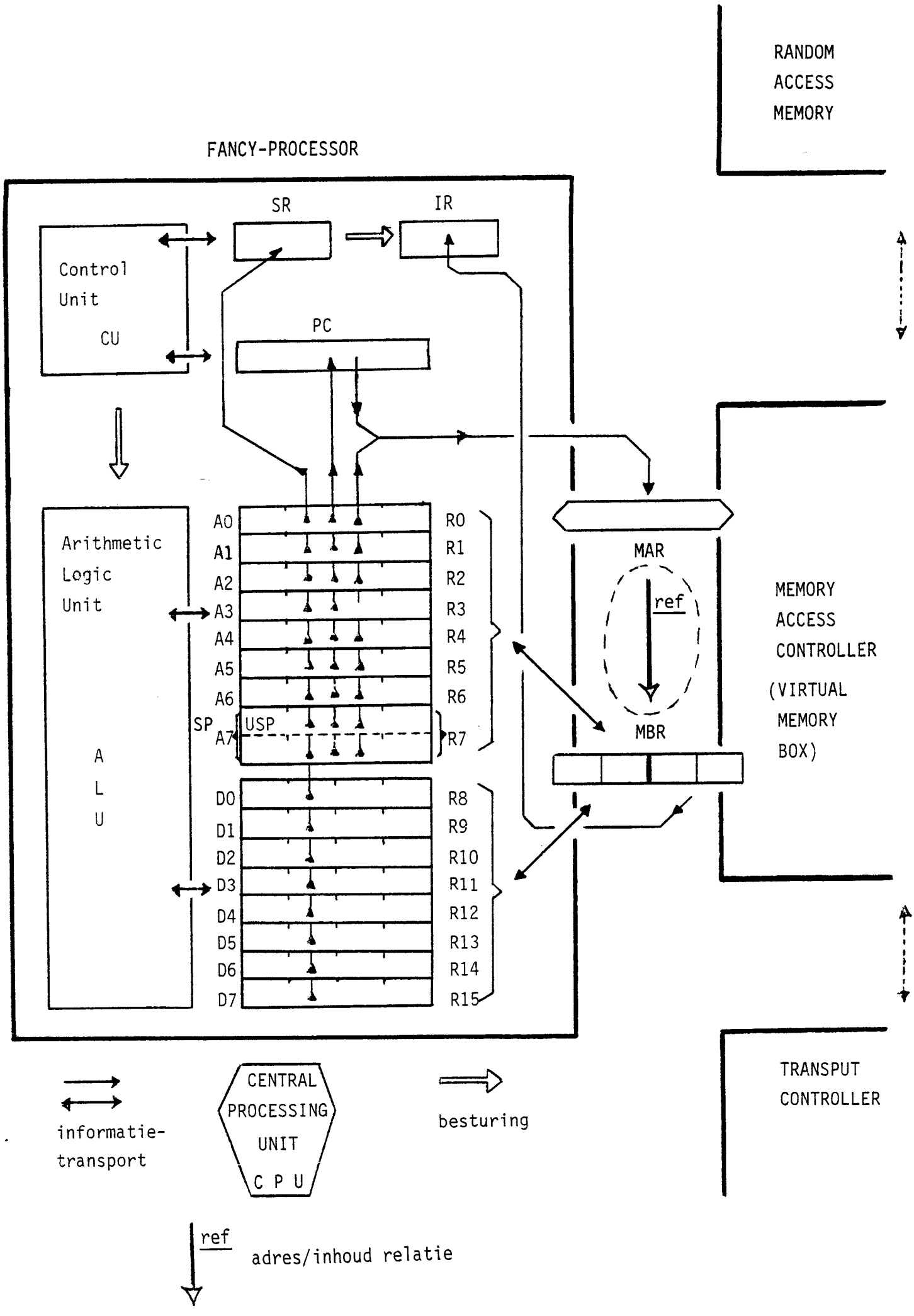
We zien de FANCY-CPU dus (althans gedeeltelijk) als een HI-processor waarvan de basiscyclus door een (de individuele bits manipulerende) LO-processor wordt geëxecuteerd (interpretatie-structuur). In het schema op de volgende pagina kunnen we A0 t/m A7, D0 t/m D7 en enkele individuele bits van SR beschouwen als de zichtbare bestanddelen van de HI-processor, alle andere registers maken essentieel deel uit van de LO-processor en zijn dus in feite onzichtbaar langs de CPU-interface.

De taak van de (HI-)processor bestaat uit het uitvoeren van programma's opgeslagen in het geheugen van de FANCY-machine.
Dit gebeurt door programmainstructies uit het geheugen te halen, te controleren en uit te voeren.

In termen van de LO-processor is de (micro-)programmering in grote lijn als volgt:

De Control Unit laadt de instructies (aangewezen door de PC), detecteert het instructietype en stuurt de ALU.
De Arithmetic and Logic Unit (ALU) voert de aritmetische en logische instructies uit.
De LO-processor bevat een klein, zeer snel flipflop-geheugen voor opslag van tussenresultaten en controle-informatie: de registers, met partieel eigen functies. Het belangrijkste register is de Program Counter (PC), dat steeds wijst naar de volgende uit te voeren instructie. Het Instruction Register (IR) bevat de instructie die op een gegeven moment wordt uitgevoerd.

Als we in de LO-processor kijken, zien we nog diverse (hier niet benoemde) "hidden registers" voor allerlei tussentijdse (adres- en reken-)informatie.


RANDOM
ACCESS
MEMORY
FANCY-PROCESSOR
SR
IR
Control
Unit
CU
PC
Arithmetic
Logic
Unit
A
L
U
A0
A1
A2
A3
A4
A5
A6
SP
USP
A7
R0
R1
R2
R3
R4
R5
R6
R7
D0
D1
D2
D3
D4
D5
D6
D7
R8
R9
R10
R11
R12
R13
R14
R15
MAR
ref
MBR
MEMORY
ACCESS
CONTROLLER
(VIRTUAL
MEMORY
BOX)
TRANSPUT
CONTROLLER
informatie-
transport
CENTRAL
PROCESSING
UNIT
C P U
besturing
ref
adres/inhoud relatie

De processor executeert een instructie in een serie kleine stappen (de basiscyclus), die (in pseudo-Algol68) als volgt kunnen worden aangegeven:

```
while  LO processor busy
   do  MAR := PC;
       PC +:= 2;
       FETCH2;
       IR := MBR2;
         if  IR requires operands
       then  FETCH operands required
         fi;
       execute IR
   od;

proc FETCH2 = void:  MBR2 := M[MAR];
    C  where MBR2 identifies the two rightmost bytes of MBR  C

proc IR requires operands = bool: <body> ;
    C  yields true if the opcode requires one or more memory-
       operands (cf. 4.4)  C

proc FETCH operands required = void: <body> ;
    C  fetches all operands required in (possibly) hidden registers C

proc execute IR = void: <body> ;
    C  executes the instruction in IR, as specified by the opcode  C
```

4.1 HET GEHEUGEN

Het geheugen van de FANCY is dat deel van de machine waarin instructies en data (programma en statusvector, zie 3.0) zijn opgeborgen. Het FANCY-geheugen is byte-adresseerbaar, dwz: ieder machinewoord (van 32 bits) is onderverdeeld in 4 bytes.

Omdat de program counter PC uit 32 bits bestaat, geldt:

$0 \leq \alpha \leq 2^{32} - 1$ α is het (virtual) memory byte-address

Dit geeft een (maximale) geheugen-omvang van 2^{32} = 4096 Mbytes = 1024 Mwords. Dit zou een gigantisch primair geheugen zijn, dat slechts zelden (of nooit) in z'n geheel aangesloten zal (kunnen) zijn. De PC-adresseringsruimte $[0:2^{32}-1]$ wordt de virtual address-space genoemd; deze moet worden afgebeeld op de physical address-space [0:mupb] waarin mupb, de memory upperbound, van machine tot machine kan verschillen. Voor verschillende mogelijkheden voor deze afbeelding, zie verderop (pag 93) en DEEL II.

Het FANCY-geheugen is alsvolgt ingedeeld:

	0	1	2	3
α-8	--------	--------	--------	--------
α-4	--------	--------	--------	--------
α	--------	--------	--------	--------
α+4	--------	--------	--------	--------
α+8	--------	--------	--------	--------

Uit het geheugen kunnen woorden, halfwoorden of bytes worden geselecteerd, waarbij de volgende adressen legaal zijn:

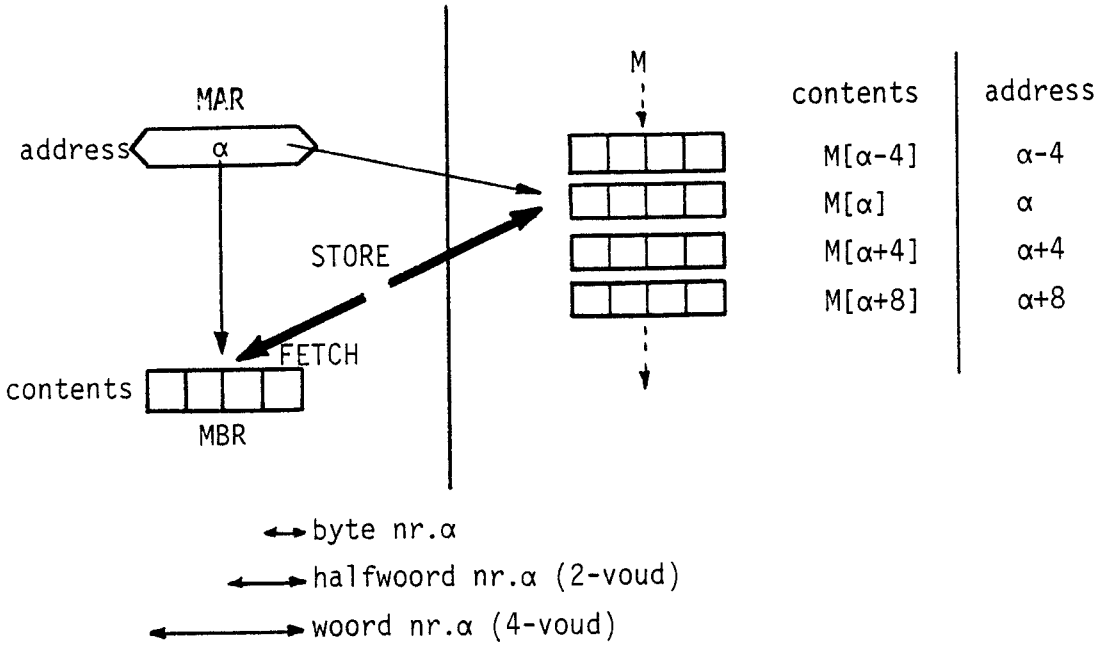
woord-selectie : α mod 4 = 0 (woordadres)
halfwoord-selectie : α mod 2 = 0 (halfwoordadres)
byte-selectie : α mod 1 = 0 dwz. iedere α

De processor communiceert via een tweetal registers met het geheugen: het Memory Address Register (MAR) en het Memory Buffer Register (MBR). Heeft de processor informatie uit het geheugen nodig (data of instructie), dan wordt het memory-address geladen in MAR en vervolgens een leessignaal (FETCH) naar het geheugen gestuurd. Het geheugen start hierop een leesaktie en na 1 memory-cycle wordt de gevraagde informatie afgeleverd in MBR (typische cycle-time k µsec met k = 0.1 ... 2.5).
Om informatie in het geheugen te schrijven, wordt door de processor het

adres in MAR geplaatst, de te schrijven informatie in MBR en vervolgens een schrijfsignaal (STORE) naar het geheugen gestuurd.
Wegens de 3 verschillende selectiemogelijkheden kent de FANCY ook 3 soorten FETCH- en STORE-instrukties:

FETCH1	STORE1	voor een byte,
FETCH2	STORE2	" " 2byte = halfwoord en
FETCH4	STORE4	" " 4byte = woord.

Schema van de processor-memory communicatie:



Bij een FETCH1/STORE1-instruktie staat de byteinformatie in het meest rechtse byte van MBR (en resterende bytes zijn 0); bij FETCH2/STORE2 in het rechterhalfwoord van MBR (en linkerhalfwoord bevat 0) en bij FETCH4/STORE4 in alle 4 bytes van MBR.

Een geheugenwoord heeft twee attributen:

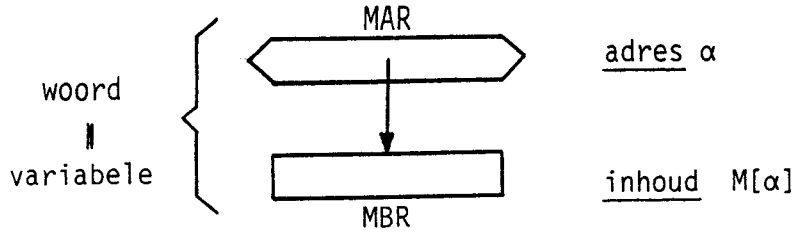
een adres α met $0 \leq \alpha \leq 2^{32} - 1$

(hierin is α een bitconfiguratie die geïnterpreteerd wordt als een nummer: $\alpha \in N$)

en een inhoud $M[\alpha]$ of (zie 4.4) $@\alpha$

(een bitrij van 32, 16 of 8 bits).

Let hierbij op het symbolisme:



De FETCH- en STORE-instructies behoren essentieel tot het repertoire van de LO-processor, dwz.: ze maken deel uit van de micro-programmering van de HI-processor (de eigenlijke CPU-interface). Hun werking en de vergaande consequenties van de MAR-MBR relatie wordt in 4.4 in detail bekeken.

De MEMORY ACCESS CONTROLLER regelt de afbeelding van de virtual address space $0 \leq \alpha \leq 2^{32}-1$ op de physical address space $0 \leq a \leq$ mupb. Voor de afbeelding van α naar a bestaan verschillende regimes, waarvan er enkele in DEEL II worden behandeld. We volstaan hier met twee uitersten:

1) Steeds is $a = \alpha$. Er moet dan worden ingegrepen zodra $\alpha >$ mupb: "illegal address interrupt". Dit is de botte bijl, tevens de methode die geen extra hardware kost en op vele eenvoudige systemen wordt toegepast.

2) Een VIRTUAL MEMORY BOX berekent voor iedere FETCH en STORE opnieuw het physical address van α: $a = \varphi(\alpha)$. Voor de afbeelding φ bestaan nog weer verschillende methoden.

Of de MEMORY ACCESS CONTROLLER nu wel of niet met een VIRTUAL MEMORY BOX is uitgerust, is eigenlijk niet interessant voor het werken langs de FANCY CPU-interface omdat het nu juist de functie is van een VIRTUAL MEMORY BOX (of eventueel een ander regime) om de afbeelding van α naar a voor de CPU-programmeur (en langs iedere hogere interface) te verbergen. We kunnen ons dus vrijwel altijd de luxe permitteren net te doen alsof steeds $a = \alpha$, en dat zullen we ook steeds doen.

In feite is het een van de funkties van FETCH en STORE om ervoor te zorgen dat langs de CPU-interface steeds $a = \alpha$ geldt, waarmee voor alle praktische toepassingen is bereikt dat de virtual- en de physical address space als identiek kunnen worden opgevat.

4.2 REGISTERS EN INSTRUCTIE-REPERTOIRE

4.2.0 De FANCY-registers

De FANCY LO-processor heeft (voor zover voor ons van belang) de volgende registers ter beschikking:

SR	= Status Register	[2 bytes]	
IR	= Instruction Register	[16 bits]	
PC	= Program Counter	[24 bits]	kan [32 bits] worden
MAR	= Memory Address Register	[32 bits]	(zie 4.1)
MBR	= Memory Buffer Register	[4 bytes]	(zie 4.1)
R_i	= ALU-registers	[32 bits]	$0 \leq i \leq 15$

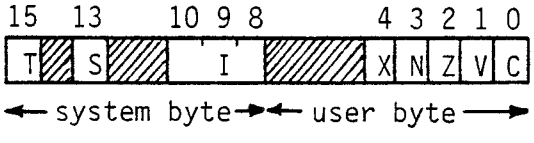
De ALU-registers worden weer onderscheiden in:

A_h	= Address register	$0 \leq h \leq 7$	$A_h \equiv R_h$
en D_h	= Data register	$0 \leq h \leq 7$	$D_h \equiv R_{h+8}$

Het Address Register A_7 wordt in de FANCY altijd aangeduid als de Stack Pointer (SP) en de FANCY-machine heeft voor zowel de User als voor het operating system een eigen SP (aangeduid met USP resp. SP, indien nodig).

Van al deze registers zijn alleen de ALU-registers (A_h en D_h) in de HI-processor zichtbaar (en indirect ook de userbyte van SR, zie verderop). Voor een goed begrip van de HI-processor is echter wel zoveel inzicht in de LO-processor nodig, dat men de functie van voornoemde registers min of meer moet kennen. Een korte bespreking volgt hieronder.

Het Status Register is als volgt ingedeeld:



De gearceerde bits in het SR worden (momenteel nog) niet gebruikt.
De betekenis van de verschillende bits is de volgende:

T = Trace mode	1 = on, 0 = off
S = System state	1 = on (processor werkt in operating system mode), 0 = off (user mode)
I = Interrupt mask	$0 \leq I < 8$ (zie hfdst. 7)
X = Extend N = Negative Z = Zero V = Overflow C = Carry	1 = yes (set), 0 = no (clear)

Het userbyte met hierin de 5 genoemde bits wordt ook wel aangeduid als het Condition Code Register (CCR).

4.2.1 Het instructie-repertoire

Het CPU-instructie repertoire laat zich grofweg onderverdelen in:

1. arithmetic-logic (ALU-) instructions (4.3)
2. control (CU-) instructions (4.5)
3. privileged instructions (zijdelings hfdst. 7)

Alle instructies hebben 0 (dat is bij slechts enkele het geval), 1 of 2 (dat is bij de meeste het geval) operanden, met vaak ook een zij-effect op de CCR-bits. Bij 1 of 2 operanden is hoogstens (en meestal precies) 1 operand de zg. doel-operand ("destination"), dat is de operand waarvan de waarde verandert (altijd een variabele dus) - de andere operand (indien aanwezig) is de zg. bron-operand ("source").

Het kenmerk van de ALU-instructies is dat de operand(en) altijd ALU-registers en/of geheugencellen zijn, beide kunnen zowel doel- als bron-operand zijn.

Het kenmerk van de CU-instructies is dat de program-counter PC altijd de doel-operand is (soms bovendien een geheugencel).

Het kenmerk van een privileged instructie is, dat hij alleen in "system-mode" uitgevoerd kan worden (dwz. als de S-bit in het Status Register on is en daardoor alleen ter beschikking staat van firmware- en basis-software programmeurs).

4.3 DE ALU-INSTRUCTIONS

4.3.0 Operaties en notaties

De Arithmetic Logic Unit executeert alle FANCY-operaties op bitrijen in de ALU-registers, en stuurt bovendien een grote variëteit aan FETCH- en STORE-instructies (zie 4.4). Veel van deze FETCHes en STOREs zijn naar resp. van anonieme registers in de ALU LO-processor, zodat het er soms de schijn van heeft dat de ALU direct opereert op bitrijen in het geheugen. Wat betreft de HI-processor ("onze" CPU) is dit dan ook zo, maar bekeken langs de microprogramming interface (de LO-processor) worden alle bitrijen (met diverse FETCHes) eerst naar anonieme registers in de LO-processor gehaald, en na de operatie eventueel (met diverse STOREs) weer naar het geheugen geschreven.

Er zijn echter ook vele ALU-operaties waarvoor géén geheugencontacten nodig zijn, deze operaties spelen zich dus geheel af in (al dan niet anonieme) LO-processor registers: we noemen ze interne operaties.
Interne operaties zijn altijd veel sneller dan operaties met (een of meer) geheugencontacten - elk geheugencontact kost nu eenmaal tijd ("memory cycle" k μsec). Een aantal ALU-operaties bestaan alleen in interne vorm. Een goed optimaliserende compiler (/assembler) streeft altijd naar een minimalisering van de geheugencontacten - probeert dus zoveel mogelijk rekenwerk te realiseren in interne instructies (zie 3.1.3 en 3.1.4).

In het nuvolgende overzicht laten we de geheugencontacten, met de nogal barokke (on)mogelijkheden van het FANCY-adresseringsmechanisme, nog even buiten beschouwing (zie hiervoor 4.4). We duiden de operanden aan met Δ (destination) en Σ (source), eventueel Δ_1, Δ_2, Σ_1, Σ_2, ... als er meerdere destinations resp. sources zijn: een Δ-operand is na afloop van de operatie veranderd, een Σ-operand nooit. Bij die operaties die alleen in interne vorm bestaan, gebruiken we wél de expliciete registernaam $(D_h, A_h, \ldots)$.
We gebruiken R_i als zowel "A" als "D" voor "R" gebruikt kan worden. We schrijven ook de registernamen in posities waar nooit iets anders kan staan.

Sommige operaties hebben alleen betrekking op individuele bits (resp. deelrijen) van de betrokken operand(en). We zullen zo'n bitrij soms zien als een bit-array: de index van de array noteren we dan tussen accolades { en } - de operand tussen de accolades (de index dus) wordt dan uiteraard altijd als een natuurlijk getal geïnterpreteerd.

Voor de operatoren gebruiken we een ALGOLachtige notatie, die voor zover nodig, kort wordt toegelicht. Overal waar dit zonder verwarring te zaaien kan, vermelden we (in kader) bovendien de FANLAN-mnemonics ("operator key") van de beschouwde operatie.

Het zij-effect op de condition-codes (X, N, Z, V en C) in SR (zie 4.2) geven we als volgt aan:

- = not effected by the operation
* = set according to the operation
0 = cleared

Bij enkele opdrachten is het zij-effect afhankelijk van de nadere definitie van Δ en Σ. In die gevallen vermelden we alleen de zij-effecten die in elk geval bestaan. In deze situatie (waarin het zij-effect dus onvoldoende is gedefinieerd) voorzien we de CCR-bits van een ∇.

Voor alle verdere details: zie FANLAN GUIDE.

4.3.1 Transfer instructions

Δ := Σ copy (assingment, "move")
Δ wordt een copy van Σ, Σ verandert niet.

X	N	Z	V	C	∇	
-	-	-	-	-	∇	COPY, COPYA

SR{0:7} := D_h{0:7} define conditioncodes
de user-byte wordt ingevuld met in D_h gegeven bitrij

X	N	Z	V	C	
*	*	*	*	*	COPY

D_h{0:15} := SR fetch SR

X	N	Z	V	C	
-	-	-	-	-	COPY

R_i :=: R_j verwissel ("swap") de bitrijen in R_i en R_j

X	N	Z	V	C	
-	-	-	-	-	SWAP

4.3.2 Logical operations

Δ ∧:= Σ Δ en Σ worden geïnterpreteerd als rijen individuele bits ("logical values").
Δ wordt vervangen door het logisch produkt ("∧") van Δ en Σ:
0∧0 = 0∧1 = 1∧0 = 0, 1∧1 = 1

X	N	Z	V	C
-	*	*	0	0

AND

SR{0:7} ∧:= Σ{0:7} De bits in SR worden vervangen door het logisch produkt van SR en Σ.

X	N	Z	V	C
*	*	*	*	*

AND

Δ ∨:= Σ Δ wordt vervangen door de (inclusieve) logische som ("∨") van Δ en Σ, waarbij Δ en Σ als logical values geïnterpreteerd worden.
0∨0 = 0, 0∨1 = 1∨0 = 1∨1 = 1

X	N	Z	V	C
-	*	*	0	0

OR

SR{0:7} ∨:= Σ{0:7} De bits in SR worden vervangen door de (inclusieve) logische som van SR en Σ.

X	N	Z	V	C
*	*	*	*	*

OR

Δ |:= Σ Δ wordt vervangen door de exclusieve logische som ("|") van Δ en Σ, waarbij Δ en Σ als logical values geïnterpreteerd worden.
0|0 = 0, 0|1 = 1|0 = 1, 1|1 = 0

X	N	Z	V	C
-	*	*	0	0

XOR

SR{0:7} |:= Σ{0:7} De bits in SR worden vervangen door de exclusieve logische som van SR en Σ.

X	N	Z	V	C
*	*	*	*	*

XOR

Δ := not Δ		Ieder bit van Δ wordt vervangen door het logisch complement ("not").
X N Z V C		
- * * 0 0	NOT	

4.3.3 Bit manipulations

Z := $\Sigma_1\{\Sigma_2\}$		Bitnummer Σ_2 in Σ_1 wordt getest en het resultaat hiervan wordt geplaatst in het Z(Zero)-bit van CCR.
X N Z V C		
- - * - -	BITTST	

Z := Δ{Σ}; Δ{Σ} := 0		Bitnummer Σ in Δ wordt getest en het resultaat hiervan komt in Z-bit van CCR. Vervolgens wordt het geteste bit 0 gemaakt (clear).
X N Z V C		
- - * - -	BITCLR	

Z := Δ{Σ}; Δ{Σ} := 1		Bitnummer Σ in Δ wordt getest en het resultaat hiervan komt in Z-bit van CCR. Vervolgens wordt het geteste bit 1 gemaakt (set).
X N Z V C		
- - * - -	BITSET	

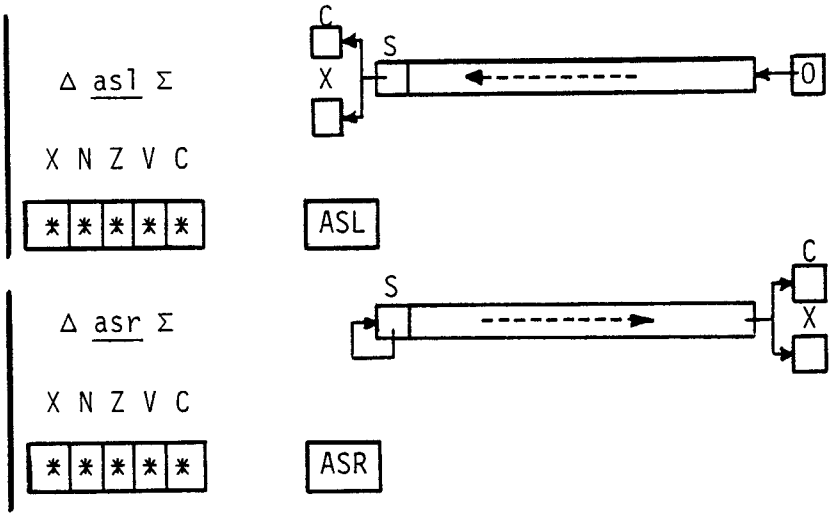
Z := Δ{Σ}; Δ{Σ} := not Δ{Σ}		Bitnummer Σ in Δ wordt getest en het resultaat hiervan komt in Z-bit van CCR. Vervolgens wordt het geteste bit geïnverteerd ("not").
X N Z V C		
- - * - -	BITINV	

In bovenstaande operaties wordt Δ dus geïnterpreteerd als rij individuele bits (logical values) en Σ (en Σ_2), zijnde het bitnummer, als natuurlijk getal.

4.3.4 Shifts & rotations

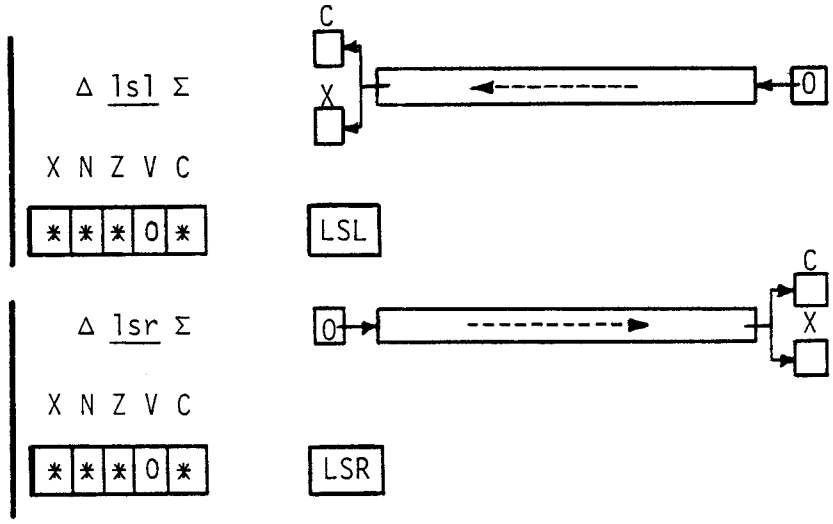
Arithmetic shifts

Δ wordt geïnterpreteerd als bitvoorstelling van een integer waarde (met tekenbit S) en Σ (= het aantal te schuiven bits) als natuurlijk getal.

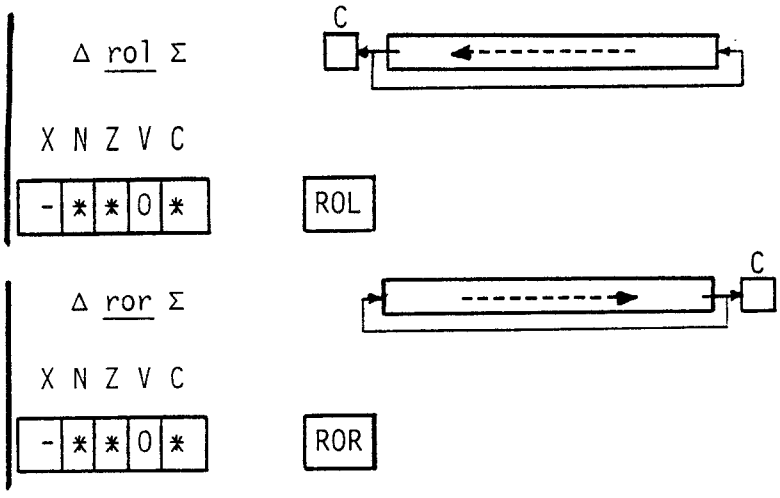


Bij de resterende shift- en rotatie-operaties wordt Δ geïnterpreteerd als rij individuele bits (logical values) en Σ (= het aantal te schuiven bits) als natuurlijk getal.

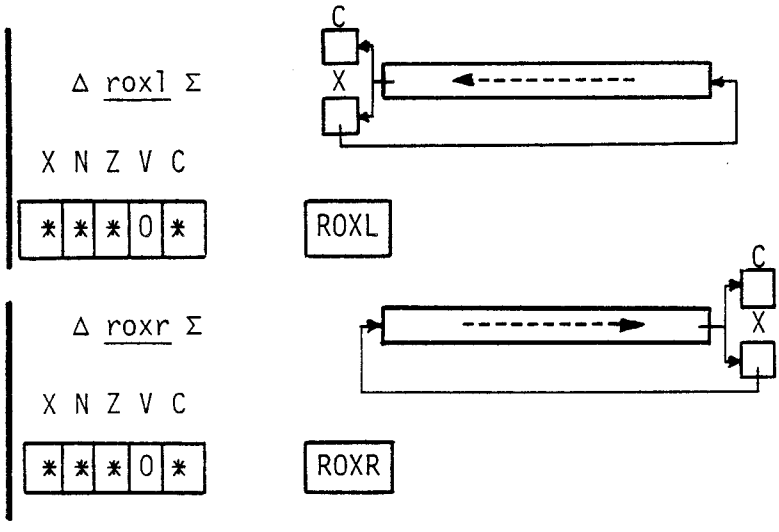
Logical shifts



Rotation without extend



Rotation with extend



4.3.5 Integer operations

Δ +:= Σ — Δ en Σ worden geïnterpreteerd als gehele getallen (integers).

X	N	Z	V	C
*	*	*	*	*

o — ADD, ADDA

Δ -:= Σ — Δ en Σ worden geïnterpreteerd als integers.

X	N	Z	V	C
*	*	*	*	*

o — SUB, SUBA

$D_h := D_h\{15:0\} * \Sigma\{15:0\}$

X	N	Z	V	C
-	*	*	0	0

MUL, MULU

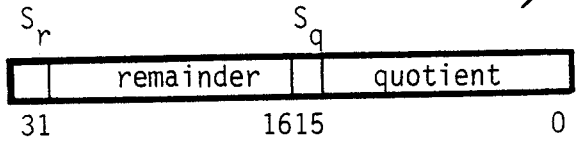
$D_h\{15:0\} := D_h$ over $\Sigma\{15:0\}$

$D_h\{31:16\} := D_h$ mod $\Sigma\{15:0\}$

X	N	Z	V	C
-	*	*	0	0

DIV, DIVU

Bij de binaire vermenigvuldiging en deling worden het destination register D_h en Σ geïnterpreteerd als integers óf als unsigned integers (natuurlijke getallen dus)



$\Delta := -\Delta$ — Δ wordt de negatie van Δ, dwz. $\Delta := 0 - \Delta$.

X	N	Z	V	C
*	*	*	*	*

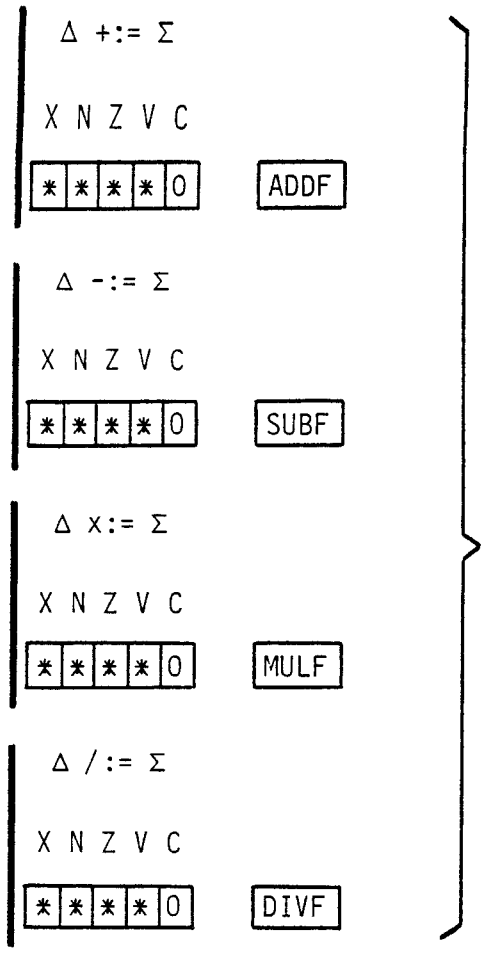
NEG

$\Sigma_1 \Leftrightarrow \Sigma_2$ — De condition code bits worden gezet volgens het resultaat van $\Sigma_1 - \Sigma_2$.

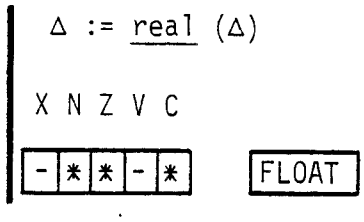
X	N	Z	V	C
-	*	*	*	*

COMP, COMPA

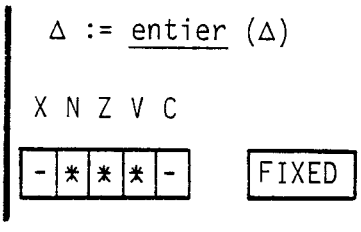
4.3.6 Floating point operations



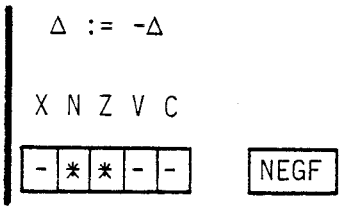
Δ en Σ worden geïnterpreteerd als floating point getallen.



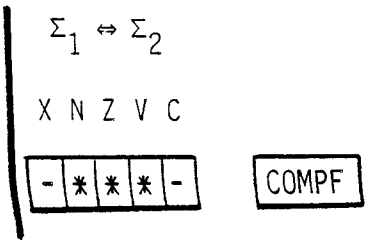
Δ (geïnterpreteerd als integer) wordt geconverteerd naar een floating point getal.



Δ (geïnterpreteerd als floating point getal) wordt geconverteerd naar een integer getal.



Δ (geïnterpreteerd als floating point getal) wordt additief geinverteerd: Δ := 0.0 - Δ

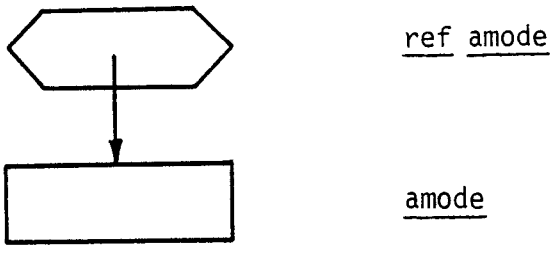


De conditioncode bits worden gezet volgens het resultaat van $\Sigma_1 - \Sigma_2$.

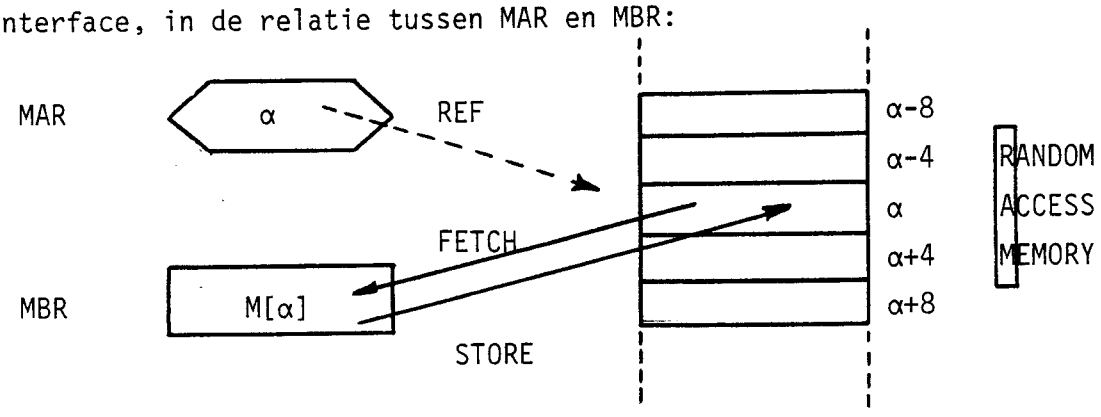
4.4 HET FANCY-ADRESSERINGSMECHANISME

4.4.0 Adres en inhoud

De in hogere programmeertalen bekende relatie tussen "adres" (ref) en inhoud (value):



vinden we vrijwel net zo terug op het veel lagere nivo van de LO-processor interface, in de relatie tussen MAR en MBR:



Hierin is α het (virtual of effective) memory byte-address, op grond waarvan de MEMORY ACCESS CONTROLLER het physical address *a* berekent. De numerieke waarde van *a* speelt voor het programmeren van de HI-processor geen enkele rol - virtuele geheugen-strategieën worden pas zichtbaar langs of zelfs onder de LO-processor (microprogramming-)interface.

Ook de numerieke waarde van α speelt langs de HI-processor nauwelijks een rol, en voor FANLAN (één interface hoger dus) in het geheel niet meer. Wèl van belang zijn vaak de numerieke verschillen van twee (virtual- of effective) adressen: $\alpha_1 - \alpha_2$ dus. Men mag er altijd van uitgaan dat voor alle praktische toepassingen de MEMORY ACCESS CONTROLLER ervoor zal zorgen dat:

$$\boxed{\alpha_1 - \alpha_2 = \sigma_1 - \sigma_2}$$

Langs de HI-processor interface kunnen we dus ook in dit opzicht zonder zorgen α en σ vereenzelvigen.

Veel belangrijker dan de $\sigma = \varphi(\alpha)$ correspondentie is dan ook de wijze waarop α langs de CPU-interface, en ihb. in FANLAN, wordt "bespeeld". Op deze uiterst belangrijke zaak gaan we nu verder in.

In hogere (machine-onafhankelijke) programmeertalen worden <identifier>s altijd gebruikt als naam voor (kleinere of grotere) bitrijen (byte, 2byte, 4byte, ...) in het geheugen. Men geeft dus de bitrij M[α] een naam, en niet zijn adres α. Het is uiteraard ook de inhoud die van belang is en niet zijn verpakking.

In het machine-afhankelijke ("assembler"-) programmeren is men (te) lang blijven vasthouden aan het (numerieke) adres-concept, en heeft men (overigens weinig consequent) ook namen gebruikt voor adressen. Wij zullen in FANLAN, en ook in de beschrijving van de CPU in dit hoofdstuk, zeer consequent namen alléén gebruiken voor bitrijen, die (!) eventueel geïnterpreteerd kunnen worden als adressen van andere bitrijen (zie 4.4.1).

We gebruiken (vooral ook in FANLAN) het volgende symbolisme:

var = M[α]	de naam (<identifier>) "var" staat voor de inhoud van een of ander adres α.

Het adres van "var" duiden we aan door de descriptieve adres-operator # (speek uit "address of", of kortweg "sharp"):

$$\boxed{\#var = \#M[\alpha] = \alpha}$$

Omgekeerd, uitgaand van een adresnummer α, kunnen we het uit hogere programmeertalen bekende symbolisme "M[--]" opvatten als een descriptieve inhouds-operator. In de klassieke assembleertalen schrijft men vaak kortweg "(--)", dus "(α)" ipv. "M[α]". Wij gebruiken hiervoor de descriptieve operator @ (speek uit "contents of", of beter "addressed by", of kortweg "at").

We krijgen aldus:

$$\boxed{@\alpha = M[\alpha] = var}$$

De notatie met "M[--]" zullen we voortaan alleen nog gebruiken als we heel erg expliciet over de volledige memory-array M[$0:2^m-1$], met m = 16, 24, 32, ..., willen spreken.

In de context van CPU-programmering en FANLAN gebruiken we steeds α.

Van belang is de identiteit:

#@α ≡ α	# is altijd een links-inverse van @

We zullen verderop zien dat @ ook gezien kan worden als een links-inverse van #. Als we consequent nooit numerieke adressen gebruiken (en in FANLAN is dat altijd te vermijden) geldt zonder restrictie: | #@ = @# |, maw.: "sharp" en "at" zijn elkaars inverse.

OPM: Laat goed doordringen dat # en @ géén machineoperaties zijn, of zelfs maar op een of andere manier deel uitmaken van een machine-operatie. Het zijn notationele operatoren, we zouden ze "notatoren" kunnen noemen.

Alle bitrijen in het geheugen kunnen door STOREs worden veranderd en zijn dus essentieel variabelen. De bitrijen in de registers zijn uiteraard ook variabelen, we zullen hiervoor steeds de FANLAN-namen gebruiken:

- in de adresregisters: de adres-variabelen A0, A1, A2, A3, A4, A5, A6, SP (de bitrij in A7 wordt altijd aangeduid met de naam SP: Stack Pointer)
- in de dataregisters : de data-variabelen D0, D1, D2, D3, D4, D5, D6, D7

In de LO-processor heeft $\#A_i$ resp. $\#D_i$ nog wel betekenis: de registers vormen ahw. een klein intern geheugen van de CPU; langs de HI-processor interface kan echter niet meer over $\#R_i$ worden gesproken (het register-adresseringsmechanisme is daar altijd achter de schermen verdwenen).

4.4.1 Adresserings nivo's

Omdat bitrijen (in het geheugen en in de registers) poly-interpretabel zijn, moeten van @α altijd nog twee attributen worden gespecificeerd:

size @α = het aantal bytes in @α
type @α = de interpretatie van @α

Men moet zich realiseren dat @α in principe nog een willekeurig lange rij bytes kan zijn. Als bijv. op adres α een tekst begint, is size @α het aantal characters in de tekst. Het FANCY-instructie repertoire onderscheidt echter slechts sizes van 1, 2 of 4 bytes.

Het type van @α wordt per instructie bepaald door het type operatie. In principe is de interpretatie die men zelf aan een bitrij wil geven (even-als zijn size) echter vrij en bepalen de specifieke FANCY-operaties alleen op welke wijze men zo'n interpretatie kan realiseren.

Een zeer belangrijke interpretatie van een 2byte (eventueel 4byte) is de interpretatie als adres. Over het algemeen worden A0.....A6 en SP als adres-variabelen gebruikt, dwz. de eigenlijke data zijn @A0, ... @A6 en @SP.
De microprogrammering die hier achter zit is natuurlijk:

```
MAR  :=  Ai      |  MBR  :=  @Ai
FETCH            |
```

Zonder problemen kan men ook een geheugen-variabele interpreteren als een adres. Bijv. (in FANLAN):

```
COPY  A0  var    |  D0  :=  @var
COPY  D0  @A0    |
```

Dit kan trouwens ook in 1 FANCY-instruction (zonder gebruik van A0):

```
COPY  D0  @var
```

Merk op dat "var" op deze wijze als adres-variabele wordt geïnterpreteerd. Men kan nog dieper adresseren, bijv.:

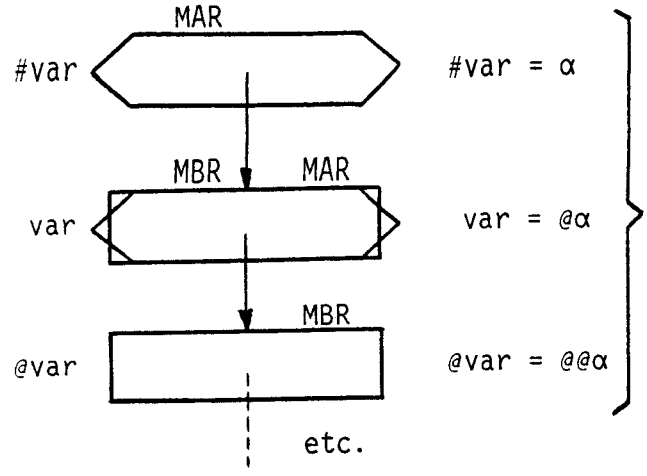
```
COPY  A0  @var   |  D0  :=  @@var
COPY  D0  @A0    |
```

Dit gaat echter niet in 1 FANCY-opdracht. In principe kan men zo "diep" adresseren als men wil, hierop berust uiteraard het pointer-mechanisme in hogere programmeertalen (REF REF -- in ALGOL68).

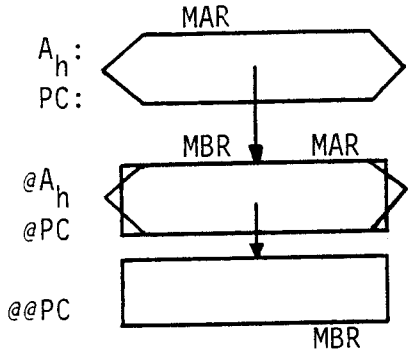
Deze (zg. "indirecte") adressering berust dus op de interpretatie:

type @α = β (het adres van een andere operand)

en op de realisering van:



In het FANCY-adresseringsmechanisme wordt de centrale rol gespeeld door de relatie:



Welke van deze adresseringen (en ook nog welke extra's) in welke instructions mogelijk zijn is niet zo eenvoudig te zeggen en wordt geheel bepaald door de instruction-codering.

4.4.2 Naamgeving

Hoewel het onderwerp van dit hoofdstuk de CPU-interface is, moeten we in deze paragraaf even vooruit lopen op de FANLAN-interface (zie hfdst. 6) en we bekijken nu eerst de FANLAN-naamgeving die ten nauwste samenhangt met het FANCY-adresseringsmechanisme.

Een FANLAN-programma is opgebouwd uit <line>s, "regels" dus die onder elkaar worden geschreven (zie FANLAN GUIDE). Elke regel specificeert iha. een <bytes>: een bitrij van 1 of meer bytes. Zo'n <bytes> specificeert een opdracht of data, of een uit deze componenten samengestelde gecompliceerde rij van bytes. Deze <bytes> komen ergens in het geheugen te staan: de pre-

ciese plaats (het numerieke adres) zal de FANLAN-programmeur nooit hoeven te kennen, maar hij kan wel een naam geven aan de <bytes> en eventueel over het (relatieve) adres beschikken door toepassing van de "address of" ("sharp") operator.

Voorbeeld:

```
                <bytes>
                   ¦
label1:         <bytes>      $ deze bitrij heet "label1"
                   ¦
var:            <bytes>      $ deze bitrij heet "var"
                   ¦
label2:         <bytes>      $ deze bitrij heet "label2"
                   ¦
                <bytes>
```

Over size en type van de bitrijen valt niets à priori te zeggen; deze worden geheel bepaald door de operatie(s) waaraan deze bitrijen worden onderworpen. Voorbeelden:

```
COPY  1  D3   var     $ size var = 1, type var = ?
COPY  2  var  D3      $ size var = 2, type var = ?
COPY     D3   var     $ size var = 4, type var = ?
ADD   2  D3   var     $ size var = 2, type var = int
ADDF     D3   var     $ size var = 4, type var = real
etc.
```

Uiteraard kan de context van een FANLAN-<line> veel suggereren. Voorbeelden:

```
pi:      FLOAT  3.1415927        $ size pi = 4 (?)
                                 $ type pi = real (?)
text:    STRING "i love fancy"   $ size text = 12 (?)
                                 $ type text = string (?)
move:    COPY  4  D3  var        $ size move = 2 (!?)
                                 $ type move = <instruction> (?)
```

Of een bitrij als "opdracht" of als "data" wordt geïnterpreteerd, hangt er maar van af of hij ooit in IR terecht komt of niet. Voorbeeld:

```
here:    HEX  220A
```

wordt geïnterpreteerd als de opdracht "COPY 4 D1 A2" zodra PC = #here wordt (wat bijv. zou kunnen door een opdracht als "GOTO here"). Maar in

een opdracht als "ADD 2 D3 here" wordt dezelfde bitrij als int geïnterpreteerd.

Hoewel het numerieke adres van een bitrij voor de FANCY-programmeur nooit van belang kan zijn, moet hij wel degelijk de mogelijkheid hebben met numerieke "afstanden" te werken. Voorbeeld:

```
label1:     <bytes>      $ #label1 is in principe onbekend
                :
                :
label2:     <bytes>      $ #label2 is in principe onbekend
```

Het aantal bytes tussen #label1 en #label2 is echter:

```
#label2 - #label1
```

Het programma-deel:

```
PUSH  #label2 - #label1
CALL  INTOUT
```

drukt dit aantal af.

Natuurlijk kan men ook schrijven:

```
PUSH  #label1
CALL  INTOUT
```

en dan wordt het "numerieke adres" wel degelijk afgedrukt.
Daar heeft men echter niets aan: het is systeem-afhankelijk of dit nu ε of α is (zie 4.4.1) of nog iets anders en het kan zelfs tijdafhankelijk zijn (zie hfdst. 5).

4.4.3 Het processor-geheugen

De registers vormen het processor-geheugen: een klein supersnel geheugen waarvan de (LO-processor) adressering niet via MAR gaat. Dit processor-geheugen, voor zover relevant voor de FANLAN-programmeur, kan men zich als volgt gedeclareerd denken:

```
A0:   SKIP  4      $ 4 bytes, adresregister
A1:   SKIP  4      $ 4 bytes, adresregister
 :     :
 :     :
 :     :
A6:   SKIP  4      $ 4 bytes, adresregister
SP:   SKIP  4      $ 4 bytes, adresregister A7, stackpointer
```

```
D0:   SKIP  4      $ 4 bytes, dataregister
D1:   SKIP  4      $ 4 bytes, dataregister
 ¦     ¦
 ¦     ¦
 ¦     ¦
 ¦     ¦
 ¦     ¦
D7:   SKIP  4      $ 4 bytes, dataregister
```

Ook de program counter PC kan een expliciete rol spelen in opdrachten (zie 4.4.15 en 4.4.16), terwijl zelfs de SR-bytes expliciet voorkomen in enkele opdrachten:

```
PC:   SKIP  4      $ 3 of 4 bytes, denk maar 4
SR:   SKIP  2      $ 2 bytes
```

De sharp-operator kan op geen van deze namen worden losgelaten, maar voor het overige kunnen ze als "gewone" variabele bitrijen worden opgevat, zij het dat ze op bijzondere posities in opdrachten kunnen optreden.

We zullen zien dat de size van A_h en D_h altijd precies 4, 2 en soms 1 is. De bitrijen zelf (4byte, 2byte of byte) zijn vrij vergaand poly-interpretabel, zij het dat de A_h ahw. voorbestemd is om adressen te bevatten (adres-registers) en D_h om data te bevatten (data-registers). PC is altijd een adres (van de volgende 2byte in de "program-space" (zie ook DEEL II)), SR nooit iets anders dan een bitrij (conditions, etc).

4.4.4 Instruction-formats

In deze paragraaf bekijken we (zeer globaal) de codering van de FANCY-instructions voor de LO-processor en tevens de notatie van deze instructions in FANLAN. De FANLAN-notatie is optimaal gestroomlijnd (zij het toch nog heel wat "hariger" dan een machine-onafhankelijke taal), de codering voor de LO-processor is zeer "tricky". Het is de taak van een assembler om FANLAN-notatie om te zetten in LO-processor code.

We zullen in het volgende vaak de COPY-instruction als voorbeeld nemen, omdat vrijwel alle adresseringswijzen daarin zijn toegestaan en alle drie standaard sizes 1, 2 en 4, en omdat de LO-code ervan goed demonstreert met wat voor middelen men opdracht-informatie in enkele bytes kan coderen.

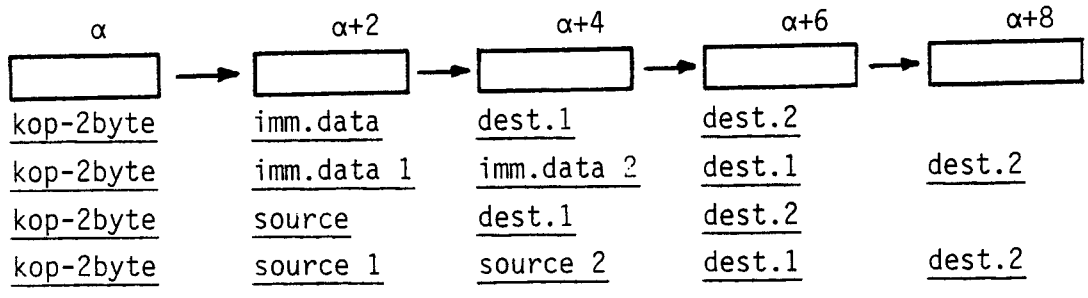
Een FANLAN-opdracht heeft de globale layout:

<key><size><destination><source><condition>

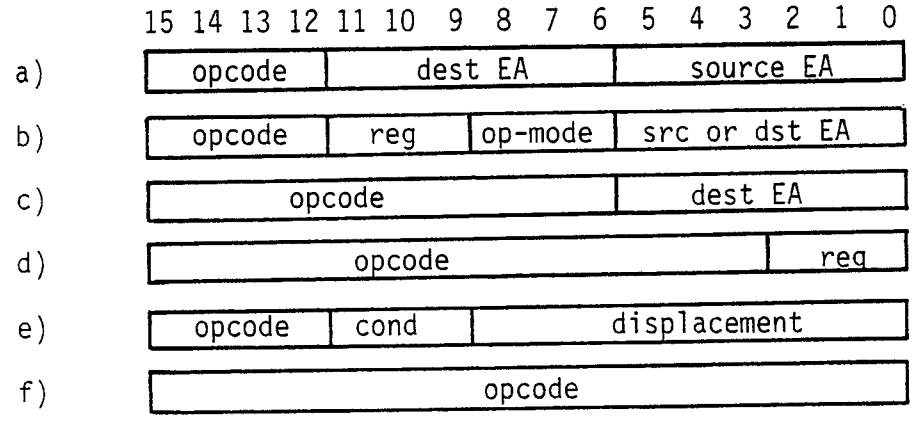
De <key> specificeert de operatie, de <size> het aantal bytes (1, 2 of 4)

waarop de operatie betrekking heeft (het niet vermelden van de <size> betekent meestal 4), de rest spreekt vanzelf. Er is slechts één opdracht waarin al deze 5 componenten voorkomen; de ALU-instructions hebben geen <condition>, de control-instructions hebben (met één uitzondering) geen <destination>.

Voor de verpakking (format) van de opdrachten in (2, 4 of meer) bytes wordt een tamelijk geraffineerd systeem gevolgd dat we niet tot in de details hoeven te kennen. Elke opdracht beslaat minstens 1 2byte, de zg. kop-2byte. De codering van de kop-2byte bepaalt of, en hoeveel extension-2bytes nog nodig zijn voor verdere specificatie van de opdracht. De interne opdrachten hebben geen extension-2bytes - dat maakt ze dus extra snel (geen geheugen-contacten voor operanden en slechts één FETCH voor de opdracht zelf). De eenvoudige opdrachten met geheugen-operand(en) hebben 1 of 2 extension-2bytes. Bij gecompliceerde adresseringen kan het aantal extension-2bytes oplopen tot 4. Vooruitlopend op de volgende paragrafen bekijken we het volgende beeld:



De kop-2bytes zijn ingedeeld in een aantal format-groepen. De belangrijkste zijn:

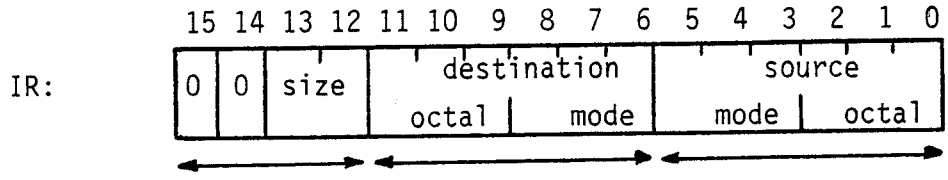


Hierin specificeert opcode altijd de combinatie <key><size>. EA bepaalt (direct of indirect via extension-2bytes) het zg. Effective Address (ε) van de operanden (<destination> en/of <source>).

De hiervoor gegeven formats zijn die van:

a) de (meeste) COPY-instructions
b) de (meeste) andere opdrachten met 2 operanden
c) opdrachten met alleen een destination
d) interne opdrachten op 1 register
e) de (meeste) control-instructions
f) opdrachten met inherente adressering

In de nuvolgende paragrafen geven we een volledige behandeling van de FANCY-adressering. We doen dit aan de hand van de COPY-instructions omdat deze alle adresseringen toelaten en daardoor exemplarisch zijn voor de LO-processor instruction formats:



De bits 15-12 geven de instruction-group aan; de COPY-instructions van het diadische type worden dan gespecificeerd met:

size { 1 → 0 0 0 1 ; 2 → 0 0 1 1 ; 4 → 0 0 1 0 }

De beide volgende 6bits in de instruction-codering (de effective address codes) specificeren de destination- en source-operanden. De operand is de bitrij (in een register of het geheugen) die bij de operatie betrokken is. Destination als de bitrij "verandert" en source indien hij niet wijzigt. Een effective address codering is opgebouwd uit 2 velden, genaamd mode en octal (beide 3 bits lang).

In de volgende subparagrafen worden alle mogelijke operand-coderingen en de hiermee te gebruiken adresseringsmogelijkheden nader bekeken.

4.4.5 Data register direct

De operand is de inhoud van dataregister D_h.

Codering: | 0 0 0 | h | (mode | octal) <operand> = D_h

Voorbeelden:

```
COPY     D3  D7
COPY  2  D7  D5
```

4.4.6 Address register direct

De operand is de inhoud van adresregister A_h.

Codering: | 0 0 1 | h | (mode | octal) <operand> = A_h

Voorbeelden:

```
COPY     A2  A6
COPY  2  SP  A3
```

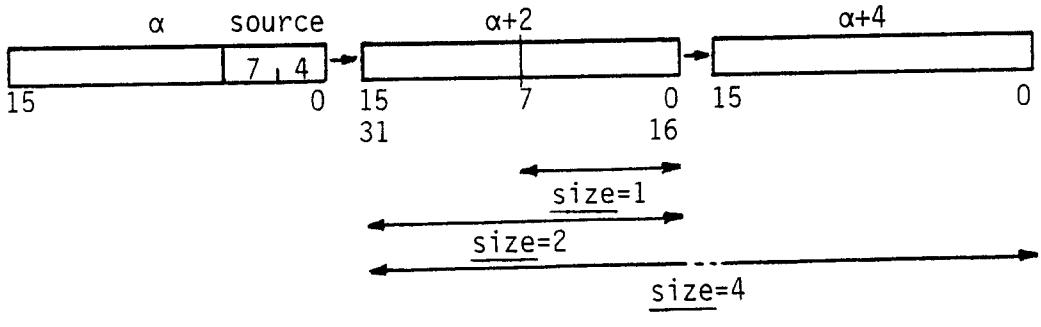
Met behulp van data register direct en address register direct zijn alle register copies $R_i := R_j$ met 3 verschillende sizes beschikbaar:

COPY	1	D_h D_k	ofwel	$D_h := D_k$
	2	D_h A_k	"	$D_h := A_k$
	(4)	A_h D_k	"	$A_h := D_k$
		A_h A_k	"	$A_h := A_k$

4.4.7 Immediate data

De operand is direct na de instruction opgeslagen (of beter gezegd: opgeslagen in 1 of 2 extension-2bytes van de instruction).

Codering: | 1 1 1 | 1 0 0 | (mode | octal) <operand> = @PC



Bij size = 1 (byte) is de data opgeslagen in het rechterbyte van de extension-2byte; bij size = 2 in de gehele extension-2byte en bij size = 4 in de twee extension-2bytes.

Als voorbeeld van deze adressering de COPY-instruction van het type

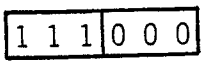
COPY	1 2 (4)	<destination> <constant>

zoals:

```
COPY     D3  -513716005
COPY     D0   3.1415927
COPY  2  D1   15701
COPY  1  D1  -37
COPY  2  A3  #var
COPY  2  A4  #var - 2
```

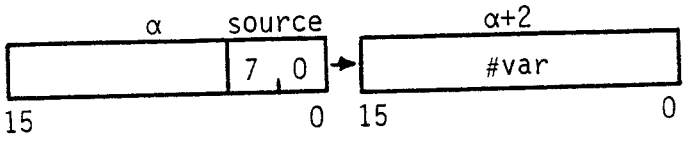
4.4.8 Direct address

Het adres van de operand is opgenomen in de extension-2byte.

Codering: 

mode octal

<operand> = @@PC



Als voorbeeld van deze adressering de COPY-instruction van het type

COPY	1 2 (4)	<destination> var var <destination>

zoals:

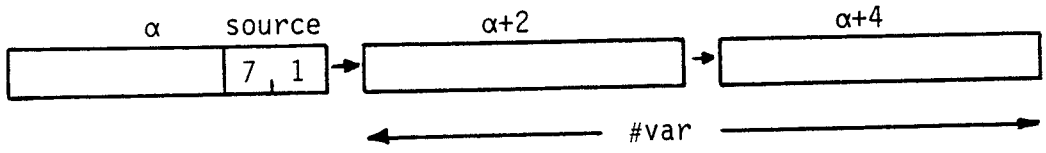
```
COPY    D0  var
COPY    A3  var
COPY 2  D5  twobyte
COPY 1  byte  D7
```

4.4.9 Direct long address

Het adres van de operand is opgenomen in 2 extension-2bytes.

Codering: | 1 1 1 | 0 0 1 | <operand> = @L@PC

mode octal



Indien nodig draagt de FANLAN-assembler zorg voor de keus tussen direct address of direct long address codering.
De voor deze adresseringsvorm te geven FANLAN-voorbeelden zijn identiek aan die bij "direct address".

4.4.10 Address register indirect

Het adres van de operand is aanwezig in een der adresregisters.

Codering: | 0 1 0 | h | <operand> = @A_h

mode octal

Voorbeeld van deze adresseringsvorm is de COPY-instruction van het type

COPY	1	<destination> @A_h
	2	
	(4)	@A_h <source>

zoals:

```
COPY      D0  @A1
COPY      A1  @A2
COPY  2  @A1  var
COPY  1  var  @A4
```

4.4.11 Address register indirect with postincrement

Het adres van de operand is aanwezig in een der adresregisters en na afloop van de operand-FETCH wordt de inhoud van het desbetreffende adresregister verhoogd met de <size> van de operand.

Codering:

0 1 1	h

mode octal

<operand> = $@A'_h$ gevolgd door

A_h +:= size <operand>

De inhoud van het adresregister wordt echter met 2 opgehoogd bij een <operand>size van 1 (byte dus); dit om deze adressering altijd op een 2byte-adres te houden.

Voorbeeld met een COPY-instruction van het type

COPY	1	<destination> $@A'_h$	met	A_h +:= 2
	2	$@A'_h$ <source>		A_h +:= 2
	(4)			A_h +:= 4

ná FETCH

4.4.12 Address register indirect with predecrement

Het adres van de operand is aanwezig in een der adresregisters, maar vóóraf wordt dit adres verlaagd met de <size> van de operand.

Codering:

1 0 0	h

mode octal

A_h -:= size <operand> gevolgd door

<operand> = $@'A_h$

Ook hier wordt net als bij de postincrement-vorm het adresregister met 2 verlaagd bij een <operand>size van 1 (byte).

Als voorbeeld een COPY-instruction van het type

COPY	1	<destination> $@'A_h$	met	A_h -:= 2
	2	$@'A_h$ <source>		A_h -:= 2
	(4)			A_h -:= 4

vóór FETCH

De address register indirect adressering met postincrement resp. predecrement dienen voor het doorlopen van arrays en het implementeren van stacks en queue's (zie hfdst. 5).

4.4.13 Address register indirect with displacement

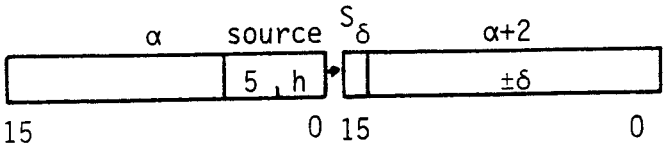
Het adres van de operand wordt gevormd als som van de inhoud van een der adresregisters en een displacement, die is opgenomen in een extension-2byte.

Codering: | 1 0 1 | h |

mode octal

<operand> = @($A_h \pm \delta$)

met $|\delta| < 2^{15}$



Als voorbeeld weer een COPY-instruction, nu van het type

COPY	1 2 (4)	<destination> @($A_h \pm \delta$) @($A_h \pm \delta$) <source>

zoals:

```
COPY     D0  @(A1-4)
COPY     D3  @(A0+ var)
COPY  2  @(A2-8)  D3
```

4.4.14 Address register indirect with index and short displacement

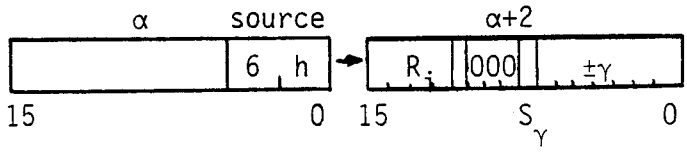
Het adres van de operand wordt gevormd als som van de inhoud van een der adresregisters, de inhoud van een indexregister en een displacement. Dit displacement én specificatie van het desbetreffende indexregister zijn opgenomen in een extension-2byte.

Codering: | 1 1 0 | h |

mode octal

<operand> = @($A_h + R_i \pm \gamma$)

met $|\gamma| < 2^7$



Als voorbeeld weer een COPY-instruction, nu van het type

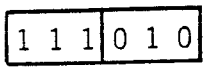
COPY	1 2 (4)	<destination> @($A_h + R_i \pm \gamma$) @($A_h + R_i \pm \gamma$) <source>

zoals:

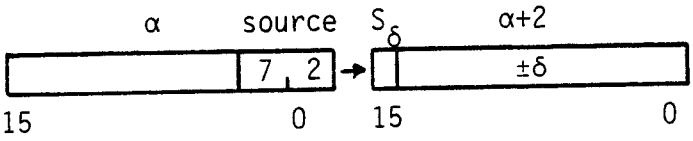
```
COPY     D0   @(A5+A2-4)
COPY  2  D4   @(A0+D0+8)
COPY  1  @(A3+D0)   D2
```

4.4.15 Program counter relative

Het adres van de operand wordt gevormd als som van de inhoud van de program counter (PC) en een displacement, die is opgenomen in een extension-2byte.

Codering: 

<operand> = @(PC±δ)
met $|\delta| < 2^{15}$



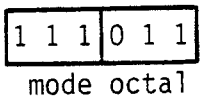
De inhoud van PC, die bij deze adresberekening gebruikt wordt is het adres van de desbetreffende extension-2byte (α+2 in het getekende voorbeeld).

Voorbeelden:

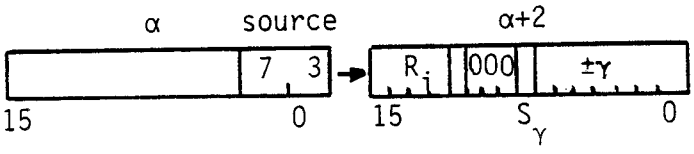
```
COPY  2  D1   @(PC-6)
COPY     @(PC-8)   D4
```

4.4.16 Program counter relative with index

Het adres van de operand wordt gevormd als som van de inhoud van PC, de inhoud van een indexregister en een displacement. Dit displacement én de specificatie van het desbeteffende indexregister zijn opgenomen in de extension-2byte.

Codering: 

<operand> = @(PC+R_i±γ)
met $|\gamma| < 2^7$



De inhoud van PC, die bij deze adresberekening gebruikt wordt is het adres van de desbetreffende extension-2byte (α+2 in het getekende voorbeeld).

Voorbeelden:

```
COPY     D0   @(PC+A2-3)
COPY  2  var  @(PC+D3+127)
COPY     @(PC+D0+37)   D2
```

4.4.17 Overzicht van de effective address coderingen

EA	EA code	<operand>	addressing mode	extens. 2bytes
0.h	000. h	D_h	data reg. direct	0
1.h	001. h	A_h	addr.reg. direct	0
2.h	010. h	$@A_h$	addr.reg. indirect	0
3.h	011. h	$@A_h'$	addr.reg. indir. & postincr.	0
4.h	100. h	$@'A_h$	addr.reg. indir. & predecr.	0
5.h	101. h	$@(A_h \pm \delta)$	addr.reg. indir. & displ.	1
6.h	110. h	$@(A_h+R_i \pm \gamma)$	addr.reg. indir. & index	1
7.0	111.000	@@PC	direct address	1
7.1	111.001	@L@PC	direct long address	2
7.2	111.010	$@(PC \pm \delta)$	program counter relative	1
7.3	111.011	$@(PC+R_i \pm \gamma)$	progr.counter relative & index	1
7.4	111.100	@PC	immediate data	1|2
7.5	111.101	---------	memory operand indirect	1|2
7.6	111.110	---------		
7.7	111.111	---------		

De EA-codes 7.5 t/m 7.7 bespreken we hier niet (de LO-processor codering is hier nogal "barok"). In FANLAN (zie DEEL II) zullen we deze adresseringsvorm verder bespreken. Het is de adressering waarbij een geheugenoperand (var) als adres geïnterpreteerd wordt (@var).

4.5 DE CONTROL(CU)-INSTRUCTIONS

4.5.0 Sprong-opdrachten en condities

Het kenmerk van de FANCY-<control-instruction>s is, dat de program counter PC één van de (en meestal de enige) doeloperanden is. We bekijken nog eens de basis-cyclus:

```
while  LO-processor busy
   do  MAR := PC;
       PC +:= 2;
       FETCH2;
       IR := MBR2;
         if  IR requires memory operands
       then  FETCH operands required
         fi;
       execute IR
    od;
```

Het effect van "execute IR" is bij een CU-instruction dus

```
PC := ε
```

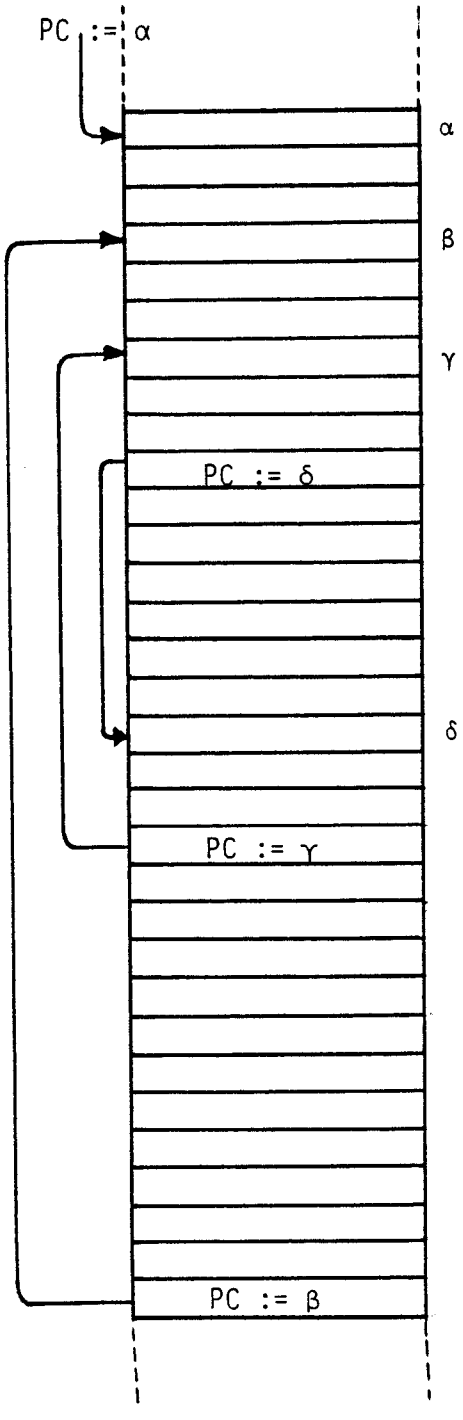
dat wil zeggen: het programma wordt niet voortgezet met "de volgende" opdracht, maar met de opdracht die begint op (het effective address) ε. De besturing van het programma maakt dus een "sprong"; vandaar de naam sprong-opdracht die vaak wordt gebruikt voor de CU-instructions. Wij zullen de naam sprong-opdracht overigens vaker in een beperktere betekenis gebruiken.

De CU-instructions vormen de primitiva voor de "control structures" in de hogere programmeertalen:

de <choise-clause>s	if then else fi	etc.
	case in out esac	etc.
de <loop-clause>s	for from by to while do od	etc.
de <procedure-call>s	print((numb, mean, standev))	
	sqrt(sqean - mean * mean)	etc.

De zg. "goto" (en "call") instructions worden thans als ontoelaatbaar primitief (want gevaarlijk) beschouwd in het machine-onafhankelijk programmeren. De "low level flow of control" is er echter geheel op gebaseerd en is in deze paragraaf aan de orde.

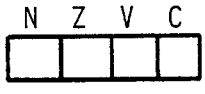
Een opdracht met het effect "PC := ε" kan resulteren in een voorwaartse of terugwaartse sprong; met de terugwaartse sprongen worden loops geconstrueerd. Een bijzondere sprongopdracht is de subroutine-sprong (-aanroep of-call), waarbij het "vertrek-adres" wordt onthouden zodat de besturing daar naar terug kan keren.



De FANCY kent 4 soorten <control-instruction>s:

1. sprong-opdracht: GOTO
2. subroutine-aanroep: CALL
3. return-opdracht: RETURN
4. tel-sprong: GOCNT

Ieder van deze opdrachten, behalve RETURN, kan worden voorzien van een <condition-option>. Hierbij wordt gebruik gemaakt van de vier conditiebits in CCR:



N = NEGATIVE V = OVERFLOW
Z = ZERO C = CARRY

Uit deze vier condition codes kunnen 16 verschillende condities afgeleid worden (zie tabel op volgende pagina): wordt aan de in de opdracht genoemde conditie voldaan, dan wordt de control-instruction uitgevoerd (dwz. PC wordt gewijzigd) en is niet aan de conditie voldaan dan wordt de control-instruction niet uitgevoerd.

code	test	FANLAN
0000	1	"IF TRUE"
0001	0	"IF FALSE"
0010	$\overline{C}.\overline{Z}$	IF NOT CORZ
0011	C+Z	IF CORZ
0100	$\overline{C}$	IF NOT CARRY
0101	C	IF CARRY
0110	$\overline{Z}$	IF NOT ZERO
0111	Z	IF ZERO
1000	$\overline{V}$	IF NOT OF
1001	V	IF OF
1010	$\overline{N}$	IF NOT NEG
1011	N	IF NEG
1100	$N.V+\overline{N}.\overline{V}$	IF NOT LT
1101	$N.\overline{V}+\overline{N}.V$	IF LT
1110	$N.V.\overline{Z}+\overline{N}.\overline{V}.\overline{Z}$	IF GT
1111	$Z+N.\overline{V}+\overline{N}.V$	IF NOT GT

Het loont de moeite, met behulp van Boolese algebra (zie 1.2) na te gaan dat de hierboven gegeven tests inderdaad de achter IF gegeven conditie realiseren.

De syntactische constructie van de control instructions is dan:

```
{ GOTO    <memory loc.>          }
{ CALL    <memory loc.>          } IF  <condition>
{ GOCNT   <D_h> <named loc.>     }
```

Voor een aantal genoemde condities bestaan in FANLAN ook alternatieven:

IF NOT NEG ≡ IF POS
IF NOT LT ≡ IF GE
IF NOT GT ≡ IF LE

Voor syntactische details, zie FANLAN GUIDE.

4.5.1 De GOTO-opdracht

Als we in een programmatekst een <label> schrijven om daarmee een opdracht te markeren waar we op een of andere wijze naartoe willen springen:

```
                <instruction>
                      ¦
                      ¦
here:         <instruction>
                      ¦
                      ¦
                <instruction>
```

en we schrijven ergens:

```
GOTO  here        eventueel        GOTO  here  IF <condition>
```

dan is het effect (indien aan de eventuele conditie is voldaan):

```
PC  :=  #here
```

Hiermee is de semantiek van de (elementaire) sprongopdracht volledig vastgelegd. Voorbeelden:

```
GOTO  again              $ een "onvoorwaardelijke terugsprong" naar
                         $ de label "again"

COMP  D3  D7             $ vergelijk D3 en D7
GOTO  equal  IF  ZERO    $ spring indien  D3 = D7

DIVF  D1  D0             $ D1 := D1/D0
GOTO  alarm  IF  OF      $ spring bij "overflow"
```

Het nogal rijk gescharkeerde adresseringsmechanisme van de FANCY bevat een grote diversiteit aan sprongopdrachten. Hierbij is het noodzakelijk een goed inzicht te hebben in de semantische kwestie:

```
GOTO  here   }
PC := #here  }  zijn equivalent
```

Strikt genomen zou men hier consequenter "GOTO #here" willen schrijven. We hebben hier dan ook te maken met een anomalie in FANLAN. We volgen hier nl. de historische notationele conventies van (vrijwel) alle assembler-talen èn (!), voor zover nog van toepassing, de hogere programmeertalen waar men altijd schrijft "goto label" ipv. "goto #label" of zoiets.

Het loont de moeite te verifiëren dat men geen problemen heeft met:

```
COPY  A1  #here
GOTO  @A1
```
} equivalent met GOTO here

```
COPY  A1  here
GOTO  @A1
```
} equivalent met GOTO @here (indirecte sprong!)

en dat "GOTO A1" in beide gevallen een paradoxale opdracht zou zijn: men kan niet naar een instruction-in-een-register springen! Als men de anomalie een keer begrijpt (en als vanzelfsprekend accepteert), kan men trouwens geen moeite hebben met het phenomeen dat "de hardware" bij wijze van spreken de #-operatie toepast op de operanden bij sprongopdrachten (en dat "kan de hardware" natuurlijk alleen als er minstens één @ staat vóór een registernaam).

Sprongopdrachten kunnen ook PC-relative zijn:

```
GOTO  @(PC - 12)
```

betekent (wat anders?) "ga 12 bytes terug in het programma". Omdat het vaak vervelend is om uit te rekenen hoeveel bytes er in een stukje programma zitten, kan men dit rekenwerk als volgt door de assembler laten doen (zie vooral ook hfdst. 6):

```
there:   <instruction>
                 ¦
                 ¦
                 ¦
here:     GOTO  @(PC - (#here - #there))
```

PC-relative sprongopdrachten schrijft men uiteraard voor dynamische relocateerbaarheid (zie 4.4.15 en DEEL II).

4.5.2 De subroutine-aanroepen (CALL)

Een buitengewoon belangrijk concept in de assembler-programmering is de subroutine: een meer of minder omvangrijk stuk programma dat vanuit verschillende punten van het "hoofdprogramma" kan worden aangeroepen, en vanwaaruit steeds wordt teruggekeerd naar het adres onder de aanroep. Voor de realisatie van een subroutine zijn dan 2 soorten spronginstructions nodig:

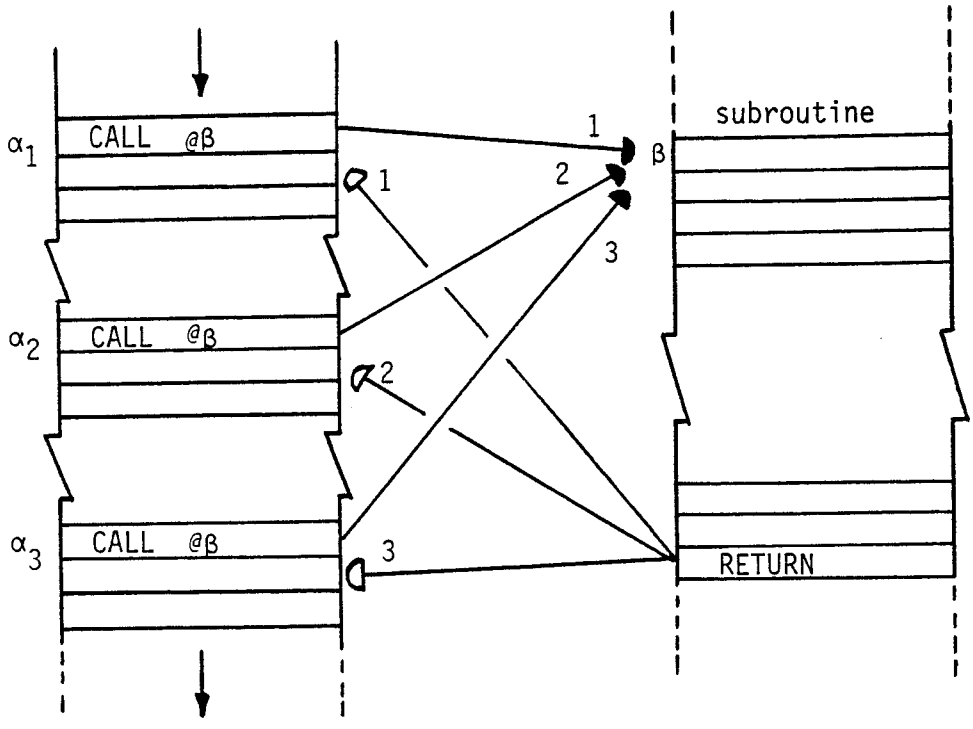
1. aanroep: CALL
   Vóór de sprong moet het terugkeeradres (de link) worden opgeborgen.
   Gebruikt symbool: ——▶

2. terugkeer: RETURN

Na het doorlopen van de subroutine moet "via de link" worden teruggesprongen naar het adres onder de aanroep.

Gebruikt symbool: 

In onderstaand figuur zijn drie aanroepen van dezelfde subroutine beginnend op adres β schematisch weergegeven.



Als we even aannemen dat het terugkeer-adres wordt opgeborgen in een hardware-bepaalde geheugenplaats LINK, en dat de subroutine is gelabeld als "subroutine", dan kan het effect van CALL resp. RETURN als volgt worden gespecificeerd:

CALL subroutine IF <condition>	if condition then LINK := PC; PC := #subroutine fi
RETURN	PC := LINK dus: GOTO @LINK

De geheugenplaats LINK is in de FANCY (zoals op alle moderne processors) een door de stackpointer SP aangewezen geheugenplaats (in de FANCY een 4byte). We kunnen de CALL- en RETURN-opdrachten nu nog iets nauwkeuriger (in pseudo-FANLAN) specificeren:

CALL subroutine IF <condition>	if condition then COPY @'SP PC GOTO subroutine fi
RETURN	GOTO @@'SP dus: COPY PC @SP'

Merk op dat het (om een aantal redenen) niet mogelijk is de CALL en de RETURN te vervangen door andere FANLAN-opdrachten. Het loont de moeite dit aan de hand van de FANLAN GUIDE even nauwkeurig na te gaan!
Voor een verdere behandeling van de subroutine-CALL en -organisatie (ihb. parameter overdracht en recursiviteit) zie DEEL II.

4.5.4 De tel-sprongen (GOCNT)

Het komt nogal eens voor dat men het aantal keren waarin een stuk programma wordt doorlopen, wil tellen en eventueel wil beperken. Dit is oa. het geval in de zg. for-loops. Het low-level primitivum hiervoor is de tel-sprong, een soort GOTO die in een aangewezen data-register (destination dus) 1 aftrekt en alleen wordt uitgevoerd als het resultaat van die aftrekking ≠ -1 is.

Omdat de tel-sprong ook nog een expliciete <condition> mag hebben, is hij dus een dubbel-geconditioneerde sprong-opdracht.

Stel dat we, tellend in D3, willen (terug-)springen naar de label "again", dit kan dan met één opdracht:

GOCNT D3 again IF <condition>	if condition then D3 -:= 1; if D3 /= -1 then PC := #again fi fi

Het loont de moeite om na te gaan, op welke wijze enkele veel voorkomende for-loops in een of andere hogere programmeertaal met behulp van de tel-sprong gerealiseerd (geïmplementeerd) kunnen worden (zie ook DEEL II).