Computerarchitectuur

Gestructureerde referentie

Dit document dient als referentie voor de cursus Computerarchitectuur in BA2 Computerwetenschappen.

Pieter Mees

www.insitehosting.be

20/6/2006

COMPUTERARCHITECTUUR

TABLE OF CONTENTS

COMPUTERARCHITECTUUR

INLEIDING

CONTACTGEGEVENS VAN DE LESGEVER

Professor Koen De Bosschere
Vakgroep ELIS, Technicum, lokaal P1.3
Sint-Pietersnieuwstraat 41
B-9000 Gent
Telefoon: 09 264 34 06
E-mail: kdb@elis.rug.ac.be

ASSISTENTEN

Veerle Desmet
Lieven Eeckhout

AANBEVOLEN LECTUUR

D.A. Patterson en J.L. Hennessy, Computer Organization & Design: the Hardware/Software Interface, Morgan Kaufmann Publishers, 1998

J.L. Hennessy and D.A. Patterson, Computer Architecture: a Quantitative Approach, Morgan Kaufmann Publishers, 2002

EVALUATIE

Studenten worden beooordeeld op een aantal practica (permanente evaluatie) en op het examen (periodegebonden evaluatie). De practica tellen mee voor 10% van de punten en het examen voor 90%. Verder zijn er een aantal regels waarmee men gratie kan verkrijgen indien het resultaat zou tegenvallen.

GESCHIEDENIS VAN DE COMPUTER

AUTOMATISERING VAN DE TOESTAND

De mens gebruikt al zeer lang hulpmiddelen om hem te assisteren bij het rekenen. Deze hulpmiddelen steunen op het concept van een algoritme. De automatisering van algoritmes begon met het extern bijhouden van de toestand in een toestandsgeheugen. Enkele voorbeelden:

- Vingers, keitjes, knoopjes
- Beenderen
- Telramen
- Geschreven symbolen

Het voordeel hiervan was dat de toestand niet volledig in het menselijk geheugen hoefde bijgehouden worden. Hierbij vond een scheiding plaats tussen waar de toestand opsloeg en waar men de transformaties uitvoerde.

AUTOMATISERING VAN TRANSFORMATIES

Een tweede golf van automatisering vind men terug in de zeventiende eeuw toen verschillende instrumenten uitgevonden werden om de transformaties waaruit de algoritmes bestaan uit te voeren. Hierdoor konden de transformaties sneller en preciezer uitgevoerd worden. Een aantal voorbeelden:

De rekenlat
Somcalculator van Blaise Pascal
Productcalculator van Gotfried Leibniz

Deze instrumenten hielden de toestand bij en voerden ook de transformaties uit. De mens was enkel nog nodig voor de controlefunctie.

AUTOMATISERING VAN DE CONTROLE

Als laatste werd ook de controle geautomatiseerd. In het begin gebruikte men ponskaarten om algoritmes te beschrijven en door te geven aan de machines. Een voorbeeld van geautomatiseerde controle is het Jacquardweefgetouw dat door middel van ponskaarten patronen kon weven.

DE EERSTE COMPUTERS

THE DIFFERENCE ENGINE

Dit apparaat is ontworpen door Charles Babbage maar is nooit gebouwd. Op het moment dat hij het ontwerp maakte was de techniek nog niet ver genoeg gevorderd om het apparaat te construeren. De specifieke bedoeling was om veeltermen tot op 6 decimalen te berekenen.

THE ANALYTICAL ENGINE

Een later ontwerp van Babbage dat ook nooit gebouwd werd. Speciaal aan het ontwerp was dat het kon geprogrammeerd worden met behulp van ponskaarten. Ada Lovelace schreef het eerste programma, maar aangezien er geen testapparaat was kon onmogelijk nagegaan worden of het concept werkte.

DE Z-1

Dit was de eerste computer die effectief gebouwd werd. Het was een mechanisch apparaat met enorm grote afmetingen.

DE Z-3

Een opvolger van de Z-1 die deze keer elektromechanisch werkte. Bovendien werkte deze versie met binaire representaties van getallen.

HARVARD MARK I

Oorspronkelijk ontwikkeld door IBM, leek deze machine veel op de de analytical engine. Vooraleer hij aan de Harvard universiteit geschonken werd, gebruikte men hem om schiettabellen te berekenen.

DE ENIAC

ENIAC staat voor Electronical Numerical Integrator And Computer en werd ontworpen om ballistische trajecten voor het Amerikaanse leger te berekenen tijdens Wereldoorlog II. De bouw was echter pas voltooid toen deze al gedaan was en de machine werd toen ingezet bij het ontwikkelen van de waterstofbom.

Het programmeren van de ENIAC was een tijdrovende bezigheid. De schakelaars en kabels moesten handmatig verplaatst worden. Om dit proces te versnellen konden hele schakelborden aangebracht worden. Terwijl de gegevens logisch gescheiden waren, bestand de rekeneenheid en het programma uit één geheel.

DE EDVAC

Een snellere en geavanceerdere versie van de ENIAC. De afkorting EDVAC staat voor Electronic Discrete Variable Automatic Computer. Een belangrijk voordeel ten opzichte van zijn voorganger was dat programma's ook in het geheugen zaten en de computer zelf de nodige verbindingen legde voor de uitvoering. Huidige computers zijn nog steeds op dit principe van John von Neumann gebaseerd. Logisch gezien zijn programma en gegevens nu een eenheid terwijl de rekeneenheid hiervan los staat.

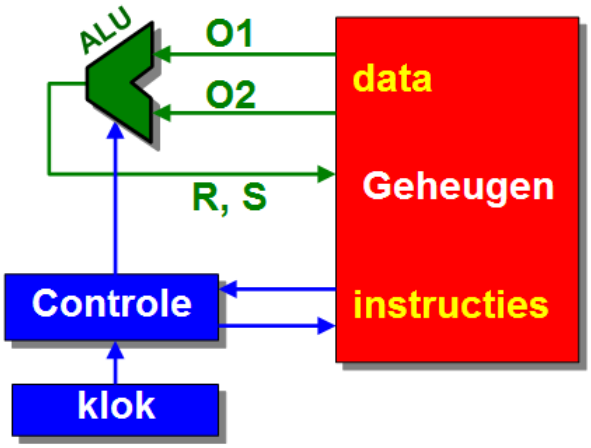
MODERNE COMPUTERSYSTEMEN

HET VON NEUMANN MODEL

Zoals reeds eerder beschreven bij de EDVAC, slaat een von Neumann machine de programmainstructies op in hetzelfde geheugen als de gegevens. Dit heet het "stored program" concept.

LOGISCHE OPBOUW

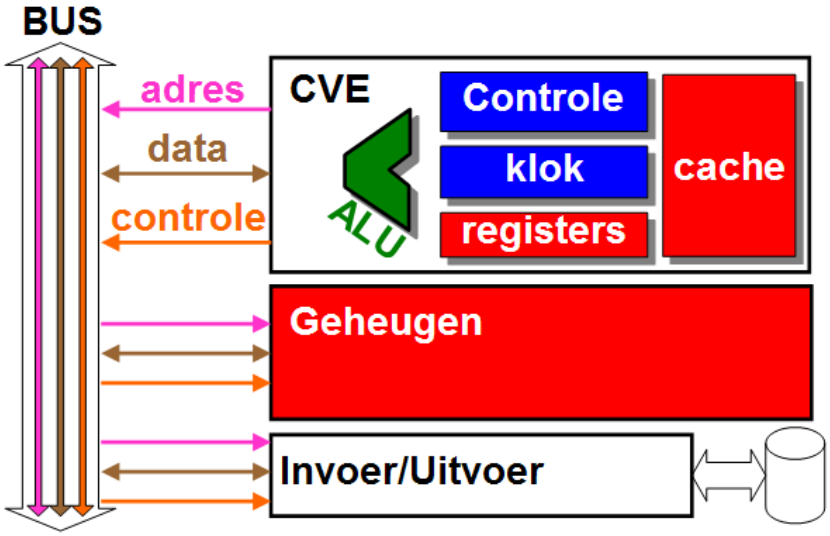
De controle-eenheid haalt instructies uit het geheugen en bepaalt zo de bewerkingen die de ALU moet uitvoeren. De instructie bevat welke transformatie op welke gegevens toegepast moet worden en waar dat resultaat moet terechtkomen. De snelheid waarmee de transformaties in de ALU zich opvolgen, wordt aangegeven door de klok. Elke klokpuls wordt een instructie uit het geheugen gelezen en uitgevoerd in de ALU.



O1 en O2 zijn de inputs voor de transformatie die de ALU zal uitvoeren. R is het resultaat en S is extra informatie over dit resultaat (teken, overdracht, …). De lengte van O1 en O2 is doorgaans gelijk, deze lengte heet de woordlengte en is op huidige computers meestal 32 of 64 bits. Vroegere architecturen gebruikten 4, 8 of 16 bits.

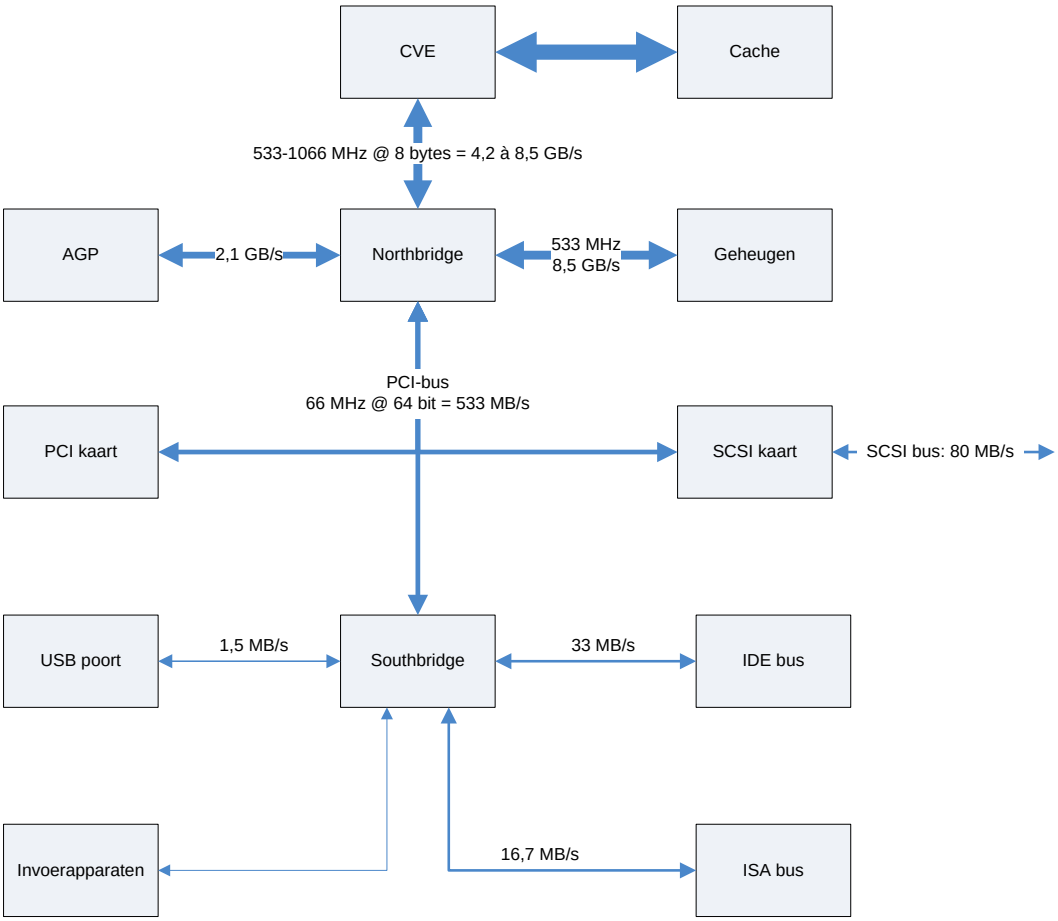
FYSISCHE OPBOUW

Wanneer men een von Neumann machine effectief bouwt, zien we een lichtjes andere opbouw dan de logische structuur.

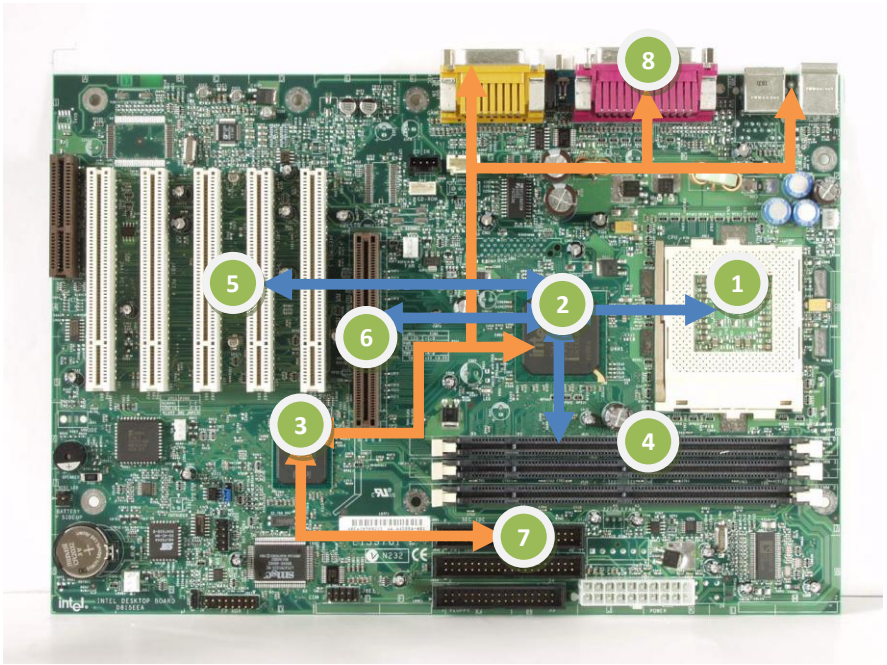


De drie grote componenten: de centrale verwerkingseenheid (CVE), het geheugen en de input/output (I/O) module; zijn allen verbonden via een systeembus. Deze bus heeft een bepaalde bit-breedte, bijvoorbeeld 256 bit en bevat drie aparte kanalen. Een kanaal dient om een adres te selecteren in het geheugen of in de I/O module, een ander kanaal bepaalt de operatie en het derde kanaal levert de data. Verderop bespreken we de systeembus in meer detail.

Het onderstaande schema toont hoe de datacommunicatie tussen de verschillende onderdelen verloopt in huidige computers:



In een realitsitsche uitvoering ziet dit er als volgt uit:



De drie grote componenten uit het diagram vinden we hier terug als:

De CVE wordt geplaatst op het witte vlak (1), dit is de socket
De geheugemodule wordt aangesproken via de northbridge 2

Invoer en uitvoer gaat via de southbridge 3

De bus loopt dus tussen CVE, northbridge en southbridge.

1 De CVE (centrale verwerkingseenheid) of CPU (central processing unit) voert de transformaties en de controlefunctie uit. Bovendien bevat de CVE twee extra sets geheugen: de registers en de cache. Deze geheugeneenheden kunnen zeer snel aangesproken worden bij berekeningen, maar bevatten slechts weinig gegevens.

Van de logische taken is de CVE verantwoordelijk voor de klok, de controlefunctie, de ALU en een deel van het geheugen (registers en cache).

2 De northbridge is verantwoordelijk voor de snelle communicatie tussen de processor, het geheugen, het AGP-slot en de PCI-sloten. Hier is ook enige nuance mogelijk. In processoren van AMD is bijvoorbeeld de geheugencontroller ingebouwd in de processor zelf. Op de tekening ziet u alle communicatie waarvoor de northbridge verantwoordelijk is in het blauw.

3 De southbridge is de tragere tegenhanger van de northbridge. Hij regelt de communicatie tussen de northbridge, de IDE-kanalen (harde schijven, DVD-drives, …), USB poorten en de invoerapparaten. De communicatie van de southbridge verloopt via de PCI-bus van de northbridge.

Het geheel van de northbridge en de southbridge wordt de “chipset” genoemd.

4 Hier bevinden zich de DIMM sockets, de geheugensloten. Deze worden aangesproken via de geheugencontroller in de northbridge of in de processor. De geheugenmodules in het voorbeeld zijn van het type SDRAM. Op dit moment zijn reeds modules tot 4GB beschikbaar, maar in de gemiddelde computer vindt men 512MB terug.

5 Dit zijn de PCI-sloten. PCI staat voor Peripheral Component Interconnect en wordt doorgaans gebruikt voor communicatie met randapparaten zoals geluidskaarten, netwerkkaarten, diskcontrollers, …

6 Het AGP-slot is een speciale interface voor grafische kaarten. AGP staat voor Accelerated Graphics Port. Dit slot wordt echter minder en minder gebruikt naarmate de nieuwe PCI-Express standard opgang maakt. Deze nieuwe interface dient als vervanging voor de PCI-sloten en het AGP-slot en komt voor in verschillende snelheden van 1x tot 16x.

7 Dit zijn de (E)IDE-sloten ((Enhanced) Integrated Device Electronics). Ze worden gebruikt om de verbindinge te maken met opslagmedia als harde schijven, CD-drives, DVD-branders, … Er zijn twee meestal twee sloten beschikbaar die elk 2 apparaten kunnen aansturen. Een apparaat is de master en een de slave.

8 Deze randapparaten worden aangestuurd via de southbridge en bestaan uit:

Een seriële poort die gebruikt wordt door oude muizen en modems
Een parallelle poort voor communicatie met oudere printers
USB-sloten (Universal Serial Bus) verzorgen de communicatie met een groot aanbod aan randapparaten van toetsenborden over joysticks naar printers en flash sticks
2 PS/2-poorten voor communicatie met toetsenbord en muis

ABSTRACTIENIVEAU’S

Om een computer zo flexibel mogelijk te maken en om de complexiteit van het hele systeem te verminderen, heeft men verschillende abstractieniveau’s ingebouwd. Een abstractieniveau wordt gekenmerkt door het feit

dat het een goed gedefinieerde interface naar buiten toe heeft, zowel naar de lagen waarop hij verderbouwt als naar de lagen die erop gebouwd worden.

Dit heeft als voordeel dat men zich niet moet verdiepen in de interne werking van een laag om ermee te kunnen werken. Bovendien kan men eenvoudig de implementatie of de werking veranderen zonder dat daarvoor alle bovenliggende systemen aangepast moeten worden.

Het concept van abstractieniveau's is essentieel bij het garanderen van opwaartse compatibiliteit. Zonder de abstracties zou het invoeren van nieuwe technologieën veel moeizamer verlopen omdat alle software en randapparaten aangepast moeten worden.

Op de volgende pagina ziet u een overzicht van de abstractieniveau's in de huidige computers.


Toepassingsprogramma's
•De laag van programma's zoals de gebruiker ze ziet
•Geschreven door programmeurs of codegeneratoren
•Maken gebruik van hoge-niveauprogrammeertalen
Hoog-niveauprogrammeertalen
•Programmeur ziet abstracties van datatypes, instructies, ...
•De compiler vertaalt naar machinetaal
Machinetaal
•Spiegelt de mogelijkheden van de onderliggende hardware
•Verzameling van instructies heet de "instructieset"
Datapad + Controlepad
•Aangestuurd door de controle-eenheid
•Controlesignalen voor elke instructie
RTL-niveau
•Register Transfer Level
•Implementatie datapad en controlepad is de microarchitectuur
•Bepaalt diepte van de pijplijn en geheugenhiërarchie
•Functionele blokken als registers, ALU's, buffers, ...
•Very High Speed Integrated Circuit (VHSIC)
•Beschreven met VHSIC Hardware Description Language
Poortnetwerken
•Eenvoudige elektronische schakelingen
•Bijvoorbeeld AND, OR, NOT, ...
•Doorgaans vaste omzetting naar transitors en verbindingen
Transistors en verbindingen
•Transistoren en verbindingen
•Automatisch of met de hand vanuit poortnetwerk
•Vooral plaatsing en routing is moeilijk

ARCHITECTUUR & ORGANISATIE

Computerarchitectuur verwijst naar het functionele gedrag van een computersysteem zoals het gezien wordt door de gebruikers en de programmeur. Dit komt in de abstractiehiërarchie overeen met de bovenste drie lagen:

Toepassingsprogramma's
Hoog-niveauprogrammeertalen
Machinetaal

De computerorganisatie heeft te maken met de structurele opbouw van de computer, onzichtbaar voor de gebruiker. In de hiërarchie bestaat dit uit de onderste vier lagen:

Datapad + Controlepad
RTL-niveau
Poortnetwerken
Transistors en verbindingen

COMPATIBILITEIT

HARDWARE-SOFTWARE INTERFACE

De strikte scheiding van architectuur en organisatie biedt interessante mogelijkheden. Ten eerste kan een bepaalde organisatie meerdere architecturen ondersteunen. Ten tweede kan een "familie van computers" met verschillende organisaties toch dezelfde architectuur draaien. Dit houdt dus ook in dat verschillende organisaties hetzelfde programma kunnen draaien als ze de architectuur ondersteunen.

BINAIRE COMPATIBILITEIT

Zie binaire compatibiliteit in het appendix.

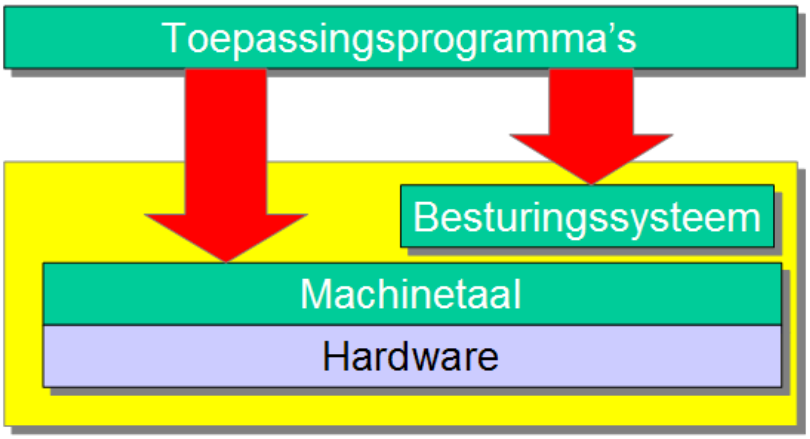
BRONCODECOMPATIBILITEIT

Zie broncodecompatibiliteit in het appendix.

PLATFORM

Zie platform in het appendix.

Tijdens de compilatie van een programma, past het zich niet alleen aan aan de specifieke machinetaal maar ook aan het besturingssysteem om toegang te krijgen tot randapparaten en andere bronnen.

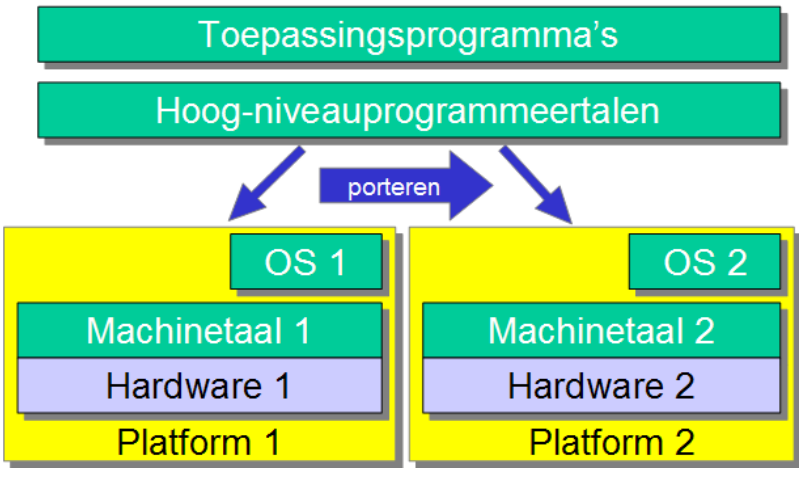


Het gele gebied duidt het platform aan, de rode pijlen de afhankelijkheden bij compilatie.

PORTEREN

Porteren is het omzetten van een programma van een platform naar een ander. Afhankelijk van de verschillen kan dit eenvoudig door te hercompileren of moet het programma helemaal herschreven worden. Meestal zijn echter sommige delen van het programma specifiek voor een bepaald platform, bijvoorbeeld code voor grafische interfaces in Windows.

In sommige gevallen is er zelfs geen compiler beschikbaar voor het nieuwe platform, dan moet de applicatie herschreven worden in een andere taal.



EMULATIE

Een emulator simuleert in software een andere architectuur in een virtuele computer. Dit maakt het mogelijk om programma's in een bepaalde machinetaal toch uit te voeren in een andere machinetaal door de instructies realtime te vertalen.

Voorbeelden van emulatie vindt men terug bij de overstap van Apple van de PowerPC processoren naar de Intel processoren en bij de Transmeta processor die Pentium instructies vertaalt naar eigen instructies.

INTERPRETERS

Zie interpreters in het appendix.

Voorbeelden:

- HTML
- Postscript
- PDF
- Perl
- Lisp
- Prolog
- PHP
- …

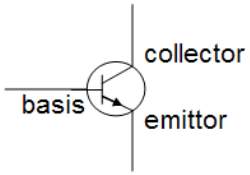
JAVA EN .NET

Dit zijn speciale tussenvormen in emulatie en interpreting van respectievelijk Sun en Microsoft. Beiden hebben een eigen specificatie voor een machinetaal, respectievelijk bytecode en MSIL. Deze machinetaal wordt niet ondersteund door een bepaalde processor, maar wordt eerder uitgevoerd op een geëmuleerde processor. Bij Java wordt de bytecode tijdens runtime vertaalt naar machinecode en bij .NET compileert de Just In Time compiler (JIT) de MSIL naar machinecode.

Door het porteren van de Java Virtual Machine en de .NET Framework naar andere platformen, is het mogelijk om dezelfde applicaties zonder hercompileren te draaien op deze platformen. We bereiken dus bytecode-compatibiliteit en MSIL-compatibiliteit tussen verschillende platformen.

TRANSISTOREN

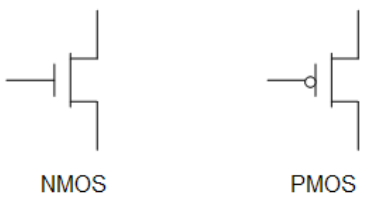
Een transistor is een schakelaar zoals het relais, maar kan veel kleiner geproduceerd worden en bevat geen mechanische of bewegende delen. Transistoren worden gemaakt in halfgeleidertechnologie en zijn het basiscomponent van de digitale elektronica.



Indien er stroom loopt in de basis, dan kan er stroom vloeien tussen de collector en de emittor. Deze component gedraagt zich met andere woorden als een schakelaar.

MOS TRANSISTOREN

In nageneog alle chips worden tegenwoordig MOSFET (metal-oxide semiconductor field-effect) transistoren gebruikt. Deze bestaan in twee varianten: NMOS en PMOS.



De NMOS transistor geleidt indien de spanning op de ingang voldoende hoog is. De PMOS daarentegen geleidt indien de spanning op de gate laag genoeg is. Technologisch gezien blijkt de NMOS geschikt voor verbindingen met de massa en de PMOS voor verbindingen met de voedingsspanning. Men zal ze dan ook vaak in paren tegenkomen.

Bij een chip met beide types spreekt men van CMOS (complementary MOS).

LOGISCHE OPERATIES EN POORTEN

Dit zijn operaties in de boolese algebra, de binaire rekenkunde. We beschouwen monadische en dyadische operatoren met respectievelijk 1 operand en 2 operandi. Met behulp van een waarheidstabel kunnen we deze operaties volledig definiëren door alle mogelijke inputs te associëren met hun output.

AND

Een dyadische operatie, geeft 1 als beide inputs 1 zijn, anders 0. Dit is een commutatieve operatie. Men kan een van beide operandi beschouwen als een masker. Enkel de bits die in het masker op 1 staan, worden van het andere operand doorgelaten. Op plaatsen waar de masker 0 is, komt in de output ook een 0.

Waarheidstabel:

O1	O2	O1 AND O2
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

Voorbeeld:

0100 0101

0000 1110

0000 0100

Logische voorstelling:

OR

Een dyadische operatie, geeft 1 als een of beide van de inputs 1 zijn, anders 0. Dit is een commutatieve operatie. Ook hier kan men een van beide operandi als masker beschouwen. Op plaatsen waar de masker 0 is, komt de waarde van het andere operand te staan. Waar de waarde van het masker 1 is, komt altijd een 1.

Waarheidstabel:

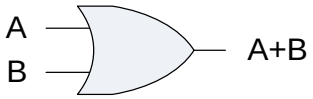
O1	O2	O1 OR O2

0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

Voorbeeld:

0100 0101
0000 1110
0100 1111

Logische voorstelling:



XOR

Een dyadische operatie, geeft 1 als één van beide inputs 1 is, anders 0. Dit is een commutatieve operatie. Opnieuw kunnen we een masker herkennen. Waar het masker een 0 heeft, wordt de output overgenomen van het andere operand. Waar het masker een 1 heeft, wordt de output van het andere operand geïnverteerd.

Merk op dat (A XOR B) XOR B = A. Bovendien is A XOR A = 0. Bovendien kan de XOR operatie ook gezien worden als de modulo-2 optelling van de inputs, of als het tellen van het aantal 1-bits in de input en 1 teruggeven als dit oneven is en 0 als dit even is.

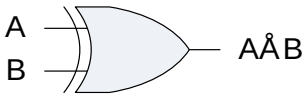
Waarheidstabel:

O1	**O2**	**O1 XOR O2**
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

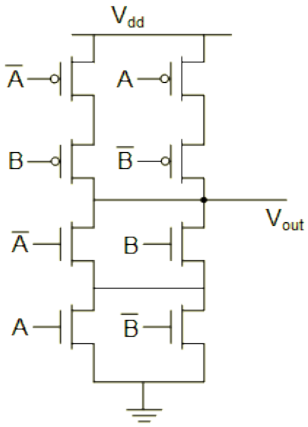
Voorbeeld:

0100 0101
0000 1110
0100 1011

Logische voorstelling:



Een XOR poort kan gerealiseerd worden met het volgende transistornetwerk:



Enkel als ofwel $\bar{A}B$ ofwel $A\bar{B}$ geleiden de bovenste transistoren en is de output hoog. In alle andere gevallen geleiden de bovenste transistoren niet en de onderste wel.

Aangezien de afzonderlijke negaties van A en B ook geconstrueerd moeten worden met behulp van invertoren, is deze poort complexer dan de NAND of NOR poort.

NOT

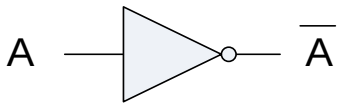
Een monadische operatie, geeft 1 als de input 0 is, anders 0. Ook soms invertor genoemd.

Een NOT poort kan ook gerealiseerd worden met een NAND poort: O1 NAND O1.

Waarheidstabel:

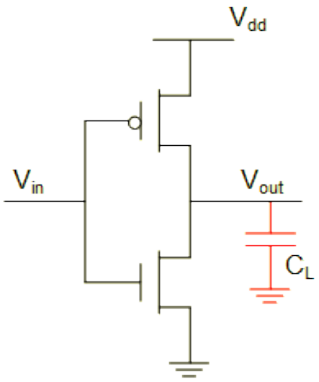
O1	NOT O1
0	**1**
1	**0**

Logische voorstelling:



Deze voorstelling is dezelfde als een buffer met een bolletje voor. Bolletjes op een in- of output geven aan dat het inverse signaal gebruikt wordt.

Een NOT poort kan gerealiseerd worden via het volgende transistornetwerk:



Als de input hoog is, verbindt de NMOS de uitgang met de massa en spert de PMOS de voeding. Als de input laag is, verbindt de PMOS de uitgang met de voeding en spert de onderste transistor de massa.

De capaciteit aan de uitgang is in werkelijkheid niet aanwezig, maar modelleert de capaciteit van de uitgaande bedrading. Indien er niet geschakeld wordt, verbruikt het netwerk geen stroom, bij het schakelen wordt stroom gebruikt om de parasitaire capaciteiten van de bedrading te laden en te ontladen. Dus hoe hoger de klokfrequentie hoe hoger het stroomverbruik.

IMPLICATIE ( )

Een dyadische operatie, geeft 0 als de eerste operand 1 is en de tweede 0, anders 1. Deze operatie is niet commutatief. Geeft overal waarde masker 0 is de inverse van de input, anders 1.

Equivalent met (NOT O1) OR O1.

Waarheidstabel:

O1	O2	O1 O2
0	0	**1**
0	1	**1**
1	0	**0**
1	1	**1**

Voorbeeld:

0100 0101
0000 1110
1011 1110

BUFFER

Een monadische operatie, geeft als output hetzelfde als de input. Dit is de identiteitsfunctie. Hoewel dit geen logische betekenis heeft, spelen buffers een belangrijke rol als signaalversterker.

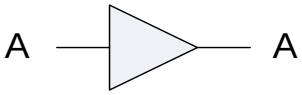
Een signaal moet soms versterkt worden wanneer het over een lange afstand moet reizen of wanneer het als input dient voor meerdere poorten.

Een buffer kan ook gerealiseerd worden met een AND poort: O1 AND O1, of met een OR poort: O1 OR O1.

Waarheidstabel:

O1	O1
0	**0**
1	**1**

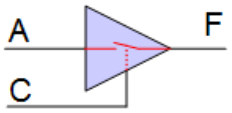
Logische voorstelling:



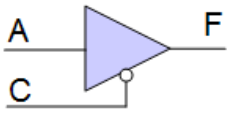
TRI-STATE BUFFERS

Een tri-state buffer gedraagt zich als een gewone buffer, maar heft een extra controle-ingang die toelaat de buffer af te koppelen. Zo heeft men 3 mogelijke outputwaarden: hoog, laag en afgekoppeld. Het verschil tussen laag en afgekoppeld is dat bij laag de uitgang nog steeds verbonden is aan het elektrische netwerk.

Er bestaan 2 versies: een met directe controle en een met geïnverteerde controle.



C	A	F
0	0	∅
0	1	∅
1	0	0
1	1	1



C	A	F
0	0	0
0	1	1
1	0	∅
1	1	∅

Men kan een tri-state buffer gebruiken om kortsluiting te voorkomen in een elektrisch netwerk.

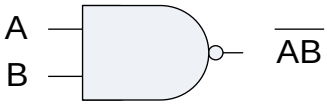
NAND

Voert NOT AND uit. Het resultaat is dus de inverse van de AND operatie.

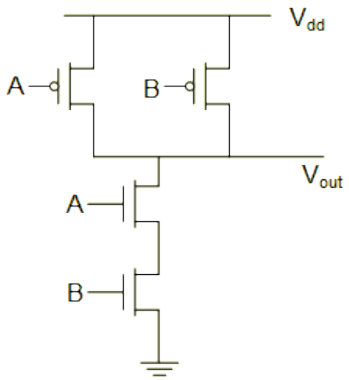
Waarheidstabel:

O1	O2	O1 NAND O2
0	0	**1**
0	1	**1**
1	0	**1**
1	1	**0**

Logische voorstelling:



Een NAND poort kan gerealiseerd worden via het volgende transistornetwerk:



Enkel als zowel A als B spanning krijgen zullen de onderste NMOS transistoren geleiden en V_{out} laag maken. In alle andere gevallen geleidt een van de bovenste PMOS transistoren en wordt de uitgang hoog.

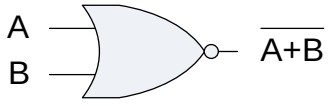
NOR

Voert NOT OR uit. Het resultaat is dus de inverse van de OR operatie.

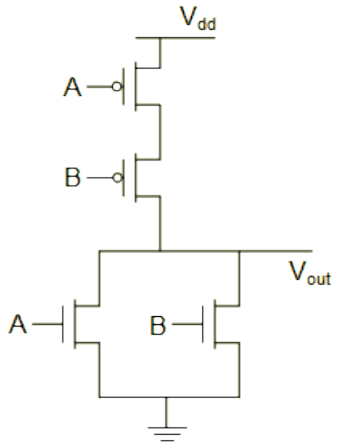
Waarheidstabel:

O1	O2	O1 NOR O2
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**0**

Logische voorstelling:



Een NOR poort kan gerealiseerd worden met het volgende transistornetwerk:



Enkel als zowel A als B zonder spanning staan geleiden de onderste twee NMOS transistoren niet en wordt de uitgang hoog. In de andere gevallen is de uitgang laag.

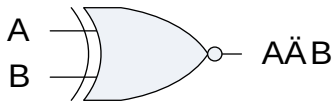
XNOR

Voert NOT XOR uit. Het resultaat is dus de inverse van de XOR operatie.

Waarheidstabel:

O1	O2	O1 XNOR O2
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

Logische voorstelling:



De XNOR poort valt te construeren door de onderste en bovenste inputs van XOR te verwisselen.

BOOLESE ALGEBRA

De boolese algebra helpt ons bij het vereenvoudigen van netwerken met logische poorten. De volgende tabel geeft de basiswetten van de boolese algebra en de duale versies ervan (waar alle AND door OR is vervangen en vice versa):

Naam	Wet
Commutativiteit	$AB = BA$ $A + B = B + A$
Distributiviteit	$A(B + C) = AB + AC$

	$A + (BC) = (A + B)(A + C)$
Neutraal element	$1A = A$ $0 + A = A$
Complement	$A\bar{A} = 0$ $A + \bar{A} = 1$
Nuleigenschap **Ééneigenschap**	$0A = 0$ $1 + A = 1$
Idempotentie	$AA = A$ $A + A = A$
Associativiteit	$A(BC) = (AB)C$ $A + (B + C) = (A + B) + C$
Dubbele negatie	$\bar{\bar{A}} = A$
De Morgan	$\overline{AB} = \bar{A} + \bar{B}$ $\overline{A + B} = \bar{A}\bar{B}$
Consensus	$AB + \bar{A}C + BC = AB + \bar{A}C$ $(A + B)\,(\bar{A} + C)\,(B + C) = (A + B)(\bar{A} + C)$
Absorptie	$A + (AB) = A$

LOGISCHE NETWERKEN

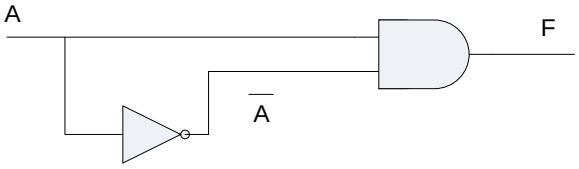
POORTVERTRAGING

Er is altijd een zekere tijd nodig vooraleer de uitgang van een poort de correcte waarde aanneemt na vearndering van de ingang. De vertraging die optreedt heet de poortvertraging of propagatietijd.

In de praktijk blijkt de vertraging ook af te hangen van de elektrische belasting op de uitgang van de poort. Een gangbare methode om de poortvertraging aan te geven is de FO4 of fan-out-of-4. Dit is de vertraging die een invertor ondervindt als hij vier identieke invertoren dient aan te sturen.

Met elke technologiegeneratie neemt de poortvertraging verder af. Men kan op dit moment uitgaan van een poortvertraging van ongeveer 10 tot 15 ps.

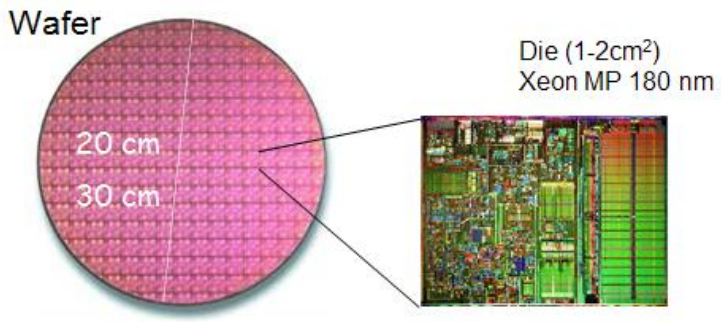
Beschouw als illustratie van het gevaar van poortvertragingen de volgende opstelling:



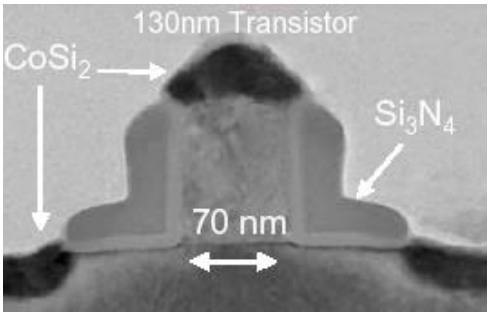
We zouden verwachten dat F steeds laag is. Maar op het moment dat A wisselt van 0 naar 1 kan door de poortvertraging in de invertor F een korte sprong vertonen. Over een tijdspanne van de poortvertraging is de ene ingang van de AND poort al 1 terwijl de 0 op de andere ingang nog moet propageren. Dit heet een glitch. Dit meestal ongewenste gedrag kan ook gebruikt worden om een pulsgenerator te bouwen.

PRODUCTIE VAN CHIPS

Chips worden gefabriceerd op wafers, platte ronde siliciumplaten. Wafers zijn typisch 20 tot 30 cm in diameter. Elke afzonderlijke chip op de wafer noemt men een die.



Met behulp van lithografie worden verschillende maskers aangebracht op het siliciumsubstraat. De chip bestaat uit miljoenen transistors onderling verbonden door metaalbaantjes in verschillende niveau's.

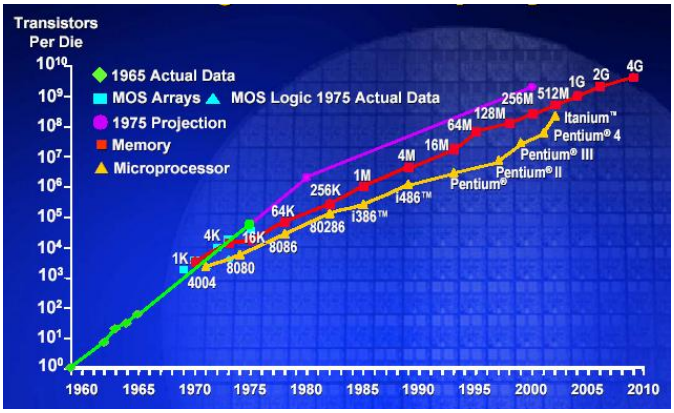


De oxidelaag tussen de gate en het kanaal van de stroom is bij 90 nm lithografie ongeveer 3 atomen dik.

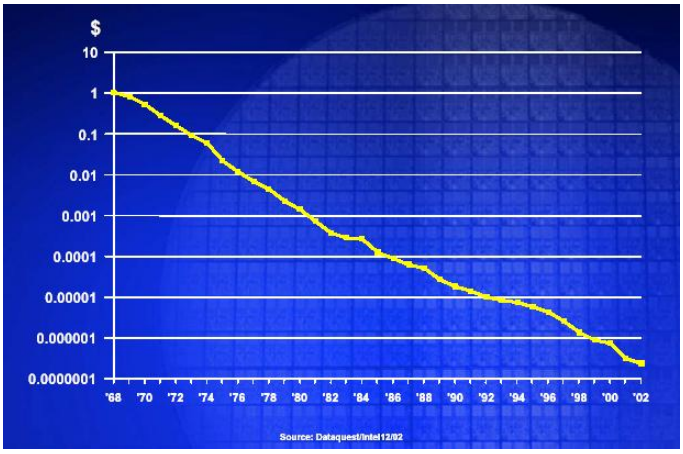
Na de productie van de wafer worden de dies eruit gesneden en in een chipverpakking gestoken.

WET VAN MOORE

De Wet van Moore stelt dat het aantal transistors per chip verdubbelt om de twee jaar.



Deze exponentiële stijging is in verband te brengen met de exponentiële daling in prijs per transistor:



REALISATIE VAN BOOLESE FUNCTIES

Boolese functies kunnen gerealiseerd worden via een som van producten of een product van sommen.

SOM VAN PRODUCTEN

Elke rij in de waarheidstabel met als resultaat 1 levert een minterm op. Een minterm is een productterm die elke variabele precies één keer bevat, ofwel direct ofwel geïnverteerd. De variabele komt direct voor indien ze in de rij een 0 voorstelt en anders geïnverteerd. De som van alle mintermen levert de boolese functie.

De som van producten kan eenvoudig geïmplementeerd worden met NOT poorten en multi-input AND poorten of multi-input OR poorten.

PRODUCT VAN SOMMEN

Elke rij in de waarheidstabel met als resultaat 0 levert een maxterm op. Een maxterm is een som waarin elke variabele precies een keer voorkomt, ofwel direct ofwel geïnverteerd. De variabele komt direct voor indien ze in de rij een 1 voorstelt en anders geïnverteerd. Het product van alle maxtermen levert de boolese functie.

Het product van sommen kan eenvoudig geïmplementeerd worden met NOT poorten en multi-input AND poorten of multi-input OR poorten.

MINIMALISATIE

Het process van minimalisatie probeert een boolese functie zoveel mogelijk te vereenvoudigen. Men gaat op zoek naar de eenvoudigste implementatievorm.

De complexiteit van een realisatie kan kwantitatief uitgedrukt worden door een telling van het totaal aantal poorten, het totaal aantal inputs, de maximale fan-out en de maximale fan-in.

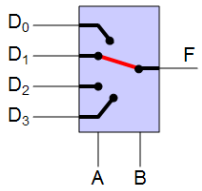
Hier is ook het concept van computationele compleetheid belangrijk, dit stelt dat men alle boolese functies kan realiseren met NAND poorten. Dit geldt ook voor de NOR poort.

DIGITALE COMPONENTEN

Hoog-niveau digitale circuits worden normal opgebouwd uit componenten in plaats van individuele poorten. Componenten zijn verzamelingen van poorten met een specifiek en vaak gebruikt gedrag.

DE MULTIPLEXER

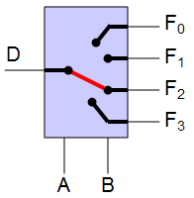
Een multiplexer is een component dat een aantal ingangen omzet tot één enkele uitgang. Door middel van controlelijnen kan men een bepaalde input selecteren als output.



Multiplexers kunnen gebruikt worden om willekeurige boolese functies te implementeren. Leg de inputs aan de controlelijnen en leg de outputs als constanten aan de inputs van de multiplexer. Het resultaat is de boolese functie.

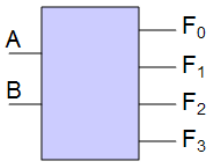
DE DEMULTIPLEXER

Een demultiplexer geeft een input door naar een van de outputkanalen afhankelijk van de controlelijnen. De demultiplexer wordt gebruikt om de data van een enkele bron naar één bestemming uit een groep van mogelijke bestemmingen te sturen.



DE DECODER

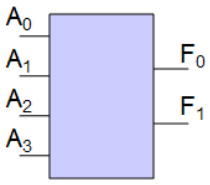
Een decoder vertaalt een code in een spatiale locatie. De inputsignalen beslissen welke van de outputs op 1 komt te staan. Op elk ogenblik staat één van de outputs op 1.



Dit is analoog aan een demultiplexer met 1 als ingangssignaal.

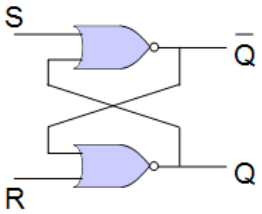
DE PRIORITEITSCODEERDER

Een prioriteitscodeerder vertaalt een aantal inputs in een binaire codering. Hij kan gezien worden als de inverse van de decoder. Elke input heeft een rangorde en de uitgang geeft de binaire code van de ingang met de hoogste prioriteit die is ingeschakeld.



Dit component wordt onder meer gebruikt om te bepalen wie als eerste een apparaat mag gebruiken.

S-R LATCH



Wanneer S en R 0 zijn, bewaart de geheugencel de waarden van Q en NOT Q.

Is Q=0 en R=S=0 en brengen we S naar 1 dan zal na een periode t de uitgang NOT Q omschakelen naar 0 en na een periode van 2 t is Q=1. Dit is de set operatie, na 2 t kunnen we het S signaal terug naar 0 brengen zonder dat Q wijzigt.

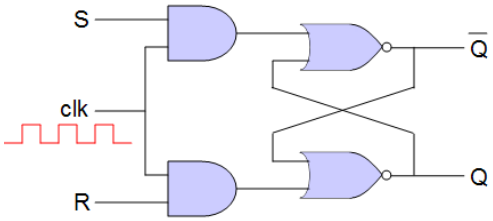
Is Q=1 en R=S=0 en brengen we R naar 1 dan zal na een periode t de uitgang Q omschakelen naar 0 en na een periode van 2 t is NOT Q=1. Dit is de reset operatie, na 2 t kunnen we het R signaal terug naar 0 brengen zonder dat Q wijzigt.

Het op 1 brengen van R en S is niet toegelaten, de uitkomst van Q kan niet voorzien worden.

Dit is de waarheidstabel voor de S-R latch:

Q_t	S	R	Q_{t+1}
0	0	0	**0**
0	0	1	**0**
0	1	0	**1**
0	1	1	-
1	0	0	**1**
1	0	1	**0**
1	1	0	**1**
1	1	1	-

GEKLOKTE S-R LATCH

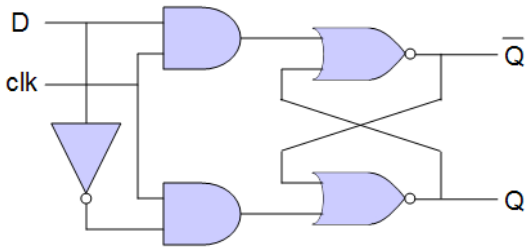


Wanneer S en R afkomstig zijn uit een ander complex circuit, is het mogelijk date r een aantal ongewenste overgangen of glitches gebeuren op S of R. Wanneer een glitch de waarde van Q zou veranderen is de werking van de latch verstoord.

Indien we een klok toevoegen kunnen we ervoor zorgen dat de bit enkel kan veranderen indien S en R stabiel zijn. Men moet er dan voor zorgen dat S en R gegarandeert stabiel zijn op het moment dat de klok hoog is.

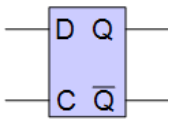
Men noemt geklokte latches niveaugestuurd aangezien ze hun toestand enkel wijzigen wanneer de klok hoog of de klok laag is.

D LATCH



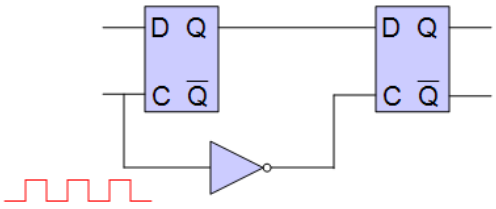
Een nadeel aan de S-R latch is dat men voor set en reset 2 verschillende lijnen heeft. Een alternatieve configuratie is de D latch waarbij er maar 1 data input is.

De logische voorstelling van een D latch is:



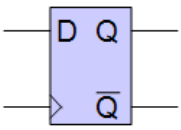
De D latch wordt vaak gebruikt in situaties waar er feedback is van een uitgang van een circuit naar een ingang. De feedback zorgt er voor dat de latch soms meer dan een keer per klokcyclus verandert. Indien men zeker wil zijn dat de latch slechts eenmaal per klokcyclus verandert van toestand, gebruikt men een master-slave D latch of D flip-flop.

D FLIP-FLOP



De D flip-flop bestaat uit twee D latches in cascade waarbij de tweede een geïnverteerde klok gebruikt. Wanneer de klok hoog is past de master zijn waarde aan, terwijl de slave zijn waarde pas aanpast als de klok laag is. De slave uitgang is nu gegarandeerd stabiel. De klok moet eerst hoog en laag gaan vooraleer de waarde wijzigt.

De logische voorstelling van een D flip-flop:

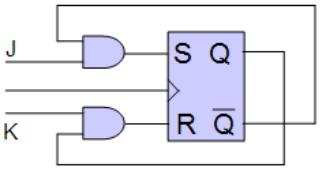


Men noemt flip-flops flankgestuurd aangezien ze alleen van waarde veranderen bij een kloktransitie van hoog naar laag of van laag naar hoog.

De tijd nodig door de slave latch tijdens de kloktransitie om een stabiele waarde aan te nemen heet de setuptijd. Gedurende deze periode moet de waarde van de master latch stabiel blijven.

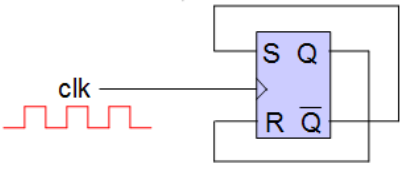
J-K FLIP-FLOP

Een J-K flip-flop is gelijkaardig aan een S-R flip-flop. Het verschil zit hem in de terugkoppeling.



Deze aanpassing in de terugkoppeling zorgt ervoor dat als J en K beide op 1 staan, de waarde van Q omgewisseld wordt. Deze flip-flop realiseert dus de 4 operaties die men op een bit kan uitvoeren: zet op 1, zet op 0, laat onveranderd en omklappen.

T FLIP-FLOP



De T flip-flop is al seen J-K flip-flop waarbij J=K=1 en klapt dus per klokperiode 1 keer om. Het effect is dat de waarde van Q opnieuw een kloksignaal is met halve frequentie. Door een aantal van deze flip-flops in serie te schakelen kan men de klokfrequentie delen door machten van 2.

GEGEVENSREPRESENTATIES

BINAIRE EENHEDEN

Onderstaande tabel geeft een aantal termen weer die vaak gebruikt worden als binaire eenheden.

4 bits	1 nibble		
8 bits	2 nibbles	1 byte	
32 bits	8 nibbles	4 bytes	1 woord

Hier hebben we voor een woord 4 bytes gebruikt. Dit is echter niet vast gedefinieerd en men komt dus ook architecturen tegen met 2 of 8 byte-woorden.

HEXADECIMALE NOTATIE

Verkorte notatie voor 4 bits of 1 nibble. Dit zijn alle mogelijkheden:

Binair	Decimaal	Hexadecimaal
0000	0	**0**
0001	1	**1**
0010	2	**2**
0011	3	**3**
0100	4	**4**
0101	5	**5**
0110	6	**6**
0111	7	**7**
1000	8	**8**
1001	9	**9**
1010	10	**A**
1011	11	**B**
1100	12	**C**
1101	13	**D**
1110	14	**E**
1111	15	**F**

OCTALE NOTATIE

Met 3 bits kunnen we een gelijkaardig systeem opstellen als voor hexadecimale notatie, hier gebruiken we echter maar de cijfers van 0-7.

NATUURLIJKE GETALLEN

BINAIRE CODERING

Deze codering is gelijkaardig met het tiendelige talstelsel maar dan met slechts 2 symbolen.

Met de volgende formule kunnen we uit een gecodeerde voorstelling $a_{n-1}a_{n-2} \dots a_1 a_0$ de decimale voorstelling bepalen:

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

Het bereik is: $[0, 2^n - 1]$
Het aantal verschillende waarden is: 2^n

Voorbeeld:

$$156 = 2^7 + 2^4 + 2^3 + 2^2 = 10011100$$

BCD CODERING

Binary Coded Decimal (binair gecodeerde decimalen) is een codering waarbij elk decimaal cijfer van het getal gecodeerd wordt naar een nibble (packed) of 2 nibbles (unpacked) volgens het binaire algoritme. Deze codering wordt vooral gebruikt in de financiële sector en databasetoepassingen waar men graag getallen heeft met een vaste lengte.

Voorbeeld:

$$156 = \langle 1|5|6\rangle = \langle 0001|0101|0110\rangle$$

EXCESS-3 CODERING

Dit alternatief voor BCD-codering heeft 2 voordelen:

- Een optelling genereert automatisch een overdracht naar de hogere rang
- Het berekenen van een complement $x \to (9 - x)$ kan eenvoudig door de individuele bits te complementeren

Er zijn ook echter 2 nadelen aan verbonden:

- De interpretatie is minder intuïtief
- De codering is niet gewogen, dit wil zeggen dat er geen gewichten per rang zijn waaruit de uiteindelijke waarde te bepalen is

De werking is als volgt:

Decimaal	BCD	Excess-3
0	0000	**0011**
1	0001	**0100**
2	0010	**0101**
3	0011	**0110**
4	0100	**0111**

5	0101	**1000**
6	0110	**1001**
7	0111	**1010**
8	1000	**1011**
9	1001	**1100**

UPC-CODERING

UPC of Universal Product Codes worden gebruikt bij barcodes. De codering gebruikt 7 bits per cijfer en voor elk cijfer bestaan er 2 varianten. Zelfs dan worden maar 20 van de 128 verschillende bitpatronen gebruikt.

Decimaal	Linkercode	Rechtercode
0	0001101	1110010
1	0011001	1100110
2	0010011	1101100
3	0111101	1000010
4	0100011	0011100
5	0110001	0001110
6	0101111	1010000
7	0111011	1000100
8	0110111	1001000
9	0001011	1110100

REFLECTIEVE GRAY CODES

In de meeste coderingen is het zo dat bij het sequentieel doorlopen van de waarden vaak een aantal bits tegelijk moeten aangepast worden. Dit is een nadeel wanneer men niet kan garanderen dat deze aanpassingen in één bewerking kunnen gebeuren. Gray codes bieden hier een oplossing aangezien bij elke overgang maar 1 bit verandert.

Gray codes kunnen algemeen beschreven worden voor n bits, hier tonen we de waarden voor G_3codering:

Decimaal	Gray code
0	**000**
1	**001**
2	**011**
3	**010**
4	**110**

5	**111**
6	**101**
7	**100**

Door 1 bit aan te passen bij de laatste waarden beginnen we terug bij de eerste waarde.

GEHELE GETALLEN

SIGNED BINAIRE CODERING

Deze codering is identiek aan binaire codering, met als verschil dat er een extra bit vooraan staat die aangeeft of het getal negatief of positief is, een tekenbit. Een 0 betekent een positief getal, een 1 een negatief getal.

Er zijn enkele nadelen verbonden aan deze codering:

- Er zijn twee mogelijke voorstellingen voor 0: 0000 en 1000, dit is een probleem aangezien dit het vergelijken van getallen moeilijker maakt
- Bovendien zijn de rekenregels voor positieve en negatieve getallen verschillend
- Ten laatste is het veranderen van de precisie niet eenvoudig: men moet eerst de tekenbit weghalen, de precisie aanpassen en dan de tekenbit terugplaatsen

Met de volgende formule kunnen we uit een gecodeerde voorstelling $a_{n-1}a_{n-2}\dots a_1a_0$ de decimale voorstelling bepalen:

$$A = (-1)^{a_{n-1}} \sum_{i=0}^{n-2} 2^i a_i$$

Het bereik is: $[-(2^{n-1}-1), 2^{n-1}-1]$
Het aantal verschillende waarden is: $2^n - 1$

1-COMPLEMENT CODERING

Deze codering beschouwt de negatieve getallen als de binaire inverse van de binaire codering van het positieve getal. Opnieuw zien we vooraan de representatie een tekenbit die een 0 heeft voor een plusteken en een 1 voor een minteken.

Voordelen van 1-complement codering:

- Men kan gemakkelijker 1 optellen bij een getal dan bij signed binaire codering, behalve in de buurt van 0
- De precisie vergroten kan eenvoudig door de tekenbit links te herhalen
- De precisie verlagen is ook eenvoudig door een aantal beginbits weg te laten, onder voorwaarde dat de tekenbit dezelfde blijft

Er zijn opnieuw twee voorstellingen voor het getal 0: 0000 en 1111. Deze codering wordt in de praktijk dan ook weinig gebruikt.

Het bereik is: $[-(2^{n-1}-1), 2^{n-1}-1]$
Het aantal verschillende waarden is: $2^n - 1$

2-COMPLEMENT CODERING

Deze codering wordt in de praktijk het meest gebruikt voor gehele getallen. De werking is bijna identiek aan 1-complement codering met het verschil dat het complement berekent wordt volgens:

$$-A = \backsim (A - 1)$$

We zien dat er nu slechts een voorstelling is voor het getal 0: 0000. De representatie 1111 betekent nu -1. Bovendien blijven dezelfde voordelen als voor 1-complement codering gelden:

> We kunnen gewoon de binaire optelling maken, het resultaat is correct en in 2-complement voorstelling
> De precisie vergroten kan eenvoudig door de tekenbit links te herhalen
> De precisie verlagen is ook eenvoudig door een aantal beginbits weg te laten, onder voorwaarde dat de tekenbit dezelfde blijft

Het bereik is: $[-(2^{n-1}), 2^{n-1} - 1]$
Het aantal verschillende waarden is: 2^n

BIASED CODERING

Ook wel verschoven voorstelling of excess representation genoemd. De getallen worden geïnterpreteerd als natuurlijke getallen, maar worden verschoven door een bias aft e trekken van hun waarde. Zo wordt het kleinste numerieke bitpatroon 0000 toegekend aan de kleinste voorgestelde waarde.

Een voordeel van deze notatie is dat het eenvoudig is om getallen te vergelijken. Een kleiner getal heeft steeds een kleiner bitpatroon. Bovendien is het bereik nu vrij te kiezen. Als nadeel worden andere berekeningen wel ingewikkelder.

Met de volgende formule kunnen we uit een gecodeerde voorstelling $a_{n-1}a_{n-2} \dots a_1 a_0$ de decimale voorstelling bepalen:

$$A = \sum_{i=0}^{n-1} 2^i a_i - B$$

Het bereik is: $[-B, 2^n - 1 - B]$
Het aantal verschillende waarden is: 2^n

BCD CODERING

De 1-complement en 2-complement coderingen zijn te veralgemenen naar een willekeurige basis N. We noemen dan de N-complement codering de basiscomplementnotatie en de (N-1)-complement codering de verminderde basiscomplementnotatie.

We kunnen deze veralgemeende complementnotaties gebruiken bij het coderen van gehele getallen met behulp van BCD codering. Hier zijn drie mogelijkheden:

SIGNED BCD CODERING

Bij deze codering wordt voor de BCD representatie een tekenbit geplaatst. Dit wordt bijna nooit gebruikt.

9-COMPLEMENT CODERING

Deze notatie is de verminderde basiscomplementnotatie voor het tiendelig talstelsel. Postieve getallen worden normaal gecodeerd, negatieve volgens 9999 – A. Positieve getallen hebben dus een eerste decimaal kleiner dan 5, negatieve een eerste decimaal groter of gelijk aan 5.

10-COMPLEMENT CODERING

Deze notatie is de basiscomplementnotatie voor het tiendelig talstelsel. Dit komt dus neer op de 9-complement codering + 1.

REËLE GETALLEN

FIXED-POINT CODERING

Geef de komma in het getal een vaste plaats in het bitpatroon en hou daar rekening mee bij de bewerkingen. Het bereik mag niet te groot zijn. Dit is vooral interessant bij BCD notaties.

FLOATING-POINT CODERING

Deze codering laat een heel groot bereik toe met een relatief klein aantal bits. Het bitpatroon bestaat uit de volgende onderdelen:

Een tekenbit (S)
Een exponent (E), de grootte hiervan bepaalt het bereik
Een mantisse (M), de grootte hiervan bepaalt de precisie

De waarde van een bitpatroon volgt uit:

$$A = (-1)^S \cdot M \cdot 2^E$$

De exponent en mantisse worden apart gecodeerd als volgt. Stel dat de exponent α bits groot is. We coderen dan de exponent met bias codering met bias $2^{\alpha-1} - 1$. De waarden met enkel nullen en enen zijn voorbehouden voor speciale getallen. De mantisse slaat men op als een genormaliseerde binaire fractie. Dit betekent dat het getal een 1 voor de komma heeft, die niet meegecodeerd wordt. Deze bit heet de “hidden bit”. Voor het getal 0 bestaat een speciale regeling. Men bepaalt de decimale waarde van de mantisse als volgt:

$$A = 1 + \sum_{i=0}^{n-1} 2^{i-n} a_i$$

De speciale reservaties van de exponenten met enkel nullen of enkel enen worden gebruikt om speciale getallen voor te stellen. Zonder deze speciale getallen zou het bijvoorbeeld niet mogelijk zijn om het getal 0 voor te stellen, een ernstige tekortkoming. Als oplossing voor dit probleem zijn de volgende conventies geïntroduceerd:

Clean zero: enkel nul bits in de exponent en de mantisse, de tekenbit is 0 of 1, dit zijn twee voorstellingen voor het getal 0
Dirty zero: enkel nul bits in de exponent, de interpretatie is dat de exponent 1 is, maar de hidden bit voor het kommatgetal van de mantisse wegvalt, de mantisse is dus nu < 1

Infinity: enkel 1 bits in de exponent en enkel 0 bits in de mantisse, deze notaties worden gebruikt om $+\infty$ en $-\infty$ voor te stellen, dit wordt gebruikt indien het resultaat van een bewerking niet gerepresenteerd kan worden
Not a Number (NaN): enkel 1 bits in de exponent en de mantisse is verschillend van 0, wordt gebruikt bij $\frac{0}{0}$, $\frac{\infty}{\infty}$ of bij het nemen van vierkantswortels uit negatieve getallen

Door het binair sorteren van floating-point getallen krijgt men een rij van oplopende positieve getallen, gevolgd door een rij aflopende negatieve getallen. De speciale gevallen worden correct gesorteerd in deze lijsten.

Gehele getallen die maximaal evenveel bits nodig hebben als de mantisse, kunnen steeds correct voorgesteld worden. Slechts een klein aantal reële getallen kan exact voorgesteld worden en worden dus afgerond, het testen op gelijkheid tussen reële getallen is dus geen aanrader. Om de gelijkheid tussen twee floats te testen, kan men kijken of het verschil tussen de getallen kleiner is dan de minimale precisie.

Oncodeerbare waarden kunnen op drie manieren afgerond worden:

Naar $\pm\infty$
Naar 0
Naar de dichtst voorstelbare waarde:
- 1 indien > 0,50
- 0 indien < 0,50
- Naar een even mantisse indien = 0,50

De ANSI/IEEE 754 standaard beschrijft twee specifieke formaten. Om de precisie te verhogen kan men nog extra informatie toevoegen:

Een guard bit
Een rounding bit
Een sticky bit

De guard bit en rounding bit vangen waarden op van de mantisse die door bewerkingen naar rechts geshift worden. Als de mantisse nog verder naar rechts shift, wordt de OR van de geshifte waarden opgeslaan in de sticky bit. Deze bits zijn handig in het elimineren van afrondingsfouten door veel sequentiële bewerkingen.

SINGLE PRECISION

Gebruikt 32 bits:

1 bit voor de tekenbit S
8 bits voor de exponent E
23 bits voor de mantisse M

Gebruikte exponenten: 255
Efficiëntie: 99,60%
Aantal decimalen: 6 – 7
Bereik: 10^{-38} – 10^{38}

DOUBLE PRECISION

Gebruikt 64 bits:

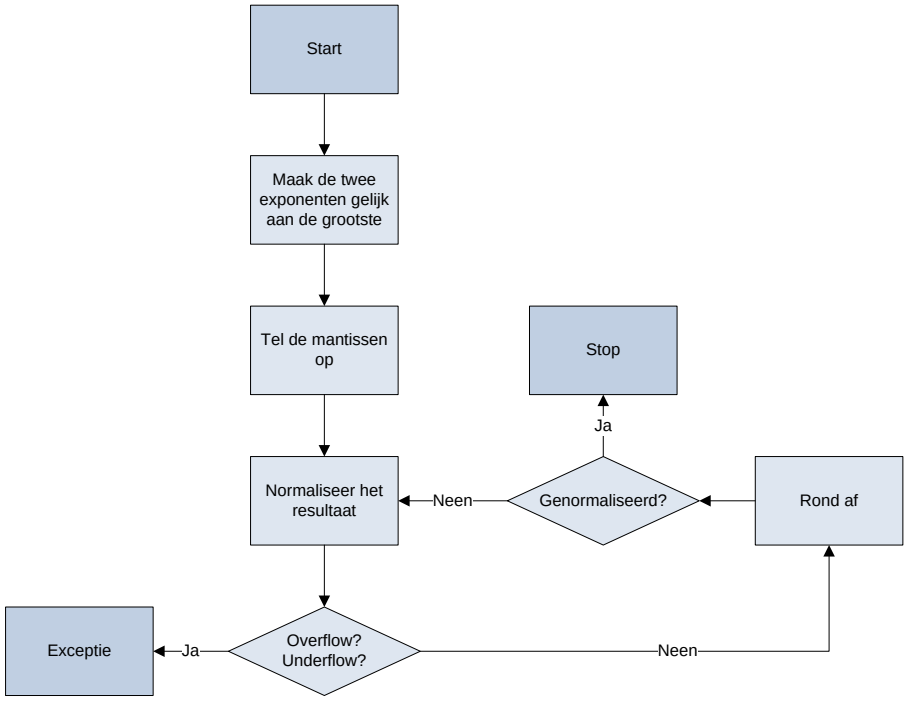
1 bit voor de tekenbit S

11 bits voor de exponent E
52 bits voor de mantisse M

Gebruikte exponenten: 2047
Efficiëntie: 99,95%
Aantal decimalen: 15 – 16
Bereik: 10^{-308} – 10^{308}

OPTELLINGEN

Dit is een stroomdiagram van het verloop van een optelling van twee floats:



STRINGS

Lettertekens zijn een eindige verzameling van tekens en kunnen eenvoudig voorgesteld worden met behulp van bitpatronen door gebruik te maken van gestandaardiseerde tabellen of codes. We bespreken een aantal van deze codetabellen:

ASCII

Dit is de American Standard Code for Information Exchange en gebruikt bitpatronen van 7 bit. Alle 128 mogelijke patronen stellen een letter-, lees- of controlekarakter voor. Sorteren is eenvoudig aangezien opeenvolgende letter- en cijferreeksen ook opeenvolgende bitreeksen zijn. Vaak wordt een uitbreiding van ASCII gebruikt: ISO-8859-1, deze gebruikt een achtste bit om accentlettertekens te onderscheiden.

Voor een volledige lijst met karakters, zie http://en.wikipedia.org/wiki/ASCII.

EBCDIC

Dit is de Extended Binary Coded Decimal Interchange Code, een 8 bit code vooral toegepast in IBM mainframes.

Voor een volledige lijst met karakters, zie http://en.wikipedia.org/wiki/EBCDIC.

UCS-4

De Universal Character Set is een 32 bit code volgens ISO 10646.

Voor een uitgebreidere beschrijving, zie http://en.wikipedia.org/wiki/UTF-32/UCS-4.

UNICODE

Een subset van UCS-4 die de meeste symbolen uit verschillende talen bevat. Met 16 bit een goede afweging tussen de beperkte functionaliteit van ASCII en de grote opslagvereisten van UCS-4.

Voor een uitgebreidere beschrijving, zie http://en.wikipedia.org/wiki/Unicode.

OPBOUW VAN HET GEHEUGEN

PRESTATIES

De prestaties van geheugen worden gemeten aan de hand van een aantal parameters:

De latentie is de tijd die verstrijkt tussen het aanleggen van een adres en het verschijnen van de eerste byte
De cyclustijd is de totale tijd van een lees- of schrijfcyclus
De bandbreedte is het aantal bytes per seconde dat maximaal kan getransfereerd worden van opeenvolgende locaties

FYSIEK GEHEUGEN

Het fysiek geheugen, hoofdgeheugen of RAM-geheugen (Random Access Memory) bestaat uit een genummerde array van geheugencellen, ook wel basic addressable units (BAU) genoemd. De BAU-cel is de kleinste eenheid die aan de hand van een adres kan aangesproken worden in het geheugen. Het corresponderende nummer voor een BAU-cel heet het adres. Afhankelijk van de architectuur is een BAU-cel N bits groot. Bij moderne architecturen is de grootte meestal 8 bits.

De adresruimte is het bereik van adressen dat kan aangesproken worden met M bits, dit is $[0, 2^M - 1]$. In een 32 bit architectuur kunnen we dus 4 GiB aanspreken.

Om een cel op een bepaald adres in het geheugen te lezen, legt de CVE dat adres aan op de adresbus en via de controlebus wordt een leescommando gegeven. Als resultaat zal het geheugen de gelezen waarden aanleggen op de databus.

Om een cel op een bepaald adres in het geheugen te schrijven, let de CVE dat adres aan op de adresbus en via de controlebus wordt een schrijfcommando gegeven. Tegelijkertijd wordt de te schrijven waarde aangelegd op de databus.

Doorgaans zijn de gevens niet parallel toegankelijk en kan men slechts 1 lees- of schrijfoperatie tegelijk uitvoeren.

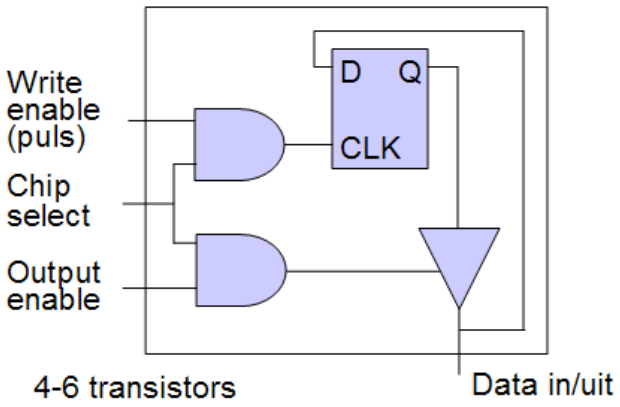
Er zijn twee technologiën voor geheugencellen in het fysiek geheugen:

	Statisch (SRAM)	**Dynamisch (DRAM)**

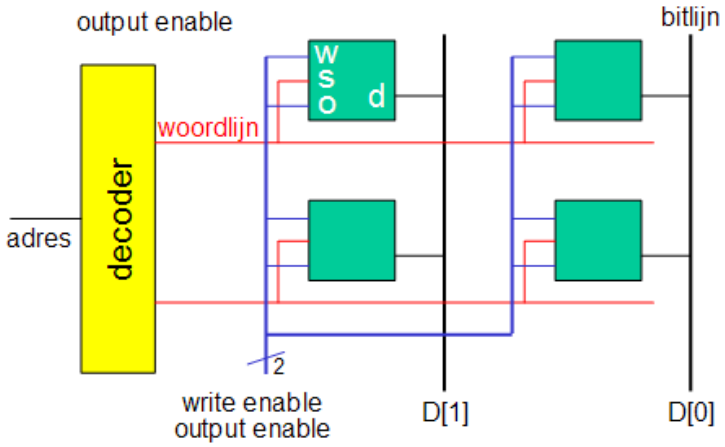
Snelheid	0,2 – 25 ns	30 – 120 ns
Omvang	4 – 6 transistoren	Transistor en condensator
Verbruik	Hoog	Laag
Prijs	Duur	Goedkoop
Toepassing	Caches	Hoofdgeheugen

STATISCH

Gebaseerd op latches, per cel zijn ongeveer 4 tot 6 transistoren vereist wat hem groter maakt dan een dynamische cel



Deze cellen kunnen als volgt in een array gecombineerd worden:

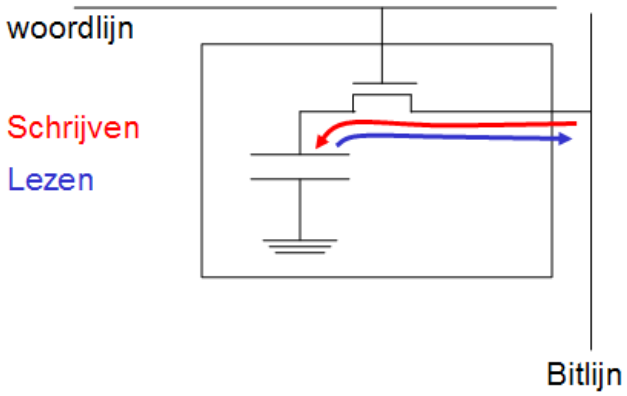


Enkel de chips met een chip select signaal reageren op de write en output signalen. Indien nooit meer dan 1 woordlijn geselecteerd is, zal er maximaal 1 geheugencel een waarde op de datalijn plaatsen.

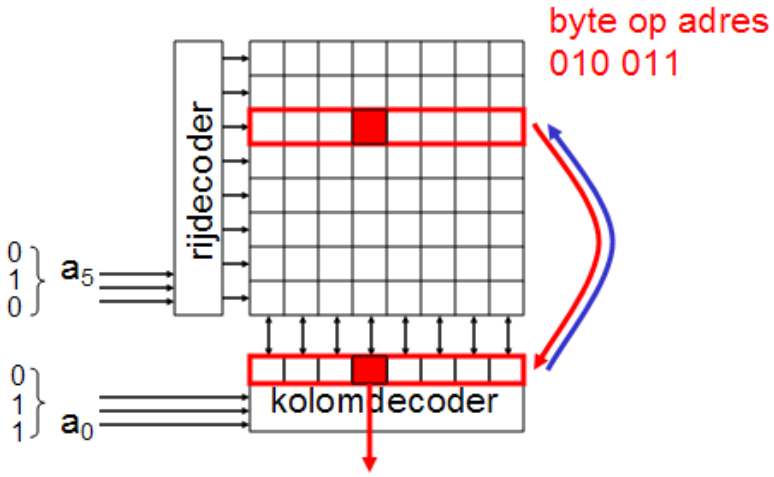
Men kan het geheugen in verschillende configuraties plaatsen, het aantal adresseerbare locaties noemt men de hoogte van het geheugen en het aantal bits per geheugenlocatie de breedte. Door geheugenblokken paralleel te schakelen kunnen we grotere geheugens creëren.

DYNAMISCH

Gebaseerd op condensatoren, de lading van de condensator bepaalt de waarde van de cel.



Men noemt de cellen dynamisch omdat de condensator zijn lading na verloop van tijd verliest. De waarde moet dus periodiek en na elke leesoperatie herschreven worden. Het periodieke herschrijven heet refreshen.

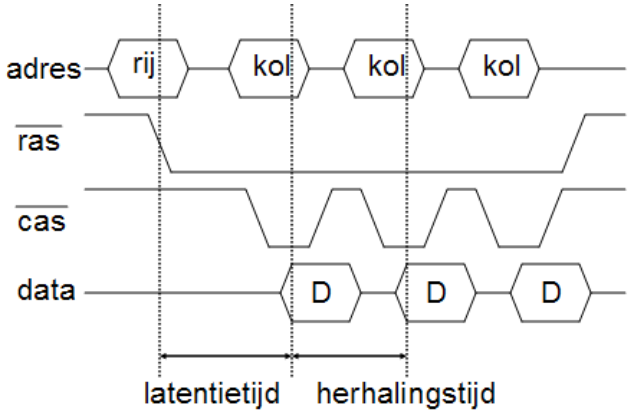


Het adres voor een geheugencel moet in twee tijden aangelegd worden, de helft als rijadres en de andere helft als kolomadres. Per tijd moet dus maar een half adres doorgegeven worden. Een leesoperatie gaat als volgt:

1. Het rijadres wordt aangelegd
2. De rij wordt gelezen en in de rijbuffer geplaatst, de oorspronkelijke waarden zijn vernietigd
3. Vervolgens wordt het kolomadres aangelegd
4. De gewenste byte wordt uit de kolom gelezen
5. De hele rijbuffer wordt teruggeschreven naar de rij

Schrijven gebeurt volledig analoog, de rijbuffer wordt aangepast en weggeschreven. Aangezien dit veel complexer is dan bij statische geheugens is de toegangstijd ook veel lager.

Door de periodieke refresh verliest men maar een aantal procenten aan prestaties, dit gecombineerd met het feit dat deze geheugens kleiner en goedkoper zijn dan SRAM, verklaart dat DRAM de meest gebruikte technologie is voor hoofdgeheugens.



Bovenstaand diagram geeft weer hoe het DRAM geheugen in burstmode gebruikt kan worden. Hierbij wordt een rij in de rijbuffer geladen en daarna meerdere kolommen uitgelezen. Met deze techniek kan de bandbreedte voor sequentiële toegang sterk stijgen.

REGISTERS

De registers bevinden zich op de CVE en zijn een speciale en zeer snelle vorm van geheugen. Registers kunnen in 1 operatie gelezen of geschreven worden. Er zijn registers beschikbaar voor algemeen gebruik, maar er zijn ook speciale registers die de CVE gebruikt om de status te beschrijven.

Het aantal beschikbare registers kan lopen van 8 tot 256 en ze zijn typisch 2 tot 8 bytes groot, afhankelijk van de woordgrootte van de architectuur. Registers zijn zeer duur om te produceren dus men moet het gebruik ervan in processors beperken.

DE PENTIUM PROCESSOR

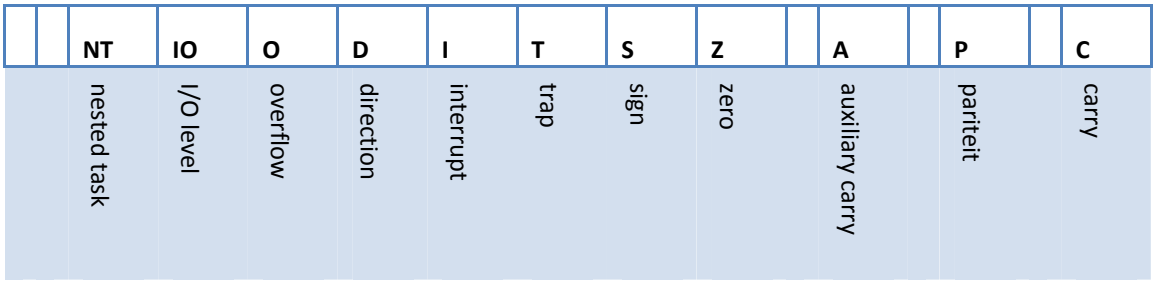
STANDAARDREGISTERS

Hier volgt een tabel die de algemene registers van de Pentium weergeeft:

Register	← 32 →			Omschrijving
EAX		AH	AL	Accumulator
EBX		BH	BL	Basis
ECX		CH	CL	Counter
EDX		DH	DL	Data
ESI		SI		Source index
EDI		DI		Destination index
EBP		BP		Basispointer
ESP		SP		Stack pointer
EIP		IP		Instruction pointer
EFLAGS		FLAGS		Flags

Hierbij is AH&AL = AX en analoog voor EBX, ECX en EDX.

De laagste 16 bits van het EFLAGS register – het FLAGS register – ziet er specifieker als volgt uit:



SEGMENTREGISTERS

Verder vinden we ook nog een aantal specifieke segmentregisters, deze worden echter meestal niet meer gebruikt:

Register	32	Omschrijving
GS	GS	Extra data segment
FS	FS	Extra data segment
SS	SS	Stack segment
ES	ES	Extra data segment
DS	DS	Data segment
CS	CS	Code segment

Het grijze gedeelte duidt op het feit dat de segmentregisters maar 16 bit groot zijn. Ze wijzen naar een segmentdescriptor in de laagste 64 KiB van het geheugen. Deze segmentdescriptor bevat een beschrijving van het betrokken segment.

Het code segment is het segment waarin de instructies opgeslaan zijn.
Het data segment is het segment waarin de statische data opgeslaan is.
Het stack segment is het segment waarin de stapel gealloceerd is.

FLOATING-POINT / MMX REGISTERS

De IA32-architectuur heeft 8 floating-point registers, deze zijn 80 bits breed en dus veel preciezer dan de vereisten van de eerder besproken ANSI/IEEE 754 standaard. Deze registers worden zowel voor single als double precision berekeningen gebruikt. De namen zijn ST0– ST7.

Gegevens in deze registers worden opgeslagen volgens een $\langle 1|15|64\rangle$ floating-point codering. De registers zijn georganiseerd in een stapelstructuur waarbij ST0 naar de top verwijst en met oplopende index verder van de top zitten.

Op het moment dat Intel zijn multimediainstructies (MMX) toevoegde aan de processor, wou men in verband met kostprijs en die size geen extra registers toevoegen. Daarom gebruikte men hiervoor de 8 bestaande floating-point registers. Het gevolg hiervan is dat een programma niet tegelijk floating-point en MMX

berekeningen uit kan voeren. De MMX registers vullen de laagste 64 bits van de ST registers en heten MMX0 – MMX7.

XMM REGISTERS

Na de MMX uitbreiding voegde Intel nog SSE en SSE2 instructies toe. Hiervoor werden 8 extra 128 bits registers toegevoegd. Deze kunnen 4 single precision floats of 2 double precision floats bevatten. De namen zijn XMM0 – XMM7.

DE ALPHA PROCESSOR

De Alpha processor steunt op een heel andere architectuur dan de Pentium. Er zijn hier 32 64 bit registers voor gehele getallen en 32 64 bit registers voor floats. De registers r31 en f31 bevatten enkel 0, men kan er dus niet naar schrijven en krijgt bij het lezen enkel 0 terug. Een extra PC register houdt de instructiewijzer bij.

Deze registerconfiguratie is een voorbeeld van hoe het hoort in een moderne architectuur en wordt niet zoals de Pentium getekend door de verschillende computer(r)evoluties.

DE ITANIUM PROCESSOR

De Itanium architectuur heeft zeer veel registers:

128 64 bit registers voor gehele getallen
128 64 bit registers voor floats
64 1 bit registers voor predikaten: hierin kan men informatie opslaan voor controletransferfuncties
8 64 bit registers voor adressen: hierin kan men adressen opslaan waarnaar gesprongen kan worden

PERMANENT GEHEUGEN

Dit zijn geheugentypes die hun inhoud niet verliezen als de stroom afgaat.

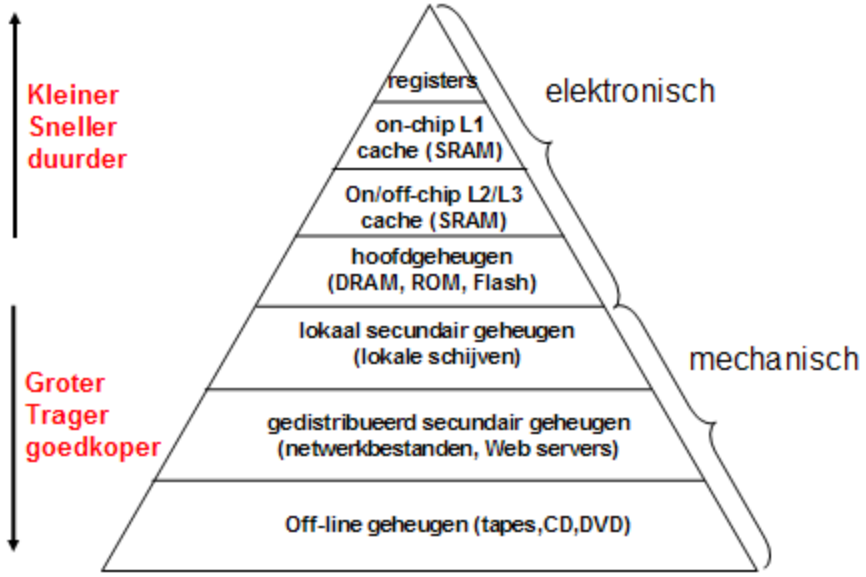
Naam	Uitleg
ROM: Read Only Memory	Kan geïmplementeerd worden als combinatorisch circuit dat voor elke ingang de gewenste waarde geeft.
PROM: Programmable ROM	Analoog aan een ROM maar kan eenmaal beschreven worden.
EPROM: Erasable PROM	Kan herschreven worden na blootstelling aan UV-licht.
EEPROM: Electrically Erasable PROM	Kan herschreven worden na elektronisch wissen (per byte).
Flash	EEPROM's die per blok in plaats van per byte gewist en geschreven kunnen worden. Hierdoor hebben ze een veel grotere bandbreedte.

GEHEUGENHIËRARCHIE

Het geheugen is een van de belangrijkste elementen in de von Neumann computer. Zowel instructies als data worden er in opgeslaan. Hoewel de processorsnelheid en geheugengrootte met rasse schreden vooruitgaan, stijgt de snelheid van het geheugen maar met 10% per jaar, een kloof die dus steeds breder wordt.

We bespreken een aantal manier om met behulp van diverse geheugentechnologiën het probleem van het trage geheugen op te lossen.

De doelstelling is om een groot, parallel toegankelijk, goedkoop en snel geheugen te maken opgebouwd uit dure, kleine maar snelle geheugens en grote, trage maar goedkope geheugens.



LOCALITEIT

Localiteit behandelt het gegeven dat programma's het grootste deel van hun tijd in een klein deel van de code spenderen, dit geldt ook voor de data.

Er zijn twee lokaliteitswetten:

Temporele lokaliteit: in een korte tijd komen dezelfde adreslocaties vaak terug
Spatiale lokaliteit: gebruikte geheugenlocaties liggen vaak dicht bij elkaar

De working set van een programma is de verzameling van geheugelocaties die het tijdens een bepaalde periode gebruikt. Door de temporele lokaliteit is deze werkverzameling vrij klein.

CACHES

Caches zijn noodzakelijk omdat de snelheid van het geheugen trager groeit dan de snelheid van processoren. Ze maken de geheugenbarrière minder hoog door tussengeheugens tussen processor en hoofdgeheugen te plaatsen. Ideaal is als de working set in de cache past, het programma voert dan uit aan de snelheid van de cache.

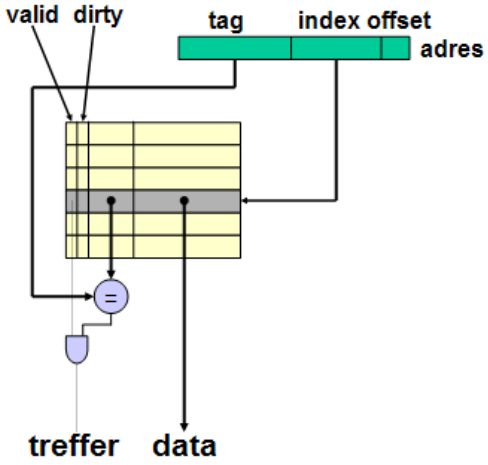
De cache wordt vaak geïntegreerd op de processorchip, hierdoor kunnen ze snel met elkaar communiceren. Verder probeert men de cache zo efficiënt mogelijk te gebruiken door gegevens uit het hoofdgeheugen in blokken op te vragen in plaats van per byte. Wanneer de CVE gegevens uit het geheugen nodig heeft, geeft hij

de vraag door aan de cache. Enkel als de waarde niet aanwezig is, wordt het hoofdgeheugen aangesproken en de gegevens in de cache geplaatst.

CACHESTRATEGIE

DIRECT-MAPPED CACHES

Bij een direct-mapped cache heft elk blok geheugen een vaste plaats in de cache, bepaald door de index. Bovendien houdt de cache bij of de entry in de cache wel een geldige waarde bevat (valid bit) en of ze aangepast is ten opzichte van het geheugen (dirty bit).



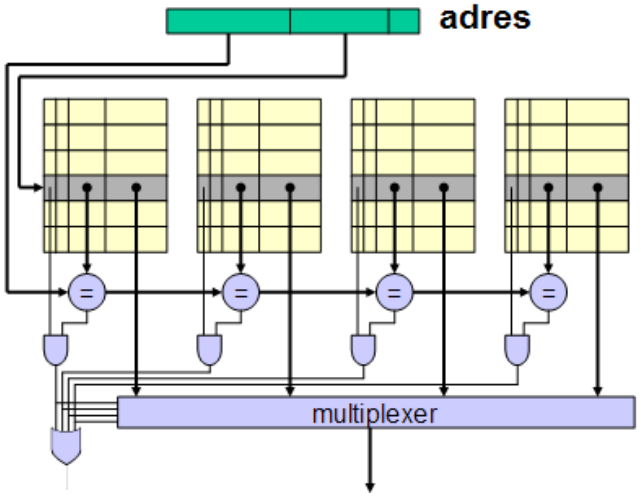
De cache kan ook meerdere sets bevatten, in dat geval heeft elke entry meerdere tags en bevat hij cellen uit verschillende geheugenblokken.

Aangezien de cache veel kleiner is dan het hoofdgeheugen kunnen zeer veel geheugenblokken op dezelfde plaats terechtkomen, daarom splitst men het geheugenadres op in een linkderdeel, de tag, een middendeel, de index en een rechterdeel de offset. De cache bepaalt of het een bepaald geheugenadres bevat door in zijn tabel de index op te zoeken, te kijken of de tags overeenkomen en te controleren of de valid bit wel op 1 staat. De offset is de positie van de geheugencel in de cache-entry.

Indien een applicatie een spaciale lokaliteit bezit die toevallig overeenstemt met dezelfde cache-entry, kan dit zeer veel cacheconflicten veroorzaken.

SET-ASSOCIATIEVE CACHES

Om te vermijden dat gegevens die precies even ver uit elkaar liggen als de cachegrootte voor conflicten zorgen, kan men ervoor kiezen om ervoor te zorgen dat elke plaast in de cache verschillende blokken kan opslaan. Dit is een N-wegs set-associatieve cache.



Dit voorbeeld bestaat uit 4 direct-mapped caches die parallel bevraagd orden. Indien in één van de wegen de gevraagde tag voorkomt, dan wordt de bijhorende data via de multiplexer naar buiten gebracht. Men kan nu vier blokken op dezelfde plaats in de cache opslaan. Dit verkleint de kans op conflicten aanzienlijk.

De set-associatieve cache levert een hogere hit-rate, maar een lagere toegangstijd door de extra multiplexer.

VOLLEDIG ASSOCIATIEVE CACHES

Bij een volledig associatieve cache kan elk geheugenblok terecht op elke plaats in de cache. Hierbij moet men dus het volledige adres van het geheugenblok bijhouden in de cache.

VERVANGINGSSTRATEGIE

Een vervangingsstrategie is noodzakelijk om als de cache vol is te bepalen welke gegevens weggehaald worden om plaats te maken voor de nieuwe.

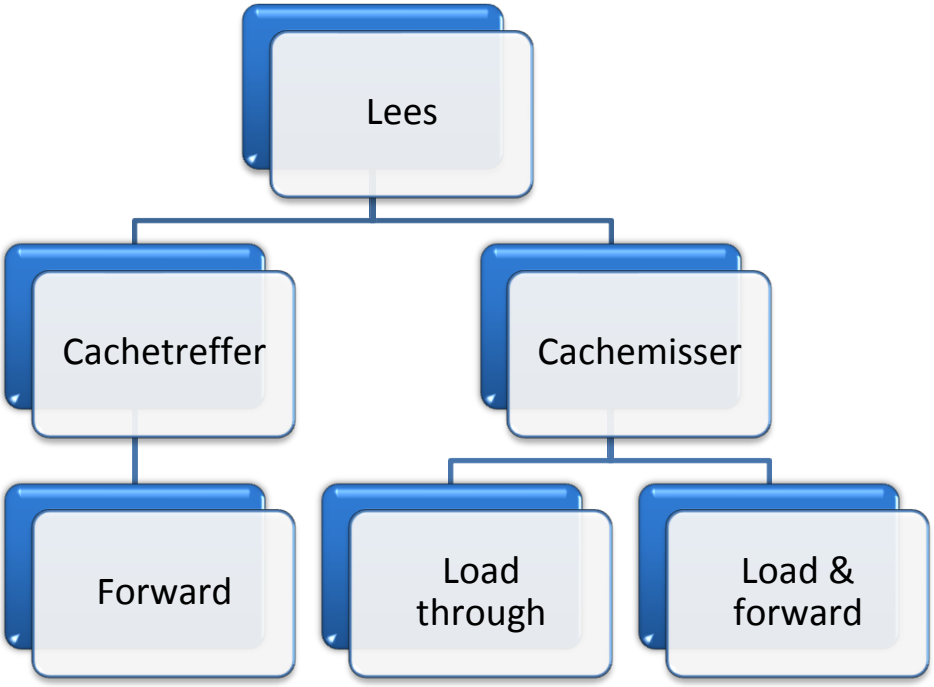
Er zijn enkele mogelijkheden:

LRU of least recently used: haal het blok weg dat het langs niet gebruikt is
FIFO of first in first out: haal het blok dat het langst in de cache zit weg
Random: haal een willekeurig blok weg, de prestatie hiervan is vaak niet veel slecter dan de andere strategiën
Opt: haalt het blok weg dat het langst niet gebruikt zal worden, praktisch niet realiseerbaar maar wel interessant om mee te vergelijken

Uit onderzoek blijkt dat de verschillen tussen deze strategiën niet groot zijn en verkleinen naarmate de caches groter worden.

LEESSTRATEGIE

Indien de data niet in de cache gevonden wordt zijn er een aantal mogelijke strategiën:

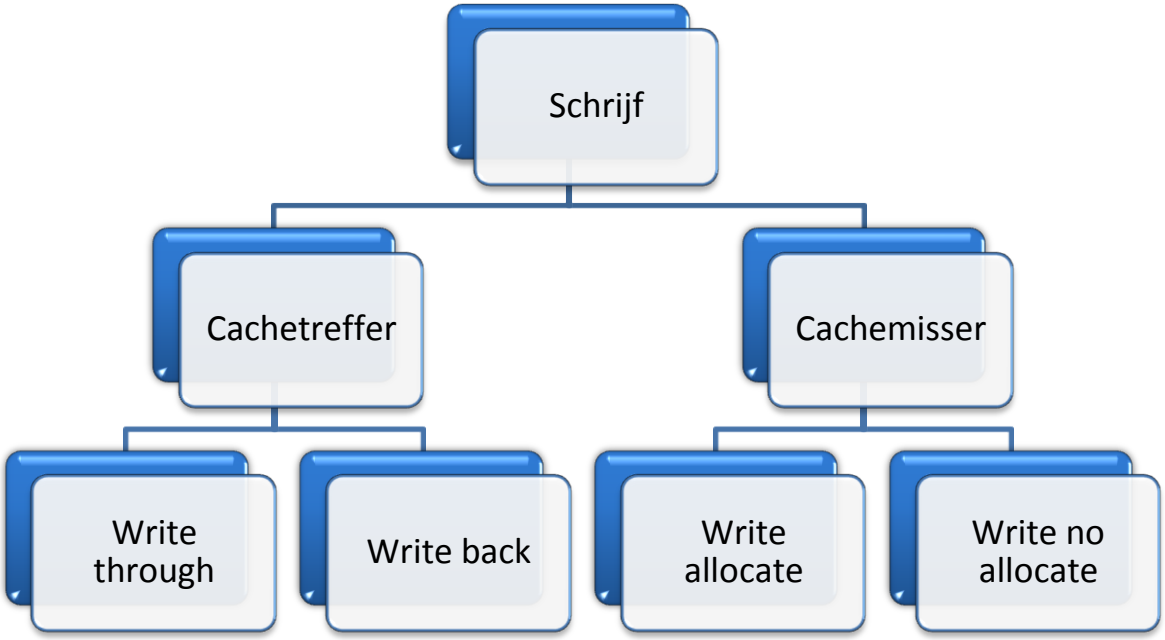


Load Through: zo snel mogelijk de data inladen en doorsturen naar de CVE, dan pas verdergaan met de rest van het geheugenblok te cachen.

Load & Forward: eerst het geheugenblok laden in de cache en dan de data doorsturen naar de CVE.

SCHRIJFSTRATEGIE

Bij het schrijven naar het geheugen zijn ere en aantal mogelijke strategiën:



Write through: voer de schrijfoperatie parallel in de cache en in het geheugen door, geheugen en cache blijven consistent.

Write back: voer de schrijfoperatie enkel uit in de cache, later bij het verwijderen uit de cache moet men de operatie doorvoeren in het geheugen.

Write allocate: alloceer het blok in de cache en pas dan de operatie toe.

Write no allocate: schrijf rechtstreeks naar het hoofdgeheugen zonder te alloceren in de cache.

De beste keuze hangt sterk af van de processorarchitectuur en de programma's.

TOEGANGSTIJD

De gemiddelde toegangstijd tot een cache is:

$$t_{AMA} = t_{hit} + (rate_{miss} \cdot penalty_{miss})$$

$$t_{AMA} = rate_{hit} \cdot t_{hit} + rate_{miss} \cdot t_{miss}$$

De volgende cacheparameters beïnvloeden de gemiddelde toegangstijd:

Parameter	Hit Time	Miss Rate	Miss Penalty
Grootte stijgt	Stijgt	Daalt	-
Associativiteit stijgt	Stijgt	Daalt	Stijgt
Blokgrootte stijgt	-	Onbepaald	Stijgt
Splitsing stijgt	Daalt	Onbepaald	-
Niveau's stijgt	Daalt	Stijgt	Daalt

Er bestaan missers in drie categoriën:

Koude missers: er is nog nooit naar het blok gerefereerd
Capaciteitsmissers: het blok werd verwijderd omdat de cache vol was (komt voor bij volledig associatieve caches)
Conflictmissers: het blok werd verdrongen door een nieuw blok (komt voor bij direct-mapped en set-associatieve caches)

CACHEGROOTTE

Een grotere cache doet de miss rate sterk dalen, uiteraard is ook hier een limiet aan de nuttige grootte. Vanaf een bepaald ogenblik is de vertraging van de cache het voordeel van de lagere miss rate teniet doen.

Een grotere cachegrootte doet ook de hit time stijgen.

ASSOCIATIVITEIT

De 2:1 regel zegt dat een direct-mapped cache een even hoge miss rate heeft als een 2-wegs set-associatieve cache van halve grootte.

Bij volledige associativiteit kan de miss penalty toenemen aangezien de verwijderingsstrategie langer zal duren.

Een grotere associativiteit doet de hit time stijgen.

BLOKGROOTTE

Men kan het effect van spatiale lokaliteit uitbuiten door meerdere woorden per cache-entry op te slaan, hierdoor kunnen we ook relatief meer data opslaan in de cache omdat we in verhouding minder tagbits nodig hebben.

Een stijgende blokgrotte zal eerst de miss rate doen dalen, maar aangezien hierdoor het aantal sets daalt zal bij stijgende blokgrootte de miss rate terug stijgen.

SPLITSING

Een gesplitste cache gebruikt aparte cachetabellen voor instructies en data. Doordat deze caches kleiner zijn, is de hit time lager. De miss rate is onbepaald, meestal lager voor de instructiecache en hoger voor de datacache.

Aangezien de meeste operaties naar de instructiecache gaan, zal een gesplitste cache meestal sneller zijn den een geünificeerde.

NIVEAU'S

In moderne processors zijn er vaak meerdere niveau's aan caches. Dit doet de miss rate stijgen, maar aangezien het om kleinere caches gaat daalt de hit time. De miss penalty stijgt ook aangezien een geheugenoperatie nu door alle caches moet passeren.

ADRESEXPRESSIES

Een adresexpressie is eenwiskundige uitdrukking die duidelijk maakt welke berekeningen er precies gebeuren om te komen tot het effectieve adres.

Er zijn drie basissoorten adresexpressies:

> Absoluut adres: *effectief adres = adres*
> Verschoven adres: *effectief adres = adres + offset*
> Indirect adres: *effectief adres = mem[adres]*

We bespreken nu een aantal adresseermodes. Een processor ondersteund meestal maar een aantal van de volgende modes. De voorbeelden gebruiken instructies die beschreven worden bij geheugentransfer instructies.

ADRESSEERMODES

ABSOLUTE ADRESSERING

De meest eenvoudige adresseermode. Geen afzonderlijke berekeningen zijn vereist. Deze adresseermode is goed bruikbaar bij globale variabelen en systeemvariabelen of scalairen. Een belangrijk nadeel is dat het volledige adres in de instructie moet opgenomen worden, op sommige architecturen is de instructiegrootte niet groot genoeg om een volledig adres te bevatten.

Bijvoorbeeld: *mov eax, [20]*

INDEXERING

Vooral gebruikt bij adressering van arrays. De vershuiving ten opzichte van een basisadres wordt bepaald met behulp van de index. Het uiteindelijke adres is de som van het basisadres en het product van het aantal elementen en de lengte van een element. Opnieuw moet een volledig basisadres opgeslaan worden.

Bijvoorbeeld: *mov [19290h + edi*4], edi*

BASISADRESSERING

Deze adresseermode wordt gebruikt bij velden in structuren. Het beginadres van de structuur ligt opgeslagen in een register en een kleine verschuiving wordt hierbij opgeteld. Aangezien de verschuiving meestal klein is, heeft deze adresseermode niet langer het nadeel van te kleine instructiegroottes.

Bijvoorbeeld: *mov eax, [ebx + 20]*

BASIS EN INDEXERING

Dit is een combinatie van de basisadressering en de indexering. Men slaat zowel het basisadres als de index op in een register. Opnieuw is deze adresseermode klein genoeg voor architecturen met een kleine instructiegrootte.

Bijvoorbeeld: *mov eax, [ebx + 20 + edi*4]*

GEHEUGENINDIRECT

De geheugenindirecte adresseermode haalt het effectieve adres uit een geheugenlocatie waarvan het adres aan de hand van een reeds bestaande adresseermode kan gespecifieerd worden. Dit wordt niet ondersteund dooe de IA32 architectuur, maar wel door bijvoorbeeld de Motorola 680x0-architectuur.

Bijvoorbeeld: *mov dword ptr [[edi + 4] + 4], 0*

REGISTERINDIRECT

Een speciale vorm van basisadressering met als basis 0.

Bijvoorbeeld: *mov dword ptr [edi], 99*

POSTINCREMENT, PREDECREMENT

Deze adresseermodes laten toe om de waarde van het register te incrementeren of te decrementeren bij een registerindirectie.

Bijvoorbeeld (Motorola 680x0): *move.l (a1)+, d3*

Deze adresseermode wordt niet ondersteund door IA32, maar wel door de Motorola 680x0. Gelijkwaardige instructies voor de Intel processor vindt u terug bij geheugentransfer instructies.

LETTERLIJK

Deze adresseermode dient speciaal om constanten te specifiëren. In plaats van deze via het geheugen te moeten laden, is het veel efficiënter om ze rechtstreeks met de instructie mee te geven.

Bijvoorbeeld: *mov eax, 0aah*

SEGMENTERING

Er zijn 6 segmentregisters. Deze verwijzen allen naar een segment descriptor die de rechten op het segment, de grootte en de basis bepaalt. Over het algemeen wijzen deze segment descriptors allemaal naar de hele adresruimte. Als men ze gebruikt moet men erop letten dat bepaalde instructies naar bepaalde segmenten gaan. Zo gaat bijvoorbeeld een jump altijd naar het code segment.

De segment descriptors verdelen het fysieke geheugen in meerdere blokken. De 6 segmentregiseters kunnen dan naar een van deze segment descriptors wijzen. De descriptors staan samen gedefiniëerd in een tabel waarnaar verwezen wordt door het gdtr register (global descriptor table register). De segmentregisters verwijzen naar een offset ten opzichte van dit basisadres.

Een segment descriptor ziet er als volgt uit:

Basis<31:24>	Extra bits	Limiet<19:16>	Basis<23:16>	Basis<15:0>	Limiet<15:0>

Hoewel de limiet maar 20 bytes is, hoeft dit niet noodzakelijk een veelvoud van bytes te zijn. Het kan ook een veelvoud van een aantal bytes zijn. Hierdoor kan men toch de hele adresruimte adresseren.

BITNUMMERING

Het is soms noodzakelijk om aparte bits in een bitpatroon te kunnen adresseren. We kunnen dit op twee manieren realiseren:

Opwaartse nummering: de meest significante bit heeft bitadres 0, het voordeel is dat de nummering consistent is met de bytenummering
Neerwaarste nummering: de minst significante bit heeft bitadres 0, het voordeel is dat de nummering overeenstemt met het gewicht van de bit

BYTENUMMERING

Bij het bewaren van woorden met meerdere bytes, kunnen de bytes in 2 verschillende volgordes opgeslaan worden:

Big endian: de meest beduidende byte komt op het laagste adres, gebruikt door Motorola en IBM
Little endian: de minst beduidende byte komt op het laagste adres, gebruikt door Intel en Digital

De Pentium processor slaat gegevens op in het little endian formaat. In de registers werkt men echter met big endian.

ALIGNATIE

Sommige geheugenarchitecturen eisen dat gegevens gealigneerd opgeslaan worden. Dit houdt in dat alle multi-byte bitpatronen op een adres moeten beginnen dat een veelvoud is van hun eigen grootte. Multi-byte patronen zijn byte-eenheden die in 1 keer gelezen of geschreven worden.

De reden van deze eis is dat gegevens uit het geheugen meestal in grote blokken ingelezen worden. Als een bitpatroon gealigneerd is, is de kans dat het gespreid is over meerdere geheugenwoorden veel lager. Alignatie kan dus de performantie verhogen. Nadeel is uiteraard dat het geheugen niet volledig efficënt gebruikt kan worden omdat er alignatiebytes wegvallen.

RANDAPPARATUUR

Hier bekijken we randapparaten in de computer en de manier waarop we er mee kunnen communiceren.

BUSSEN

Bij het implementeren van een interface tussen de processor en randapparaten, wensen we die zo uniform mogelijk te maken. Hierbij stuitten we echter op enkele problemen die dit lastig maken:

De snelheden die nodig zijn voor de communicatie kunnen sterk verschillen, van een aantal toetsaanslagen tot streaming video
Randapparaten werken asynchroon met de CVE, hierdoor is het noodzakelijk om controlesignalen voor synchronisatie te hebben
De error rate in de communicatie ligt veel hoger dan bij communicatie met het geheugen
Randapparaten kunnen om een veelheid aan redenen falen

Indien men zomaar alle componenten in een computer rechtstreeks zou verbinden, zou ere en onontwarbaar kluwen aan verbindingen nodig zijn. Om dit problem op te lossen is de bus geïntroduceert.

Een bus is een verbinding die gedeeld wordt door meerdere apparaten. Ze bestaat uit verschillende lijnen: adreslijnen, controlelijnen, datalijnen en voedingslijnen. Alle apparaten luisteren simultaan naar de bus, maar slechts een kan er op een bepaald tijdstip naar schrijven. Bussen zijn een goedkope en effecteive methode om een aantal componenten te verbinden.

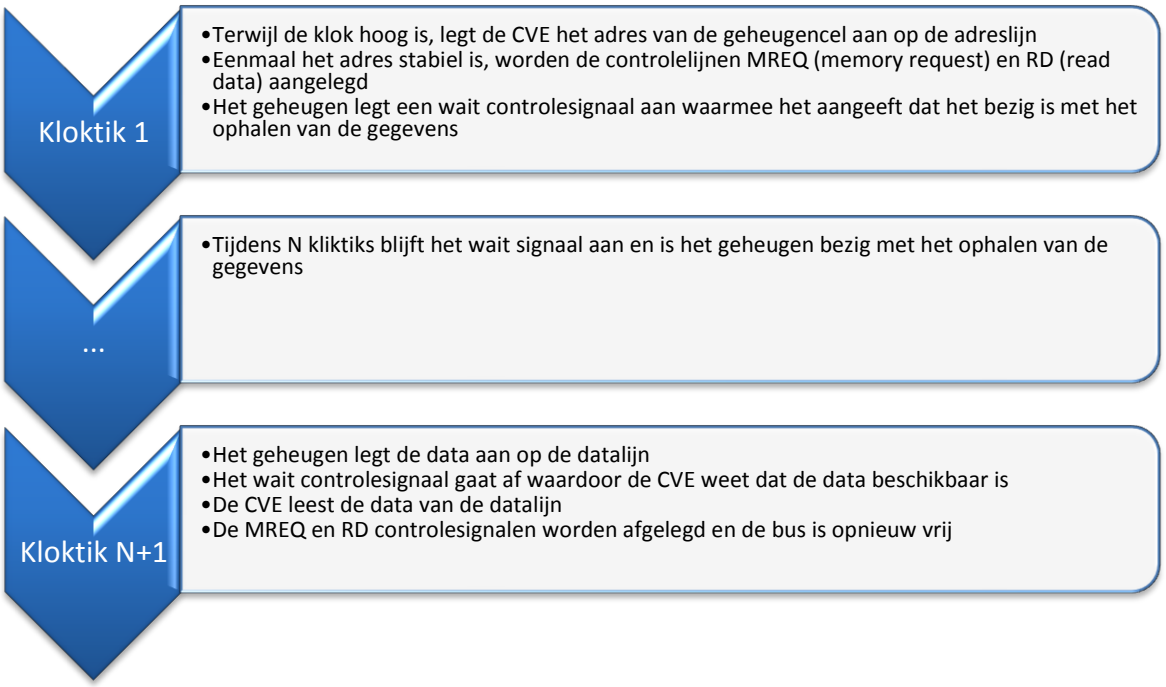
Nadelen aan de bus zijn dat om technische redenen maar een beperkt aantal apparaten aangesloten kan worden en de selectie van het apparaat dat kan schrijven op de bus.

Qua signaalsturing zijn er twee subtypes bussen: synchrone en asynchrone.

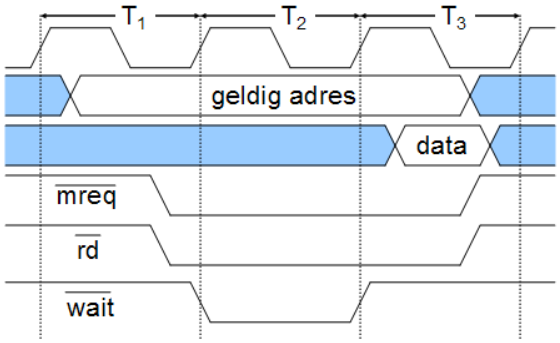
SYNCHRONE BUS

Bij dit type bus word teen kloksignaal verspreid door een van de componenten. Alle operaties worden gesynchroniseerd met dit signaal.

Als voorbeeld van een synchrone busoperatie nemen we het lezen van een geheugencel door de CVE:



Een timingdiagram van de operatie:

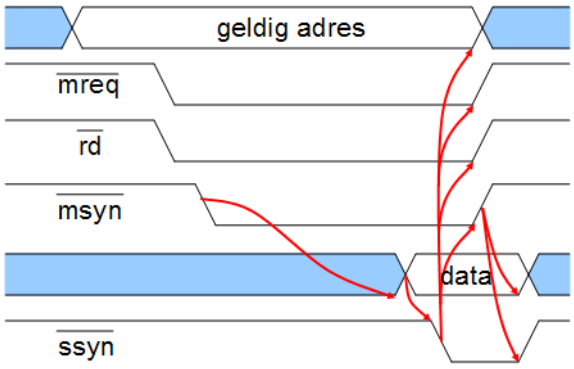


ASYNCHRONE BUS

Bij een asynchrone bus werkt elk component op zijn eigen tempo. Het systeem als een geheel is echter complexer.

Als voorbeeld nemen we een asynchrone leesoperatie in het geheugen:

De CVE legt een geldig adres aan op de adreslijn en brengt de controlesignalen MREQ (memory request) en RD (read data) hoog
Vervolgens brengt de CVE het MSYN (master synchronisation) hoog
Eenmaal MSYNC hoog is, weet het geheugen dat de opereatie uitgevoerd kan worden
Na het uitvoeren van de leesoperatie brengt het geheugen de data op de datalijnen aan en brengt het SSYN (slave synchronisation) hoog
Hierop weet de CVE dat de data beschikbaar is en leest hij de data van de databus, het adressignaaal en de MREQ, RD en MSYN controlesignalen worden opnieuw laag gebracht



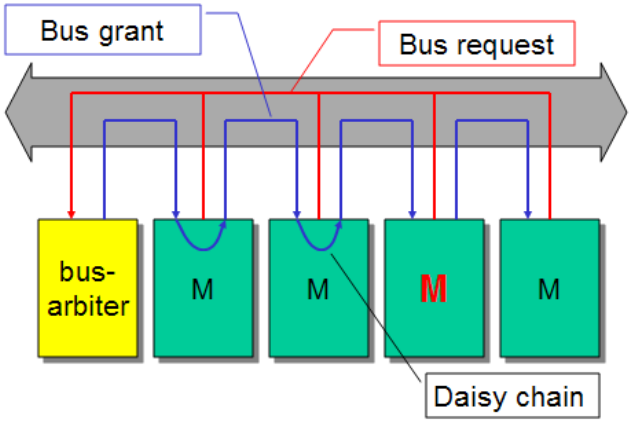
Sommige apparaten nemen de rol van bus master op, de andere zijn bus slaves. Enkel een bus master kan de bus gebruiken om te zenden of om te ontvangen, een slave kan enkel reageren op signalen van een master. Indien er meerdere masters aanwezig zijn op de bus, is er een arbitragesysteem nodig dat bepaalt welke master op welk tijdstip mag schrijven. Hier zijn verschillende methodes voor.

DAISY CHAIN ARBITER

Bij daisy chaining delen alle componenten een bus request draad die naar de bus arbiter gaat. Bovendien loopt er een bus grant draad door alle componenten heen.

Wanneer een apparaat bus master wil worden, activeert het de bus request draad. De bus arbiter detecteert dit en als de bus vrij is, activeert hij het bus grant signaal. Dit signaal gaat van component tot component tot aan het eerste apparaat dat bus master wil worden, hier wordt het signaal niet verder doorgegeven en dat apparaat is dan bus master.

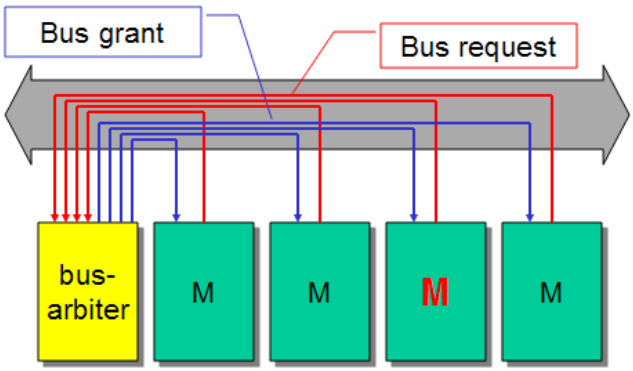
Het is dus zo dat hoe dicheter men bij de arbiter zit, hoe hoger de prioriteit is om bus master te worden.



CENTRALE ARBITER

Een volledig gecentraliseerde bus arbiter heeft aparte request en grant lijnen lopen naar elk component. De arbiter kan nu volledig autonoom beslissen welk apparaat bus master wordt.

Hoewel deze oplossing de meeste functionaliteit biedt, is ze ook de duurste en minst schaalbare door de vele extra verbindignen die noodzakelijk zijn.



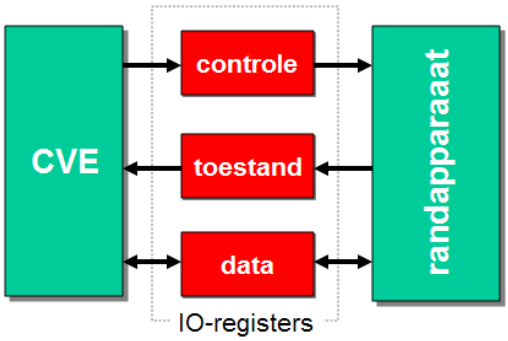
DECENTRALE ARBITER

Bij een gedecentraliseerde method proberen de mogelijke masters onder elkaar te beslissen wie master wordt. Een extreme vorm hiervan is het ethernet protocol waarbij een apparaat gewoon de kabel begint te gebruiken en pakketjes opnieuw verstuurd wanneer er een conflict optreedt.

COMMUNICATIE

Bij de communicatie tussen twee autonome componenten (de CVE en het randapparaat), moeten we twee taken uitvoeren: synchronisatie en de effectieve overdracht.

De overdracht van gegevens tussen de CVE en randapparaten gebeurt via een klein aantal I/O registers, een gemeenschappelijk geheugen.



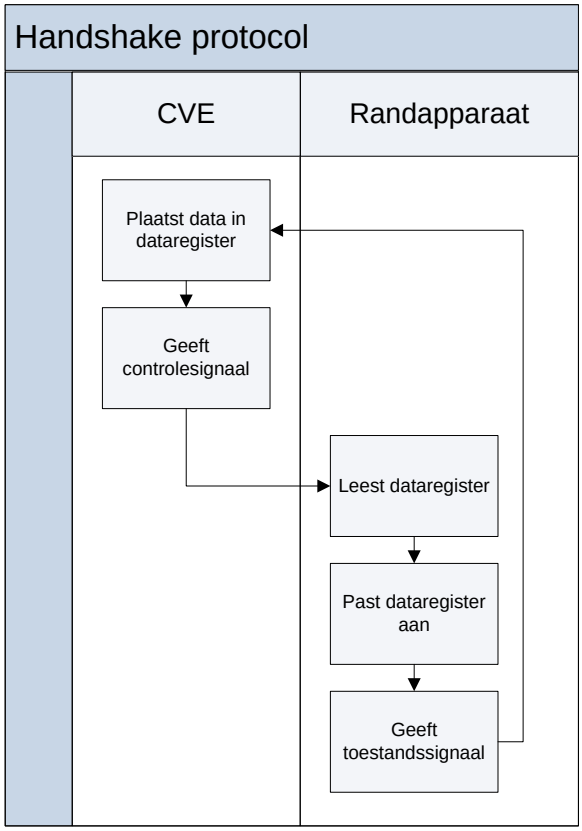
We zien dat er drie soorten communicatie zijn: controlesignalen van de CVE, toestandsignalen van het randapparaat en de gedeelde data.

Om de I/O registers aan te spreken moeten ze een adres hebben. Dit adres kan ofwel een geheugenadres zijn, ofwel een adres in een aparte I/O adresruimte. Het voordeel van geheugenadressen is dat men met de reeds aanwezige processorinstructies voor geheugenbewerkingen kan communiceren met randapparaten. Een aparte I/O adresruimte heeft als voordeel dat het gemakkelijker te implementeren is aangezien de communicatie via heel aparte paden moet lopen.

SYNCHRONISATIE

De synchronisatie maakt duidelijk welke I/O registers beschikbaar zijn voor wie en garandeert de stabiliteit van de overdracht. Zo bepaalt men eenduidig op welk moment wie naar een bepaald register mag schrijven.

De voorschriften die CVE en randapparaat volgen bij de communicatie heet het communicatieprotocol. Een voorbeeld hiervan is het handshake protocol voor het delen van een register:



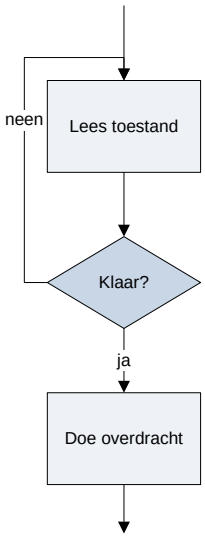
Dit wordt onder andere gebruikt voor printercommunicatie.

GEPROGRAMMEERDE OVERDRACHT

ACTIEVE SYNCHRONISATIE

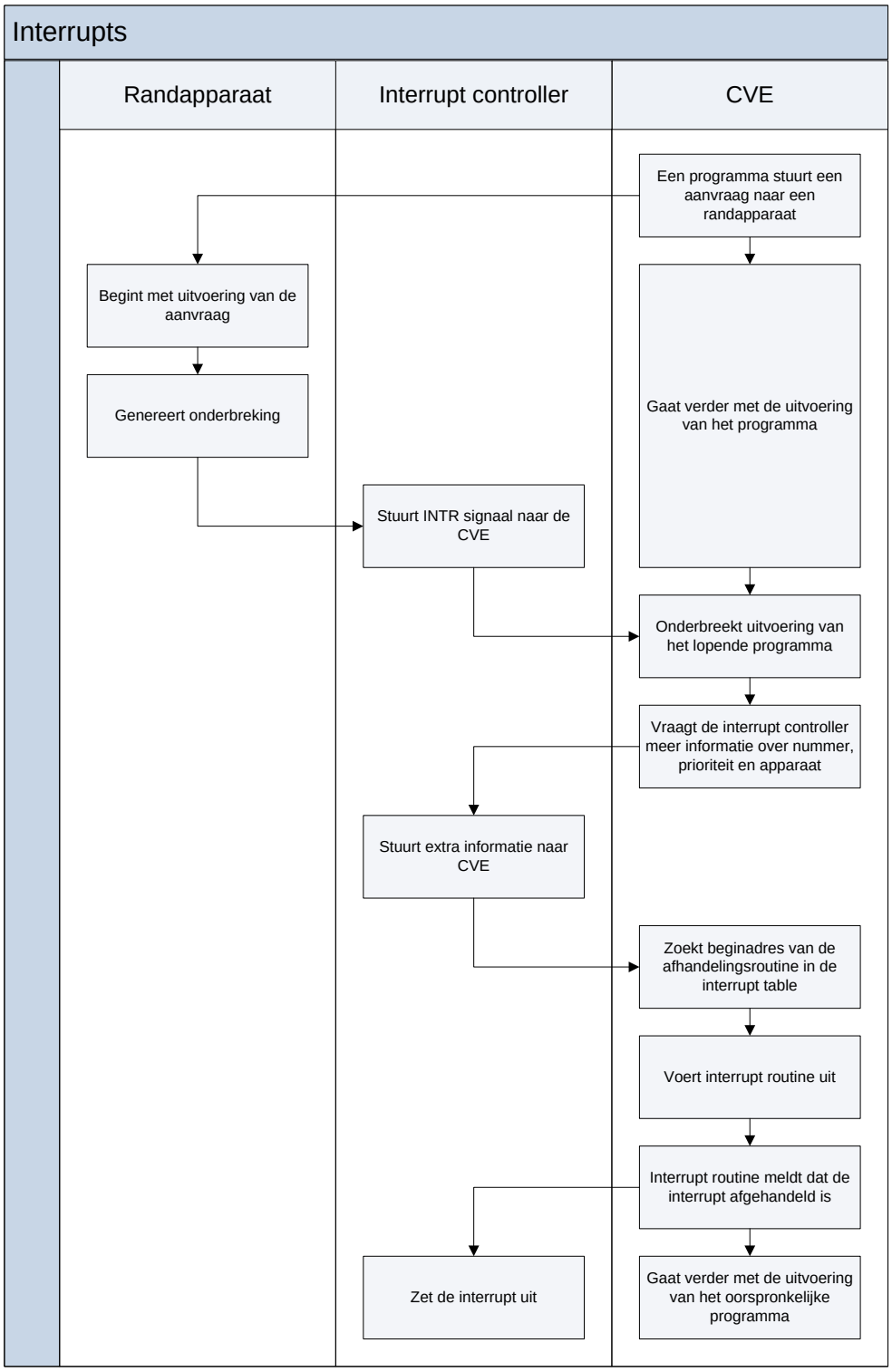
Bij actieve synchronisatie of polling leest de CVE continu de toestand van het randapparatuur tot het aangeeft dat de CVE actie mag ondernemen. Met deze methode kan de CVE ondertussen geen ander werk verrichten, dit is dus helemaal niet interessant bij trage randapparaten.

De methode verloopt als volgt:



ONDERBREKINGEN

Om de problemen van actieve synchronisatie te overkomen werd het systeem van onderbrekingen ontworpen. Hierbij hangt op de systeembus een interrupt controller die signalen van de randapparaten krijgt wanneer ze aandacht van de CVE vereisen. Dit verloopt als volgt:



Indien meerdere interrupts tegelijk binnenkomen, sorteert de interrupt controller ze op prioriteit en stuurt ze in die volgorde naar de CVE. Het is bovendien mogelijk om meerdere interrupt controllers in cascade achter elkaar te schakelen.

Eenmaal de CVE het nummer van de onderbreking kent, zoekt hij de routine op die deze interrupt af kan handelen. De beginadressen voor deze routines worden in het geheugen opgeslaan in een interrupt descriptor table. Het begin (32 bits) en de lengte (16 bits) van deze tabel is opgeslaan in het idtr register. Elke interrupt descriptor bestaat uit een 32 bit adres, gecombineerd met een nieuwe waarde voor het code segment en enkele extra bits met informatie over het verdere verloop van de interrupt.

De interrupt descriptor ziet er als volgt uit:

eip<31:16>	extra bits	cs<15:0>	eip<15:0>

Indien nodig kan met het onderbreken van een programma door interrupts tijdelijk afzetten.

Na het opzoeken van de interrupt descriptor pusht de processor eerst eip, dan cs en vervolgens eip op de stack zodat hij die later kan herstellen eenmaal de interrupt afgelopen is.

Dit zijn gereserveerde systeeminterrupts voor de IA32 architectuur:

Nummer	Omschrijving	Uitleg
0	Divide by 0	Wordt opgeroepen bij een nuldeling.
1	Single step	Gebruikt door debuggers, indien de single step toestandsbit in eflags aan staat, dan genereert de CVE na elke instructie een single step interrupt. Zo kan het programma instructie per instructie uitgevoerd worden.
2	Non maskable interrupt (parity error)	Bij ernstige hardware fouten. Kan niet uitgeschakeld worden.
3	Breakpoint	Gebruikt door debuggers om breakpoints in te voegen.
4	Overflow	Opgeroepen door de *into* instructie die overflows test.
5	Bound range exceeded	
6	Invalid opcode	
7	Device not available (no math co-cpu)	
8	Double fault	
9	Co-cpu segment overrun	
10	Invalid TSS	
11	Segment not present	
12	Stack-segment fault	
13	General protection	
14	Page fault	
31	Reserved	

Dit zijn de interrupts in de onderbrekingsregelaar, ze kunnen afgebeeld worden op onderbrekingen in de processor:

IRQ	Omschrijving
0	Hardware timer
1	Keypress

2	Tweede interrupt controller
3	COM1
4	COM2
5	Hard disk
6	Floppy disk
7	LPT1
8	Real-time clock
9	Redirect cascade
10	Reserved
11	Reserved
12	Mouse
13	Coprocessor exception
14	Fixed disk
15	Reserved

Men kan onderbrekingen onderverdelen in de volgende categoriën:

DIRECTE GEHEUGENTOEGANG

Directe geheugentoegang of Direct Memory Access (DMA) stelt het randapparaat in staat om data rechtstreeks in en uit het geheugen te transfereren zonder dat dit langs de CVE moet passeren. DMA operaties worden gecontroleerd door een DMA controller.

Bij een DMA overdracht stuurt de CVE naar de DMA controller het geheugenadres, de lengte en het doelapparaat. De DMA controller begint dan autonoom de gegevens te transfereren naar het apparaat zonder tussenkomst van de CVE en houdt controleert zelf de toestand van het apparaat.

Aangezien de DMA controller dus ook geheugentoegang nodig heeft, is dit een bus master. Om bij grote transfers de geheugentoegang niet af te nemen van de CVE, neemt de DMA controller slechts per woord beslag op de bus. Er wordt dus per keer maar 1 geheugencycle weggenomen van de CVE, dit heet cycle stealing.

SECUNDAIR GEHEUGEN

MAGNETISCHE SCHIJVEN

Dit zijn apparaten met een zeer hoge opslagdichtheid en relatief snelle toegangstijd. De standaardschijven in een gewone computer zitten in een luchtdicht metalen omhulsel ter grootte van een walkman.

Magnetische schijven bestaan uit een opeenstapeling van platters op enkele millimeters van elkaar. Deze platters draaien rond een gezamelijke as en er is een kop die over de schijf heen kan bewegen.

Elke plaat heeft 2 oppervlakken uit aluminium of glas, bedekt met een magnetisch materiaal. Door het magnetiseren van kleine gebieden op het oppervlak, kan men bits opslaan.

Per oppervlak is er een kop, die allemaal samen bewegen. De kop zweeft over het oppervlak en leest en schrijft de platter. De arm waarop de koppen gemonteerd zijn, kan zeer precies bewegen over het oppervlak. Op elk moment kan op elk oppervlak een spoor gelezen worden. De verzameling leesbare sporen voor een armpositie heet de cilinder.

Een platteroppervlak is onderverdeeld in concentrische sporen die opnieuw opgedeeld zijn in sectoren van typisch 512 bytes. Tussen de sectoren en sporen zitten lege ruimtes om de positionering van de koppen te vereenvoudigen.

Performantie van magnetische schijven is afhankelijk van drie parameters:

Seek time: de tijd nodig om de kop op de juiste cilinder te verplaatsen
Rotational latency: de tijd die verstrijkt vooraleer de juiste sector onder de kop komt
Transfer time: de tijd nodig om de gegevens te transfereren

Om de performantie te verhogen hebben de meeste schijven een eigen cache geheugen (meestal tussen de 2 en 16 MB) om veelvoorkomende leesoperaties te bufferen. Ook schrijfoperaties kunnen gebuffered worden zodat de aanroepende code gewoon verder kan gaan terwijl de schijf haar buffer wegschrijft.

MASTER BOOT RECORD

Dit is een gereserveerde sector op de schijf dat de indeling van de schijf bijhoudt. Dit is meestal sector 0 van spoor 0 van oppervlak 0. De MBR bevat een lijst met de partities op de schijf en een bootloader die het besturingssysteem van een van de partities kan starten. Dit is het eerste gebied van de schijf dat ingeladen wordt bij het starten van de computer.

FLOPPY DISK

Diskettes werken op hetzelfde principe als harde schijven. Hier werkt men echter met flexibele plastieken schijven.

De capaciteit en snelheid van floppies zijn veel lager dan harde schijven. Ze roteren aan 300 RPM en hebben een capaciteit van 1,44 MB (80 sporen, 18 sectoren / spoor). De betrouwbaarheid laat ook te wensen over, onder meer omdat hier de kop wél de schijf raakt en er dus slijtage optreedt. Al bij al is dit een duur en te traag systeem.

MAGNETISCHE BANDEN

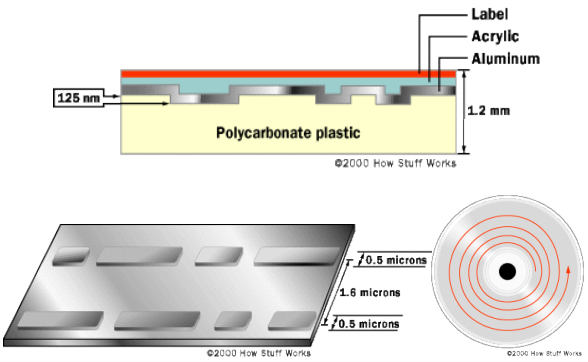
Magnetische tapes worden vooral gebruikt in backup-toepassingen. Ze hebben het heel lastig met willekeurige operaties en zijn vooral bedoeld voor sequentiële toegang. Het is echter een goedkope methode van archivering.

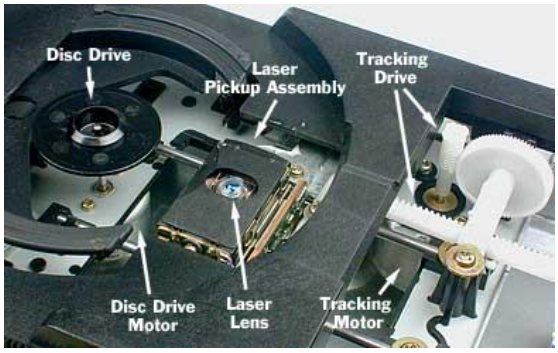
OPTISCHE SCHIJVEN

CD-ROM

Bestaat uit een plastieken onderlaag bedekt met aluminium. De data wordt opgeslagen als kleine vlakken en putjes in de onderlaag, zodat het licht anders reflecteert. Audio CD's hebben een capaciteit van 74 minuten of 783 MB, CD's met data gebruiken meer foutcorrectie en hebben dus maar een capaciteit van 650 MB.

CD's worden gedrukt met een master exemplaar. Dit is een goedkoop massaproces.





In tegenstelling tot harde schijven, is de data op een CD geschreven als een lange spiraal. Bovendien draaien harde schijven met een constant toerental of constant angular velocity, terwijl CD's een constante lineaire snelheid aanhouden (constant linear velocity). De putjes en vlakken in het oppervlak liggen op dezelfde afstanden en dus is er een constante datasnelheid.

Er bestaan ook schrijvers waarmee particulieren zelf CD's kunnen maken in kleine oplage. Hier bestaan schrijfbare en herschrijfbare varianten in. Het schrijven of herschrijven is altijd trager dan lezen.

DVD-ROM

De DVD of Digital Versatile Disk is bijna identiek aan een CD met kleinere bits. Het grote verschil is dat men nu ook meerdere lagen gegevens op hetzelfde oppervlak kan aanbrengen. Enkellaagse enkelzijdige DVD's hebben een capaciteit van 4,7 GB, dubbellaagse 8,5 GB en een dubbellaagse dubbelzijdige DVD kan 17 GB bevatten.

Op dit moment is de industrie bezig met het ontwerpen van een opvolger voor de DVD. Er zijn echter twee concurrerende standaarden: de HD-DVD en Blu-ray. Van de eerste zijn op dit moment (juni 2006) al spelers op de markt, van Blu-ray worden de komende maanden de eerste spelers verwacht.

INVOERAPPARATEN

TOETSENBORD

Een toetsenbord bestaat uit 101 (enhanced) of 104 (windows) toetsen. Deze kunnen afhankelijk van land tot land verschillende layouts hebben (qwerty, azerty, qwertz, dvorak, …). Deze layouts stammen uit de tijd van de schrijfmachines en plaatsen veelgebruikte letters ver uiteen om de gebruiker te vertragen en zo mechanische storingen te voorkomen. Dit is uiteraard een overbodige maatregel bij computers, daarom werd de dvorak-layout ontwikkeld om sneller te typen. Deze layout is echter weinig populair.

Intern bestaat een toestenbord uit een matrix waarbij elke toetsaanslag een schakelaar kortsluit. De x, y coördinaten worden naar de computer gezonden die deze omzet naar een karakter. Een toetsenbord kan intern ook een aantal codes bufferen voor het geval de computer niet zou kunnen volgen.

Bij het aanslaan van een toets wordt de schakelaar soms tweemaal kortgesloten, dit heet bounce. Het toetsenbord detecteert dit en geeft maar eenmaal het signaal door.

Indien men een toets blijft indrukken bepaalt typematics hoeveel maal het teken herhaald wordt per tijdseenheid.

MUIS

Oude muizen werken met een roterende bal. Wanneer de gebruiker de muis over een oppervlak beweegt zet de muis de rotatie van de bal om tot digitale signalen. Wanneer de bal beweegt vangen twee rollers deze beweging op en zetten die om naar een x, y vector. Deze omzetting wordt gedetecteerd door een infrarode lichtbron op een geperforeerde schijf. Het aantal pulsen dat door de schijf komt geeft de afgelegde afstand.

Moderne muizen gebruiken optische technieken om hun verplaatsing over een oppervlak te detecteren.

ANDERE

Andere invoerapparaten zijn:

- Trackball
- Touchpad
- Pointing stick
- Joystick
- LIchtpen
- Aanwijsscherm
- Digitiser en tablet

UITVOERAPPARATEN

PRINTER

Een parallelle poort heft drie I/O registers:

Controle	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Adres: 37Ah				IRQ enable	Selectin	Init	Autofeed	Strobe

Strobe: het controlesignaal wanneer het dataregister door de printer gelezen mag worden.
Autofeed: produceert een carriage return na elke linefeed.
Init: reset de printer.
Selectin: selecteert de printer.
IRQ enable: laat de printer een onderbreking genereren na het ontvangen van de databyte.

Toestand	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Adres: 379h	Ready	Ack	PE	Select	Error			

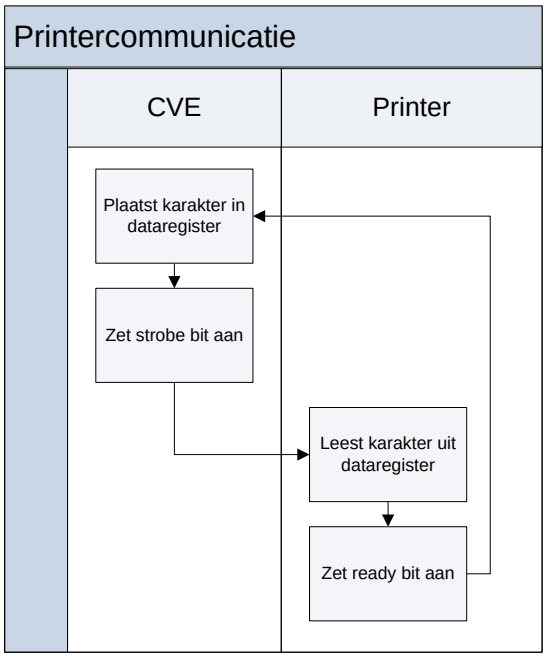
Error: rapporteert een fout.
Select: geeft aan of de printer geselecteerd staat.
PE: (Paper Empty) geeft aan of het papier op is.
Ack: een bevestiging van de ontvangst van het laatste teken.
Ready: geeft aan dat de printer klaar is om het volgende teken te ontvangen.

Data	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**

Adres: 378h								

Het controleregister wordt enkel door de CVE geschreven en stuurt commando's naar de printer. Het toestandsregister wordt enkel door de printer geschreven en geeft feedback over operaties aan de CVE.

Het protocol dat de printer gebruikt voor communicatie gaat als volgt:



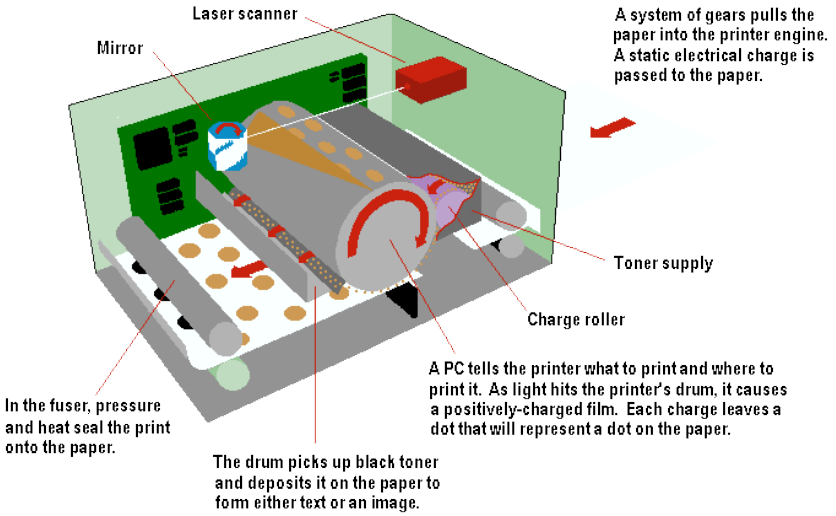
LASER PRINTERS

Een laser printer bestaat uit een elektrisch geladen drum. Via een roterende spiegel bestraalt de laser de drum en ontlaadt een aantal gebieden. Lijn voor lijn scant de laser zo de drum die ronddraait. De gebieden die geladen blijven op de drum nemen elektrostatisch geladen tonerpoeder op en zetten die later af op het papier.

Verderop in de printer wordt het papier opgewarmd en onder de druk en warmte van de rollers versmelt het poeder permanent met het papier.

Hierna wordt de toner gekuist en het proces herbegint voor een volgend blad.

Het tonerpoeder is een soort plastiek, het papier neemt dit niet op maar versmelt ermee.

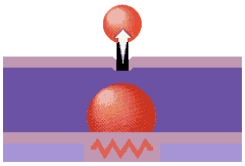


INK JET PRINTERS

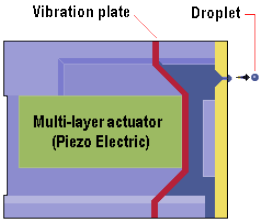
Dit zijn goedkopere printers dan laser printers, maar de inkt is duurder en ze zijn trager. Bovendien is de resolutie ook gemiddeld lager dan bij een laser printer. Voor thuisgebruikers is dit echter geen probleem en ze drukken ook kleur.

Een ink jet printer spuit kleine druppeltjes inkt op het blad. Om hoge resoluties te halen moeten de spuitgaatjes zeer klein en nauwkeurig zijn. Er zijn twee methoden om de druppeltjes inkt te vormen:

Thermisch: door lokale opwarming van de inkt ontstaat een kleine dampbel die voldoende druk uitoefent om een druppeltje innkt naar buiten te persen



Piëzo-elektrisch: een kristal onder invloed van spanning dient als een kleine zuiger om een druppeltje inkt naar buiten te persen



VIDEO DISPLAYS

In computerbeeldschermen bestaan twee grote technologiën: CRT of Cathode Ray Tube en LCD of Liquid Crystal Display.

Belangrijke eigenschappen voor een scherm zijn:

Afmeting: typisch gegeven in inch, veel voorkomende afmetingen zijn 15, 17, 19 of 21 inch

Aspect ratio: geeft de verhouding tussen de horizontale en verticale lengte, meestal 4:3 of 16:9
Oriëntatie: meestal landscape (breedte > hoogte), andere mogelijkheid is portrait
Maximale resolutie: maximaal aantal beeldpixels, XGA is 800x600 en UXGA is 1600x1200
Kleurdiepte: aantal verschillende kleuren die weergegeven kunnen worden, over het algemeen true color of 8 bits per basiskleur
Dot pitch: grootte van een pixel, hoe kleiner hoe gedetailleerder, meestal tussen de 0.25 en 0.31 mm
Refresh rate (voor CRT's)

De combinatie van maximale resolutie en kleurdiepte bepaalt het nodige videogeheugen. Bijvoorbeeld: 3 bytes / pixel * 800 * 600 pixels = 1,4 MiB geheugen.

Beeldschermen kunnen analoog of digitaal aangesloten worden op de computer:

Analoog: VGA of Video Graphics Array
Digitaal: DVI of Digital Video Interface

Bij digitale overdracht gaat er geen kwaliteit verloren bij overgang naar analoge signalen.

CRT

Een kathodestraalbuis heeft een elektronenkanon dat lijn per lijn van onder naar boven en van links naar rechts een elektronenstraal over het scherm beweegt. Het scherm is aan de achterzijde bedekt met een fosforlaag die oplicht wanneer het geraakt wordt door een elektronenstraal. Op die manier produceert men een beeld.

Om kleurbeelden voor te stellen zijn drie soorten fosfor nodig, een voor elke basiskleur, en drie elektronenkanonnen. Omdat de fosfor maar een beperkte tijd oplicht, is het noodzakelijk om op tijd een nieuw beeld te projectern. Een refresh rate van 75 Hz of hoger is aangeraden.

Indien men lange tijd hetzelfde beeld projecteert, brandt dit in het scherm. Daarom is het gebruikelijk met een screensaver te voorkomen dat hetzelfde beeld lange tijd blijft staan.

LCD

Deze displays zijn opgebouwd uit een matrix van vloeibare kristallen die de eigenschap hebben lichtdoorlatend te zijn afhankelijk van de spanning die erop staat. Deze spanning kan op twee manieren aangebracht worden:

Passieve matrix: de rij en kolom worden apart aangestuurd zodat op de kruising ervan bij de pixel een spanning ontstaat
Actieve matrix: heeft een afzonderlijke transistor per pixel, deze methode is veel nauwkeuriger en betrouwbaarder

De meeste LCD schermen hebben een backlight waardoor ze hun eigen licht produceren, er zijn echter ook LCD's die met een spiegel het omgevingslicht reflecteren.

EXTERNE VERBINDINGEN

Dit zijn veelvoorkomende externe verbindingen:

Naam	Snelheid
Infrarood (IrDA)	9 kB/s

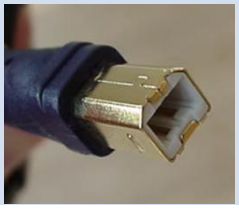
Seriële poort	14,3 kB/s
Parallelle poort	1 MB/s
USB 1.1 en USB 2.0	1,5 MB/s – 60 MB/s
PS/2 poort	
Ethernet poort	10 MB/s
SCSI poort	80 MB/s
Fibre channel (FC-AL)	400 MB/s

USB

Universal Serial Bus apparaten kunnen alle trage randapparaten vervangen. Alle apparaten op dezelfde bus hebben samen 1,5 MB/s bandbreedte ter beschikking. In versie 2.0 is dit zelfs 60 MB/s.

Op dezelfde bus kan men maximaal 127 apparaten aansluiten.

Er zijn USB-A, USB-B connectoren:

USB-A	**USB-B**
	

ASSEMBLER

GEHEUGENTRANSFER INSTRUCTIES

We bekijken nu de machinetaalinstructies die nodig zijn om informatie uit te wisselen tussen de processorregisters en het fysieke geheugen zoals beschreven in het von Neumann model.

Een gegevenstransferoperatie brengt waarden over van een bronlocatie naar een doellocatie en heeft drie parameters:

- Het bronadres
- Het doeladres
- De lengte van de gegevens

Hier een tabel met de meest courante instructies:

Instructie	**Uitleg**
mov doel, bron **mov byte ptr doel, bron**	Deze instructie kopieert de waarde van het bron naar doel. Bron en doel kunnen registers of geheugenadressen zijn.

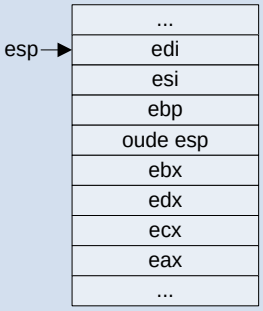
mov word ptr doel, bron **mov dword ptr doel, bron**	Men kan aan de mov instructie nog extra informatie toevoegen die de lengte van de gegevens aangeeft.
ld r, bron **st r, bron**	In moderne architecturen is er een onderscheid tussen lezen en schrijven naar het geheugen. Load en store instructies hebben 1 geheugenoperand en 1 registeroperand. De load instructie laadt een waarde uit het geheugen in een register, een store operatie laadt een waarde uit een register in het geheugen. De Pentium processor ondersteund deze instructies niet.
xchg a, b	Deze operatie wisselt de inhoud van 2 locaties.
bswap a	Wijzigt de volgorde van de bytes in een multi-byte woord. Dit komt dus overeen met het wisselen tussen little endian en big endian.
in r, port **out r, port**	Deze twee instructies verzorgen de communicatie met randapparatuur. Elk apparaat krijgt een aantal I/O poorten toegewezen uit 65536 beschikbare poorten. Door het registeroperand aan te passen aan de lengte van de gegevens (bijvoorbeeld al, ax, eax) kan men 1, 2 of 4 bytes per keer naar een randapparaat sturen.
stosb **stosw** **stosd**	Bewaart een byte, woord of dubbelwoord uit het bronregister en incrementeert of decrementeert. Deze instructies hebben geen operandi en werken dus op vaste registers in: Doeladres: edi Bronregister: al, ax of eax Increment- of decrementregister: edi De direction flag in het eflags register duidt aan of er een postincrement of een predecrement uitgevoerd wordt. Met behulp van de cld en std instructies kan de direction flag aangepast worden.
lodsb **lodsw** **lodsd**	Laadt een byte, woord of dubbelwoord uit het bronadres en incrementeert of decrementeert. Deze instructies hebben geen operandi en werken dus op vaste registers in: Doelregister: al, ax of eax Bronadres: esi Increment- of decrementregister: esi De direction flag in het eflags register duidt aan of er een postincrement of een predecrement uitgevoerd wordt. Met behulp van de cld en std instructies kan de direction flag aangepast worden.
movsb **movsw** **movsd**	Verplaatst een byte, woord of dubbelwoord van geheugen naar geheugen en incrementeert of decrementeert. Deze instructies hebben geen operandi en werken dus op vaste registers in: Doeladres: edi Bronadres: esi Increment- of decrementregisters: edi en esi De direction flag in het eflags register duidt aan of er een postincrement of

	een predecrement uitgevoerd wordt. Met behulp van de cld en std instructies kan de direction flag aangepast worden.

STACK INSTRUCTIES

De stack is een gegevensstructuur met LIFO-gedrag (last in, first out). Een stack of stapel heeft een beginpunt in het geheugen en groeit in een bepaalde richting (meestal richting de dalende geheugenadressen).

Men kan enkel aan de top van de stack een element toevoegen (pushen) of een element verwijderen (poppen). De stapelwijzer ESP wijst steeds naar de top van de stapel en houdt zo bij hoever de stapel al gegroeid is.

Instructie	Uitleg
push bron	Voegt de waarde in het register bron toe aan de stack en past de stack pointer aan. Met *push esp* zetten we de waarde van esp op de stack van voor het uitvoeren van de instructie.
pop doel	Herstelt de bovenste waarde in de stack naar het register doel en past de stack pointer aan.
pushad	Bewaart alle registers op de stack en past de stack pointer aan. De volgorde waarin de registers op de stack staan:  De waarde van esp op de stack is de waarde bij het begin van de operatie.
popad	Herstelt alle registers met behulp van de topwaarden in de stack en past de stack pointer aan. Zie *pushad* voor de volgorde van de registers op de stack. Ook bij het herstellen wordt de waarde voor esp pas herstelt na eax. Anders zouden de waarden voor ebx, edx, ecx en eax verloren gaan.
pushfd	Bewaart het toestandsregister in de stack en past de stack pointer aan.
popfd	Herstelt het toestandsregister met de topwaarde van de stack en past de stack pointer aan. Sommige bits worden niet herstelt om mogelijke problemen te voorkomen.

REGISTER INSTRUCTIES

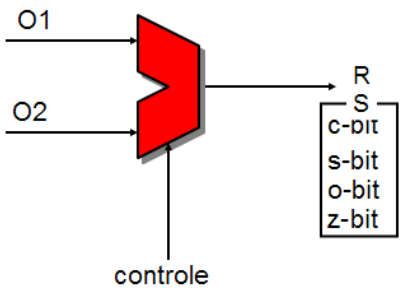
Met deze instructies is het mogelijk om toegang te krijgen tot de speciale registers die niet bereikbaar zijn met de normale geheugeninstructies.

Instructie	Uitleg
lahf	Laad de laagste 16 bits van het eflags register (flags) in het ah register.
sahf	Wijzigt de S, Z, A, P en C bits in het eflags register met de waarden uit het ah register.
cld	Zet de direction flag in het eflags register af. Hierdoor voeren de volgende instructies een postincrement uit: stosb, stosw, stosd lodsb, lodsw, lodsd movsb, movsw, movsd Voor een omschrijving van deze instructies kan u kijken bij geheugentransfer instructies.
std	Zet de direction flag in het eflags register aan. Hierdoor voeren de volgende instructies een predecrement uit: stosb, stosw, stosd lodsb, lodsw, lodsd movsb, movsw, movsd Voor een omschrijving van deze instructies kan u kijken bij geheugentransfer instructies.
stc	Zet de carry flag in het toestandsregister aan.
clc	Zet de carry flag in het toestandsregister af.
sti	Zet de interrupt flag in het toestandsregister aan.
cli	Zet de interrupt flag in het toestandsregister af.

ARITHMETISCHE INSTRUCTIES

Een arithmetische operator heeft als inputs 2 operandi en controlesignalen en zet deze om naar een output en statusbits. Deze statusbits worden ogpeslaan in het toestandsregister van de CVE. Een aantal van de statusbits:

C-bit: carry, overdracht uit de meest beduidende bit
S-bit: sign, geeft het teken van het resultaat aan
O-bit: overflow, het resultaat is te groot of te klein om te worden voorgesteld
Z-bit: zero, het resultaat is nul



We definiëren de volgende notaties:

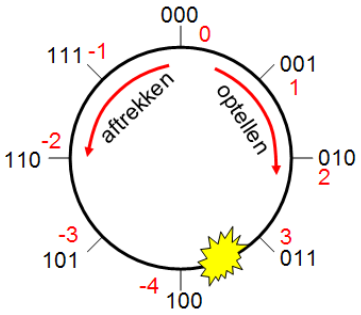
$A\langle m|n\rangle$ verwijst naar het getal voorgesteld door het bitpatroon te vinden in A tussen de m^{de} en n^{de} bit, met 0 de minst beduidende
$carry(m) = O1\langle m|m\rangle + O2\langle m|m\rangle + carry(m-1)$ met $carry(-1) = 0$, deze functie definieert of er een overdracht is bij de bitsommatie van bit m

Een overflow valt bij de 2-complementgetallen van N bits te detecteren als:

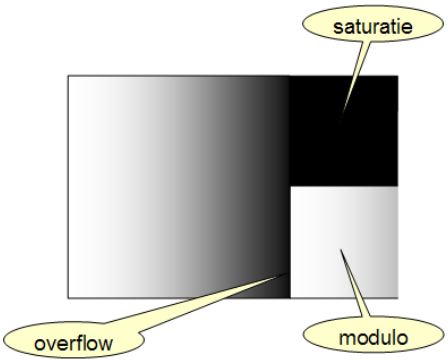
$$O = carry(N-1)\ XOR\ carry(N-2)$$

Een overflow kan op 2 manieren afgehandeld worden:

Via modulorekenen: hier beginnen werken we met de getallencirkel en tellen we bij een overflow gewoon verder in de cirkel



Via saturatierekenen: bij een overflow wordt de waarde beperkt tot de extremaal voorstelbare waarde, hoewel het resultaat verkeerd is, levert het meestal een resultaat op dat dichter bij de verwachtingen ligt



Bij het vergelijken van natuurlijke getallen gebruikt men de volgende toestandsbits:

Vergelijking	Binair	2-complement
a < b	C = 1	S XOR O = 1
a ≥ b	C = 0	S XOR O = 0
a = b	Z = 1	Z = 1
a ≤ b	C OR Z = 1	(S XOR O) OR Z = 1
a > b	C OR Z = 0	(S XOR O) OR Z = 0

Enkele courante arithmetische instructies:

Instructie	Uitleg
add doel, bron	Sommeert bron en doel en slaat het resultaat op in doel.
adc doel, bron	Sommeert bron en doel en telt hierbij de carry-bit van de vorige operatie. Deze operatie is interessant indien men brede getallen opsplitst in kleinere operaties ter grootte van de woordbreedte.
sub doel, bron	Trekt bron van doel af en slaat het resultaat op in doel.
sbb doel, bron	Trekt bron en doel af en trekt hiervan de carry-bit van de vorige operatie af. Deze operatie is interessant indien men brede getallen opsplitst in kleinere operaties ter grootte van de woordbreedte.
mul bron	Unsigned vermenigvuldiging Als bron 8 bits is: vermenigvuldigt al met bron en slaat het resultaat op in ax Als bron 16 bits is: vermenigvuldigt ax met bron en slaat het resultaat op in dx:ax Als bron 32 bits is: vermenigvuldigt eax met bron en slaat het resultaat op in edx:eax
imul bron	Signed vermenigvuldiging Als bron 8 bits is: vermenigvuldigt al met bron en slaat het resultaat op in ax Als bron 16 bits is: vermenigvuldigt ax met bron en slaat het resultaat op in dx:ax Als bron 32 bits is: vermenigvuldigt eax met bron en slaat het resultaat op in edx:eax
imul doel, bron	Vermenigvuldigt bron met doel en slaat het resultaat op in doel.
imul doel, bron1, bron2	Vermenigvuldigt bron1 met bron2 en slaat het resultaat op in doel.
div bron	Unsigned deling Als bron 8 bits is: deelt ax door bron en slaat het quotiënt op in al en de rest in ah Als bron 16 bits is: deelt dx:ax door bron en slaat het quotiënt op in ax en de rest in dx Als bron 32 bits is: deelt edx:eax door bron en slaat het quotiënt op in eax en de rest in edx
idiv bron	Signed deling Als bron 8 bits is: deelt ax door bron en slaat het quotiënt op in al en de rest in ah Als bron 16 bits is: deelt dx:ax door bron en slaat het quotiënt op in ax en de rest in dx Als bron 32 bits is: deelt edx:eax door bron en slaat het quotiënt op in eax en de rest in edx De rest in edx is altijd hetzelfde teken als de bron.
neg doel	Draait het teken om van doel en slaat het resultaat terug op in doel.

inc doel	Incrementeert doel en slaat het resultaat op in doel.
dec doel	Decrementeert doel en slaat het resultaat op in doel.
cmp doel, bron	Vergelijkt doel en bron door ze van elkaar af te trekken, maar gooit het resultaat weg. Enkel de toestandsbits worden gehouden.
test doel, bron	Doet bron AND doel en gooit het resultaat weg. De toestandsbits blijven echter behouden. Deze instructie kan gebruikt worden in combinatie met een masker om te kijken of bepaalde bits aan staan door te kijken of het resultaat 0 was.
lea doel, expressie	Load Effective Address, rekent de adresexpressie uit en slaat het resultaat op in doel.
xadd doel, bron	Slaat in bron de waarde van doel op, en in doel de som van de oorspronkelijke waarde van bron en doel.
cmpxchg doel, bron	Als eax gelijk is aan doel, stelt de instructie doel gelijk aan bron. In het andere geval wordt eax gelijk gesteld aan doel. Deze instructie wordt gebruikt om op atomaire wijze een waarde in het geheugen aan te passen.

LOGISCHE INSTRUCTIES

Enkele courante logische instructies:

Instructie	Uitleg
and doel, bron	Voert bron AND doel uit en slaat het resultaat op in doel.
or doel, bron	Voert bron OR doel uit en slaat het resultaat op in doel.
xor doel, bron	Voert bron XOR doel uit en slaat het resultaat op in doel.
not doel	Voert een bitsgewijze inversie uit op doel en slaat het resultaat op in doel.
shl doel, aantal **sal doel, aantal**	Shift Left, schuift doel aantal posities op naar links, rehcts worden nullen ingevoegd. Het resultaat wordt opgeslaan in doel. Deze operatie is numeriek gelijk aan het vermenigvuldigen met 2.
shr doel, aantal	Shift Right, shcuift doel aantal posities op naar rechts, links worden nullen ingevoegd. Het resultaat wordt opgeslaan in doel. Deze operatie is numeriek gelijk aan het delen door 2 (enkel voor binaire getallen).
sar doel, aantal	Shift Arithmetic Right, gelijkaardig als Shift Right (shr) maar voegt links in plaats van nullen hetzelfde teken als de tekenbit in. Deze operatie is numeriek gelijk aan het delen door 2 (voor binaire en 2-complement getallen).
shld doel, bron, aantal	Voert een logische shiftoperatie naar links uit op het registerpaar doel:bron en slaat de 32 meest significante bits op in doel. Dit komt erop neer dat er aantal linkerbits van bron rechts ingeschoven

	worden bij doel.
shrd doel, bron, aantal	Voert een logische shiftoperatie naar rechts uit op het registerpaar bron:doel en slaat de 32 minst significante bits op in doel. Dit komt erop neer dat aantal rechterbits van bron links ingeschoven worden bij doel.
rol doel, aantal	Rotate Left, shift de bits in doel aantal posities naar links en plaatst de afgeschoven bits er aan de rechterkant terug bij.
ror doel, aantal	Rotate Right, shift de bits in doel aantal posities naar rechts en plaatst de afgeschoven bits er aan de linkerkant terug bij.
rcl doel, aantal	Rotate with Carry Left, roteert de bits in doel aantal posities naar links maar dan met 33 bits (een extra carry bit).
rcr doel, aantal	Rotate with Carry Right, roteert de bits in doel aantal posities naar rechts maar dan met 33 bits (een extra carry bit).
bt doel, n	Bewaart de n^{de} bit van doel in de carry-bit. Hierbij begint men te tellen met 0 vanaf de minst significante bit.
bts doel, n	Bewaart de n^{de} bit van doel in de carry-bit en plaatst hem dan op 1. Hierbij begint men te tellen met 0 vanaf de minst significante bit.
btr doel, n	Bewaart de n^{de} bit van doel in de carry-bit en plaatst hem dan op 0. Hierbij begint men te tellen met 0 vanaf de minst significante bit.
btc doel, n	Bewaart de n^{de} bit van doel in de carry-bit en inverteert daarna die bit. Hierbij begint men te tellen met 0 vanaf de minst significante bit.
bsf doel, bron	Zoekt voorwaarts in bron naar de eerste bit die op 1 staat en slaat het resultaat op in doel. Hierbij begint men te tellen met 0 vanaf de minst significante bit. Indien er geen bit op 1 staat wordt de Z-toestandsbit op 1 geplaatst.
bsr doel, bron	Zoekt achterwaarts in bron naar de eerste bit die op 1 staat en slaat het resultaat op in doel. Hierbij begint men te tellen met 0 vanaf de minst significante bit. Indien er geen bit op 1 staat wordt de Z-toestandsbit op 1 geplaatst.
set<cc>doel	Controleert in het toestandsregister of de conditie <cc> waar is en zet doel op 1 als de conditie waar is en op 0 als de conditie vals is. De conditiecodes voor <cc> zijn: z: zero c: carry o: overflow p: parity s: sign nz: no zero nc: no carry no: no overflow np: no parity ns: no sign

	g: greater nle: not less or equal ge: greater or equal nl: not less l: less nge: not greater or equal le: less or equal ng: not greater e: equal a: above nbe: not below or equal ae: above or equal nb: not below b: below nae: not above or equal be: below or equal na: not above

FLOATING-POINT INSTRUCTIES

De floating-point instructies maken gebruik van de floating-point registers die we eerder hebben beschreven.

De floating-point rekeneenheid ondersteung niet alleen floats. Ze kan rekenen met de volgende datatypes:

Single precision float (32 bit)
Double precision float (64 bit)
Double extended precision float (80 bit)
Integer woord (16 bit)
Integer dubbelwoord (32 bit)
Integer quadwoord (64 bit)
BCD tekenbyte met 18 nibbles (80 bit)

Aangezien de registers voor floating-point berekeningen een stapelstructuur hebben, zijn er een speciale adresseermodes

Zonder operandi worden als operandi ST0 en ST(i) ondersteld
Met één operand, wordt het eerste operand ST0 ondersteld en het tweede het opgegeven operand
Als men twee operandi opgeeft, moet één ervan ST0 zijn
Geheugenadressering gebeurt met <adres>

Enkele courante floating-point instructies:

Instructie	**Uitleg**
fadd doel, bron	Sommeert bron en doel en slaat het resultaat op in doel.
fsub doel, bron	Trekt bron af van doel en slaat het resultaat op in doel.
fdiv doel, bron	Deelt doel door bron en slaat het resultaat op in doel.
fprem	Geeft de rest bij het delen van ST0 door ST1 en slaat het resultaat op in ST0. Nota: deze operatie geeft een “partiële rest”, soms kan het zijn dat het

	resultaat maar een tussenresultaat is. Als regel geldt dat men fprem moet blijven uitvoeren tot de C2-toestandsbit op 0 staat.
fmul doel, bron	Vermenigvuldigt bron en doel en slaat het resultaat op in doel.
fabs	Berekent de absolute waarde van ST0 en slaat het resultaat op in ST0.
fchs	Inverteert de tekenbit van ST0 en slaat het resultaat terug op in ST0.
fcomi a, b	Vergelijkt a en b en past het toestandsregister aan.
fsqrt	Berekent de vierkantswortel van ST0 en slaat het resultaat op in ST0
fsin	Berekent de sinus van ST0 en slaat het resultaat op in ST0.
fcos	Berekent de cosinus van ST0 en slaat het resultaat op in ST0.
fptan	Berekent de tangens van ST0 en slaat het resultaat op in ST0.
fpatan	Berekent der arctangens van ST0 en slaat het resultaat op in ST0.
fyl2x	Vermenigvuldigt ST1 met $\log_2 ST0$, slaat het resultaat op in ST1 en haalt ST0 van de stack. Op het einde zit het resultaat dus in ST0.
fyl2xp1	Vermenigvuldigt ST1 met $\log_2(ST0 + 1)$, slaat het resultaat op in ST1 en haalt ST0 van de stack. Op het einde zit het resultaat dus in ST0.
f2xm1	Berekent $2^{ST0} - 1$ en slaat het resultaat op in ST0. Hierbij moet ST0 in het begin in $[-1, 1]$ liggen.
fbld bron	Laadt een BCD gecodeerd getal uit adres bron en pusht het op de register stack.
fbstp doel	Slaat ST0 op als BCD gecodeerd getal op adres doel en popt de register stack.
fild bron	Laadt een integer uit adres bron en pusht het op de register stack. De integer kan 16, 32 of 64 bit zijn.
fist doel	Slaat ST0 op als integer op adres doel. De integer kan 16, 32 of 64 bit zijn.
fld bron	Laadt een float uit adres bron en pusht het op de register stack. De float kan 32, 64 of 80 bit zijn.
fst doel	Slaat ST0 op als float op adres doel. De float kan 32, 64 of 80 bit zijn.
fld1	Pusht de waarde 1 op de register stack.
fldl2t	Pusht $\log_2 10$ op de register stack.
fldl2e	Pusht $\log_2 e$ op de register stack.
fldpi	Pusht op π de register stack.
fldlg2	Pusht $\log_{10} 2$ op de register stack.
fldln2	Pusht $\log_e 2$ op de register stack.
fldz	Pusht 0 op de register stack.

MMX INSTRUCTIES

Met de opkomst van het internet kende de jaren ’90 een explosie aan multimediale data. Deze elementen in deze data kenmerken zich door:

Kleine grootte
Groot aantal
Onafhankelijk van elkaar

Een oplossing voor dit probleem is deze kleine elementen te groeperen in blokken die in een woord passen. Bijvoorbeeld 4 blokken van 16 bits passen in een 64 bits woord.

Om deze functionaliteit aan de Pentium toe te voegen, introduceerde Intel de MMX instructies. Deze maken gebruik van de registers van de floating-point eenheid, maar maken enkel gebruik van 64 van de 80 bits. Hier passen 8 bytes, 4 woorden, 2 dubbelwoorden of 1 quadwoord in.

De MMX extensie bevat 57 nieuwe instructies die toelaten om gegevens in geschikte vorm in de registers te krijgen en er parallelle bewerkingen op uit te voeren.

Enkele courante MMX instructies:

Instructie	Uitleg
paddb doel, bron **paddw doel, bron** **paddd doel, bron** **paddq doel, bron**	Telt parallel de bytes, woorden, dubbelwoorden of quadwoorden van doel en bron op en slaat het resultaat op in doel.
paddsb doel, bron **paddsw doel, bron**	Telt parallel de signed bytes, woorden, dubbelwoorden of quadwoorden van doel en bron op en slaat het resultaat op in doel. Deze instructies gebruiken saturatie om overflows af te handelen.
paddusb doel, bron **paddusw doel, bron**	Telt parallel de unsigned bytes, woorden, dubbelwoorden of quadwoorden van doel en bron op en slaat het resultaat op in doel. Deze instructies gebruiken saturatie om overflows af te handelen.
psubb doel, bron **psubw doel, bron** **psubd doel, bron** **psubq doel, bron**	Trekt paralleel de bytes, woorden, dubbelwoorden of quadwoorden van bron af van doel en slaat het resultaat op in doel.
psubsb doel, bron **psubsw doel, bron**	Trekt parallel de signed bytes, woorden, dubbelwoorden of quadwoorden van bron af van doel en slaat het resultaat op in doel. Deze instructies gebruiken saturatie om overflows af te handelen.
psubusb doel, bron **psubusw doel, bron**	Trekt parallel de unsigned bytes, woorden, dubbelwoorden of quadwoorden van bron af van doel en slaat het resultaat op in doel. Deze instructies gebruiken saturatie om overflows af te handelen.
pmullw doel, bron	Vermenigvuldigt parallel de woorden in doel en bron en slaat de minst beduidende 16 bits van de resultaten op in doel.
pmulhw doel, bron	Vermenigvuldigt parallel de woorden in doel en bron en slaat de meest beduidende 16 bits van de resultaten op in doel.

pmaddwd doel, bron	Vermenigvuldigt parallel de woorden in doel en bron, sommeert de opeenvolgende elementen 2 aan 2 en slaat de resultaten op in doel. Hier zijn de elementen dus van 4 keer 16 bits naar 2 keer 32 bits gegaan en hebben we veel minder kans op overflow.
pcmp<cc>d doel, bron	Vergelijkt parallel de dubbelwoorden in doel en bron en slaat het resultaat op in doel. Het elk element in het resultaat bevat ofwel allemaal 1'en indien de test slaagde, ofwel allemaal 0'en indien de test niet slaagde. <cc> is hier opnieuw een van de mogelijkheden zoals bij de set<cc> instructie bij logische operatoren.
psraw doel, aantal **psrad doel, aantal**	Voert arithmetische shifts naar rechts uit op elk van de elementen. De ingevoegde bits zijn gelijk aan de sign bit van het element.
psllw **pslld** **psllq**	Voert arithmetische shifts naar links uit op elk van de elementen. De ingevoerde bits zijn 0.
psrlw **psrld** **psrlq**	Voert arithmetische shifts naar rechts uit op elk van de elementen. De ingevoegde bits zijn 0.
pand doel, bron	Voert een bitsgewijze bron AND doel uit en slaat het resultaat op in doel.
pandn doel, bron	Berekent eerst het complement van doel, voert dan bron AND doel uit en bewaart het resultaat in doel.
por doel, bron	Voert een bitsgewijze bron OR doel uit en slaat het resultaat op in doel.
pxor doel, bron	Voert een bitsgewijze bron XOR doel uit en slaat het resultaat op in doel.
punpckhbw doel, bron **punpckhwd doel, bron** **punpckhdq doel, bron**	Creëert een nieuwe vector van elementen door afwisselend een element uit de hoogste helft van bron te nemen en dan een element uit de hoogste helft van doel. De nieuwe vector wordt opgeslaan in doel.
punpcklbw doel, bron **punpcklwd doel, bron** **punpckldq doel, bron**	Creëert een nieuwe vector van elementen door afwisselend een element uit de laagste helft van bron te nemen en dan een element uit de laagste helft van doel. De nieuwe vector wordt opgeslaan in doel.
packsswb doel, bron **packssdw doel, bron**	Deze operaties reduceren woorden naar bytes of dubbelwoorden naar woorden op saturerende wijze. De elementen worden signed ondersteld. Eerst worden de elementen van doel gepacked, gevolgd door de elementen van bron. Het resultaat wordt opgeslaan in doel.
packuswb	Deze operaties reduceren woorden naar bytes of dubbelwoorden naar woorden op saturerende wijze. De elementen worden unsigned ondersteld. Eerst worden de elementen van doel gepacked, gevolgd door de elementen van bron. Het resultaat wordt opgeslaan in doel.
emms	Geeft het einde van het gebruik van de MMX extensie aan. Hierdoor worden

	de ST0-ST7 registers terug correct ingesteld.

SSE/SSE2 INSTRUCTIES

Deze instructiesets geven gelijkaardige functionaliteit als de MMX extensie, maar dan voor single precision floating-point getallen. Er werden 8 nieuwe registers geïntroduceerd: XMM0-XMM7. Deze registers zijn 128 bits groot en kunnen dus 4 single precision floats bevatten.

Enkele SSE instructies:

Instructie	Uitleg
addps doel, bron	Telt vier single precision elementen in bron en doel op en bewaart het resultaat in doel.
addss doel, bron	Telt het meest rechtse single precision element in bron en doel op en bewaart het resultaat in doel. De andere drie elementen blijven dus ongewijzigd.

Met de komst van SSE2, is het mogelijk om het parallellisme uit te breiden naar integers en double precision floats. Het is dus niet meer nodig om de MMX extensies te gebruiken.

Enkele SSE2 instructies:

Instructie	Uitleg
addpd doel, bron	Telt twee double precision elementen in bron en doel op en bewaart het resultaat in doel.
addsd doel, bron	Telt het meest rechtse double precision element in bron en doel op en bewaart het resultaat in doel. Het andere element blijft dus ongewijzigd.
shufps doel, bron, pattern	Combineert twee elementen uit bron en twee elementen uit doel tot een patroon in doel. Met behulp van pattern kan men kiezen welke elementen uit bron en doel gecombineerd moeten worden. Pattern is 8 bytes groot, 2 bytes per element.

CONTROLETRANSFER INSTRUCTIES

Normaalgezien verloopt het uitvoeren van instructies volkomen sequentieel. Controletransfer instructies zijn instructies die dit sequentiële verloop veranderen. Ze bereiken dit door de instructiewijzer of program counter aan te passen, dit is het register eip. Dit register wijst naar de volgende instructie die uitgevoerd moet worden.

SPRONG INSTRUCTIES

Er zijn drie soorten spronginstructies, we zullen ze elk bespreken.

We kunnen een programma met sprongen als volgt visualiseren. Beschouwen we basisblokken van instructies die sowieso samen uitgevoerd worden en waarvoor geen sprong bestaat die midden in het blok wijst. De aaneenschakelijk van deze blokken met pijlen voor de sprongen heet een controleverloopgraaf en toont alle mogelijke controlepaden doorheen het programma. Volle pijlen wijzen op een effectieve sprong, gestippelde op een doorvalpad.

ONVOORWAARDELIJKE SPRONGEN

Onvoorwaardelijke sprongen of jumps zorgen ervoor dat het programma zonder meer voortgaat vanaf het adres dat de instructive aanduidt.

Voor deze sprongen is maar een instructie nodig: *jmp adres*. Deze instructie springt onvoorwaardelijk naar het opgegeven adres.

VOORWAARDELIJKE SPRONGEN

Bij een voorwaardelijke sprong of branch, gaat het programma verder op het adres dat de instructive aanduidt indien aan een gestelde voorwaarde voldaan is, zoniet gaat het programma sequentieel verder.

De volgende conditionele spronginstructies zijn beschikbaar:

Instructie	Uitleg
jz adres	Springt naar adres als de zero toestandsbit op 1 staat.
jc adres	Springt naar adres als de carry toestandsbit op 1 staat.
jo adres	Springt naar adres als de overflow toestandsbit op 1 staat.
jp adres	Springt naar adres als de parity toestandsbit op 1 staat.
js adres	Springt naar adres als de sign toestandsbit op 1 staat.
jnz adres	Springt naar adres als de zero toestandsbit op 0 staat.
jnc adres	Springt naar adres als de carry toestandsbit op 0 staat.
jno adres	Springt naar adres als de overflow toestandsbit op 0 staat.
jnp adres	Springt naar adres als de parity toestandsbit op 0 staat.
jns adres	Springt naar adres als de sign toestandsbit op 0 staat.
jg adres **jnle adres**	Springt naar adres als bij de vorige vergelijking groter dan geldt. Deze instructies zijn voor 2-complement voorstelling
jge adres **jnl adres**	Springt naar adres als bij de vorige vergelijking groter dan of gelijk aan geldt. Deze instructies zijn voor 2-complement voorstelling
jl adres **jnge adres**	Springt naar adres als bij de vorige vergelijking kleiner dan geldt. Deze instructies zijn voor 2-complement voorstelling
jle adres **jng adres**	Springt naar adres als bij de vorige vergelijking kleiner dan of gelijk aan geldt. Deze instructies zijn voor 2-complement voorstelling
je adres	Springt naar adres als bij de vorige vergelijking gelijk aan geldt. Is identiek aan *jz adres*.
ja adres **jnbe adres**	Springt naar adres als bij de vorige vergelijking groter dan geldt. Deze instructies zijn voor binaire voorstelling
jae adres	Springt naar adres als bij de vorige vergelijking groter dan of gelijk aan geldt.

jnb adres	Deze instructies zijn voor binaire voorstelling
jb adres **jnae adres**	Springt naar adres als bij de vorige vergelijking kleiner dan geldt. Deze instructies zijn voor binaire voorstelling
jbe adres **jna adres**	Springt naar adres als bij de vorige vergelijking kleiner dan of gelijk aan geldt. Deze instructies zijn voor binaire voorstelling

BEREKENDE SPRONGEN

Bij een statische sprong is het adres vast gegeven. Bij een berekende sprong is het adres het gevolg van een voorgaande berekening.

Vaak worden de sprongen ook relatief uitgedrukt. In plaats van een vaste locatie in het geheugen, wordt dan verwezen naar een relatieve positie ten opzichte van de eip. Het voordeel hiervan is dat wijzigingen in andere delen van de code er niet voor zorgen dat alle sprongen aangepast moeten worden, dit is positie-onafhankelijke code. Voorbeeld: *jmp eip – 20*.

LUS INSTRUCTIES

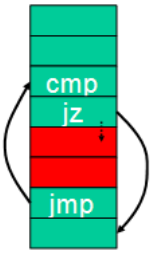
Een lus is een programmadeel dat zich herhaalt tot aan een bepaalde conditie voldaan is. Een specifieke instructie hiervoor is de *loop adres* instructie. Deze decrementeert elke uitvoering het register ecx en springt dan naar adres tenzij ecx gelijk is aan 0.

We bespreken enkele veel voorkomende lussen.

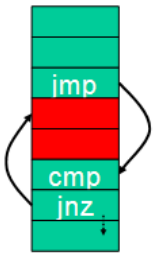
WHILE/FOR

Er zijn twee mogelijke strategiën:

Men plaatst de test vooraan en springt uit de lus indien de test faalt, op het einde van de lus is opnieuw een sprong nodig om in de lus te blijven. Het voordeel hier is dat het doorvalpad het meest zal voorvallen.



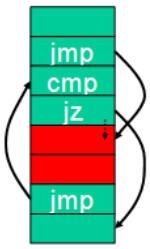
Men plaatst de test achteraan en springt voor de lus eerst naar de test die naar de eerste instructie van de lus springt als de test slaagt. Het voordeel hier is dat er geen twee sprongen per lusiteratie nodig zijn.



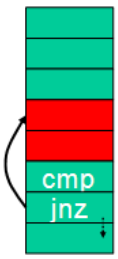
DO-WHILE

Er zijn opnieuw twee mogelijke strategiën:

Men plaatst de test vooraan en springt voor de lus er rechtstreeks in voorbij de test, op het einde van de lus is opnieuw een sprong nodig die naar de test springt. Het voordeel hier is dat het doorvalpad het meest zal voorvallen.



Men plaatst de test achteraan en die springt naar de eerste instructie van de lus als de test slaagt. Het voordeel hier is dat er geen twee sprongen per lusiteratie nodig zijn.



PROCEDURE INSTRUCTIES

Bij een functie-oproep wenst men een ander stuk code uit te voeren en nadien verder te kunnen doen waar men gestopt was. De instructiewijzer op het moment van de oproep mag dus niet verloren gaan.

Deze instructies zorgen voor deze functionaliteit:

Instructie	Uitleg
call adres	Pusht eip op de stack en springt naar adres.
call r, adres **bal r, adres**	Slaat eip op in het register r en springt naar adres. Door in de subroutine terug te jumpen naar de waarde in r, keert men terug naar de aanroepende routine.
ret	Popt eip terug van de stack.
ret n	Popt eip van de stack en verhoogt daarna esp met n.

Door eip tijdelijk op de stack op te slaan kunnen we hem na de functie-aanroep herstellen en voortgaan met de oorspronkelijke routine.

In de controleverloopgraaf, tekenen we een volledige pijl naar de aangeroepen functie en een doorvalpijl naar de instructie volgend op call.

Het is aangeraden om in subroutines alle registers die aangepast worden eerst naar de stack te pushen en op het einde terug te poppen, anders kunnen waarden van de aanroepende functie in deze registers overschreven worden.

Indien men weinig registers ter beschikking heeft, kan men parameters voor een subroutine op de stack pushen. Men mag dan wel niet vergeten om ze terug van de stack af te halen door ze ofwel te poppen, ofwel door esp aan te passen. Dit opruimen van de stack kan ofwel in de aanroepende functie, ofwel in de subroutine zelf gebeuren. In het laatste geval kan met de instructie *ret n* gebruiken.

Bij het creëren van lokale variabelen is het gebruikelijk om deze op de stack te alloceren. Hierdoor kan men eenvoudig recursieve functies ondersteunen.

Het opzetten van een stack frame kan gebeuren met de volgende instructies:

Instructie	Uitleg
enter n	Pusht ebp op de stack en slaat esp op in ebp, ebp wijst nu naar zichzelf op de stack. Daarna wordt esp verlaagd met n zodat er n bytes aan geheugen gereserveert zijn voor lokale variabelen.
leave	Plaatst esp terug naar ebp en popt ebp van de stack.

Tussen een oproepende functie en de opgeroepen functie moeten de volgende afspraken gemaakt zijn:

Worden de argumenten meegegeven via de stack of via registers?
In welke volgorde worden de argumenten doorgegeven?
Welk register fungeert als stapelwijzer?
Worden register gepusht door de oproeper of de opgeroepene?
Komt het terugkeeradres op de stapel of in een register?

VARIA

Enkele algemene instructies:

Instructie	Uitleg
nop	Doe niets
lock {instructie}	Maakt instructie atomair, dit is enkel mogelijk bij bepaalde instructies.

OPTIMALISATIES

IDEMPOTENTIE

Als men een register pusht, het niet gebruikt en terug popt, kan men de push en pop weglaten.

DODE WAARDEN

Als men een waarde kopieert naar een register, maar daarna het register nooit gebruikt, kan men de transferinstructie weglaten.

KOPIEPROPAGATIE

Indien een waarde berekend wordt in register A om daarna gekopieerd te worden in register B, dank an men het beter meteen in B opslaan.

BASE POINTER VERWIJDEREN

Door alle referenties naar ebp te veranderen naar referenties van esp (met de nodige aanpassing van de offsets), kan men het gebruik van ebp elimineren. ebp wordt een dode waarde.

TIJDELIJKE REGISTERS

Indien men twee registers gebruikt om tijdelijke warden op te slaan, en men heft de warden niet op overlappende tijdstippen nodig, dank an men een van beide veranderen door de andere.

DOORVALSPRONG

Sprongen naar het doorvalpad kan men weglaten.

INLINING

Kleine functies kunnen rechtstreeks in de aanroepende code ingevoegd worden. Dit spaart de call en ret instructies uit.

INSTRUCTIECODERING

Instructiecodering is de binaire representative van een instructive zoals die gezien wordt door de processor. Zelfs bij het programmeren op lage niveau's komt men hier meestal niet mee in aanraking. Elke instructie krijgt een opcode die indentificeert welke operatie uitgevoerd wordt, daarna volgt extra informatie over de datagrootte, registers of adresexpressies, constanten, …

De handleiding van de processor zal elke instructie volledig beschrijven, inclusief de toestandsbits die ze aanpast en het aantal klokcycli die nodig zijn om ze uit te voeren.

MACHINETYPES

Hoewel geen enkele architectuur dezelfde is, kunnen we algemeen enkele grote types onderscheiden afhankelijk van het aantal registersen hun toegankelijkheid.

STAPELMACHINES

Bji een stapelmachine zijn de registers georganiseerd in een stack structuur. Alle instructies halen hun operandi rechtstreeks van de stack en plaatsen het resultaat terug op de stack. De instructies zijn dus heel klein. Met push en pop instructies kan men de operandi op de stack plaatsen.

Bijvoorbeeld: HP48 zakrekenmachine, Java Virtual Machine

ACCUMULATORMACHINES

Bij een accumulatormachine wordt een van de operandi het accumulator register ondersteld. De andere registers zijn individueel bereikbaar. Het resultaat wordt altijd in de accumulator geschreven. De instructies zijn vrij kort aangezien een van de operandi impliciet ondersteld wordt. Nadeel is dat het systeem niet zo flexibel is, men moet steeds de accumulator klaarzetten. Meestal hebben deze machines weinig registers, en dit machinetype werd dan ook vaak gebruikt in de tijd dat de technologie het aantal registers beperkte.

Bijvoorbeeld: klassieke rekenmachines (accumulator is het display)

REGISTERMACHINES

Een registermachine heeft vrije toegang tot alle registers. Er zijn geen impliciete operandi. Sommige architecturen schrijven het resultaat naar een van de operandi, maar flexibiler zijn de architecturen waarbij men het doelregister kan kiezen.

Een belangrijk onderscheid bij deze machines is hoeveel geheugenlocaties er kunnen opgegeven worden. Indien men geen geheugenlocaties mag meegeven, dan noemt men dit een load/store architectuur. In het andere geval noemt men het een geheugen-naar-geheugen architectuur indien men tegelijk bron en doel een geheugenlocatie mag maken, en anders een geheugen-naar-register architectuur.

Een geheugen-naar-geheugen machine is niet automatisch de beste oplossing. Aangezien drie adresexpressies meegegeven worden, zijn de instructies veel groter dan bij de andere machinetypes. Bovendien is geheugen-naar-geheugen complexer en vereist dit dus een tragere werking.

CODE BOUWEN

Het proces waarbij een hoge-niveauprogrammeertaal omgezet wordt tot een uitvoerbaar bestand voor een bepaald platform bevat een aantal stappen:

1. Eerst zet de compiler de broncode om naar machinecode, dit levert een objectbestand. De meeste programmeeromgevingen ondersteunen meerdere objectbestanden zodat men het overzicht houdt bij grote projecten.
2. De linker combineert alle nodige objectbestanden tot een uitvoerbaar bestand en voegt eventueel nog routines uit bibliotheken toe.

DE COMPILER

Het compileren van broncode is een complexe opgave. Algemeen bestaat ze uit de volgende stappen:

1. Lexicale analyse: interpretatie van de ruwe tekst tot lexemen, elementaire taalelementen
2. Syntactische analyse: interpretatie van de volgorde van lexemen tot een programmastructuur
3. Semantische analyse: controle van declaratie, scope en datatype van variabelen
4. Optimalisatie: poging om het programma te optimaliseren volgens de regels beschreven in optimalisaties.
5. Codegeneratie: omzetting van de de programma-expressies tot assemblerexpressies
6. Scheduling: veranderen van de volgorde van instructies, berekeningen en geheugentoegangen om het programma te versnellen

De compiler produceert een objectbestand dat uit 3 delen bestaat:

Globale veranderlijken: zijn beschikbaar in de hele module en soms zelfs buiten het objectbestand
Instructies: de programmacode

Extra informatie om de linker te helpen en voor tijdens de uitvoering: naam, grootte van het bestand, types geheugen, adres van de main functie, eventueel gebruikte externe routines, …

DE LINKER

De linker voegt een set gerelateerde objectbestanden samen tot een uitvoerbaar bestand. In een eerste fase betekent dit het samenbrengen van de informatie en het oplossen van alle externe verwijzingen.

DE LADER

Dit is een onderdeel van het besturingssysteem dat uitvoerbare bestanden kan laden in het geheugen om de uitvoering te starten. Dit komt onder andere neer op het inladen van de nodige datasegmenten voor veranderlijken en instructies en het klaarzetten van esp en eip.

Bovendien moet de lader ook nog enkele finale adresaanpassingen doen in het uitvoerbare bestand zodat de code herpositioneerbaar wordt. De informatie welke adressen aangepast moeten worden zit in het uitvoerbaar bestand.

Eenmaal het programma gestart is, kan het aan het besturingssysteem extra geheugenruimte in de heap vragen. Hier kan indien nodig dynamisch geheugen gealloceerd worden. Bovendien kan het geheugen terug vrijgegeven worden eenmaal het niet meer nodig is.

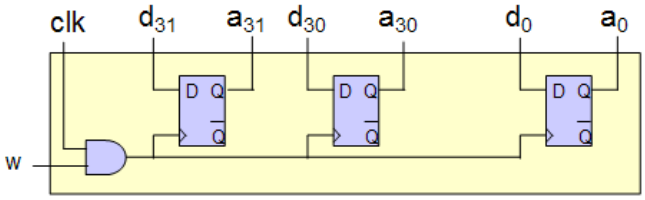
PROCESSORONTWERP

DE KLOK

De processorklok geeft het tempo aan waarmee constrolesignalen worden gegenereerd. De snelheid van de processorklok is altijd een veelvoud van de systeembusklok. De maximaal haalbare snelheid is 4 GHz, bij hogere frequenties is het met de “beperkte” snelheid van het signaal niet meer mogelijk om de hele chip te doorkruisen. Het is bovendien niet zinvol om de de klokperiode kleiner te maken dan 12 FO4 omdat er anders te weinig poortvertragingen kunnen voorkomen.

REGISTERS

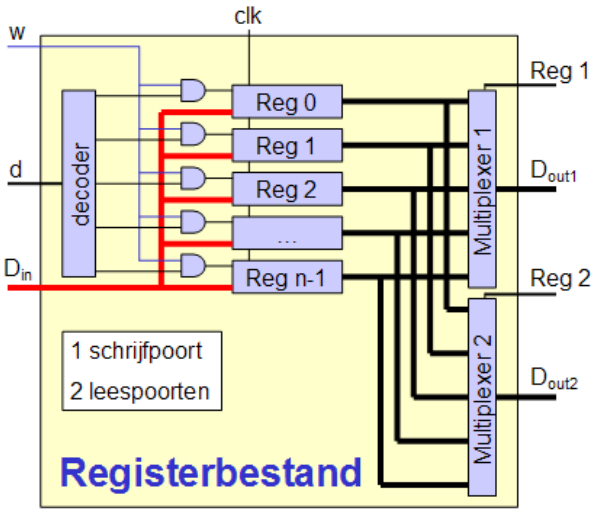
Een register is een zeer snelle vorm van geheugen die gebruikt wordt om gegevens op te slaan in de processor.



Registers zijn opgebouwd uit een evenveel D flip-flops als ze bits moeten opslaan. Ze maken allen gebruik van hetzelfde controlesignaal dat een combinatie is van de klok en een writesignaal.

Dankzij het gebruik van flip-flops kan een register tijdens dezelfde klokcyclus een waarde beschikbaar stellen (opgeslaan in de slave latch) als beschreven worden (in de master latch).

De registers in een processor worden gecombineerd in een registerbestand. Een registerbestand heeft een aantal schrijfpoorten en een aantal leespoorten.



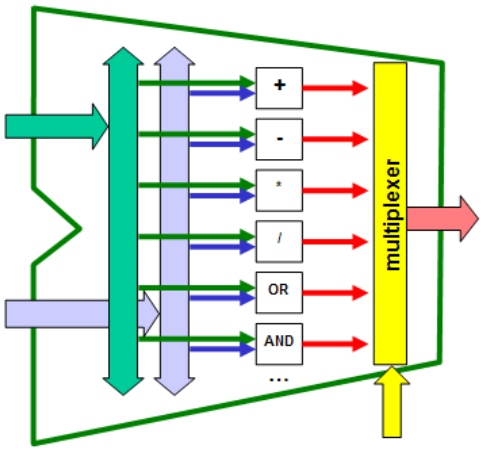
Om te schrijven bepaalt menvia een decoder het register dat men wenst te beschrijven en stelt de waarde beschikbaar via het datakanaal van de schrijfpoort.

Bij het lezen kiest men een register met behulp van een multiplexer en leest de data uit het datakanaal van de leespoort.

Qua performantie zullen bijkomende poorten het registerbestand vertragen, daarom probeert men het aantal poorten zo klein mogelijk te houden.

DE ALU

Essentieel bestaat een ALU uit een aantal inputs, de parallelle berekening van alle functies en op het einde een demultiplexer die het resultaat van de gewenste functie selecteert.



We bespreken nu enkele bouwblokken van de ALU:

OPTELLERS

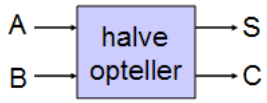
We bouwen een volledige opteller uit een reeks van subcomponenten.

We beginnen met de halve opteller of half adder. Deze berekent de som van 2 bits en stelt het resultaat voor als 2 bits: de som S en de carry C.

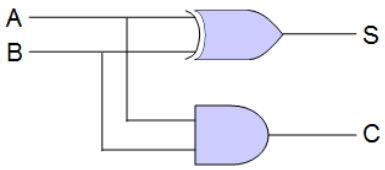
De waarheidstabel voor de half adder is:

O1	O2	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

De logische voorstelling is:



En we kunnen hem implementeren als volgt:



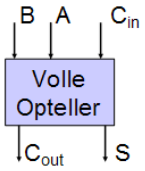
We zien duidelijk dat dit een eenvoudige bewerking is en zeer snel kan uitgevoerd worden. Er is slechts 1 poortvertraging.

Als uitbreiding op de half adder, bekijken we de volle opteller of full adder. Deze maakt de som van 3 bits: 2 operandi en 1 inkomende carry bit. Het resultaat wordt opnieuw voorgesteld als 2 bits: de som S en de carry C.

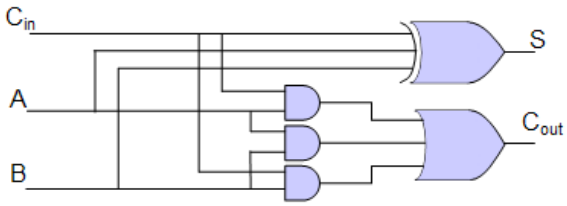
De waarheidstabel voor de full adder is:

C_{in}	O1	O2	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

De logische voorstelling is:



En we kunnen hem implementeren als volgt:

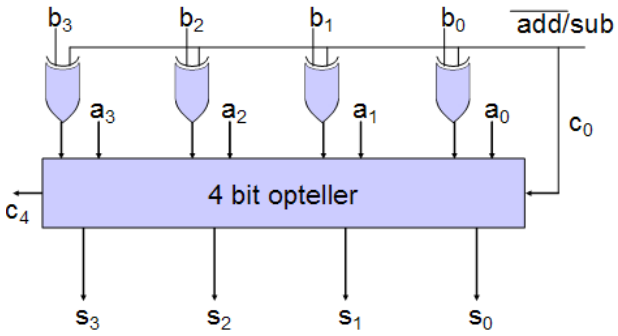


De full adder is complexer dan de half adder aangezien er meer poorten, poortvertragingen en inputs zijn. De som wordt nu opnieuw na 1 poortvertraging gevonden, maar de carry pas na 2 poortvertragingen.

Nu kunnen we een M full adders in cascade plaatsen om een M-bit optelling te realiseren. Dit heet een doorsijpelopteller of ripple carry adder. Hierbij propageert de carry langzaam doorheen het hele circuit, er treedt dus een significante vertraging op.

Op volledig analoge wijze kunnen we ook een ripple borrow subtractor realiseren. Hierbij heemt men dan geen carry maar een borrow bit die propageert. Deze heeft bovendien dezelfde performantiekenmerken als de ripple carry adder.

We kunnen bovendien een component realiseren dat zowel kan optellen als aftrekken. Beschouw het volgende netwerk:



Wanneer de subtract controlelijn op 1 staat, dan wordt c_0 1 en berekenen de XOR poorten –B (een invertering van alle bits + 1). Dus berekent de opteller A+(-B). Indien de subtract controlelijn op 0 staat, dan is c_0 gelijk aan 0 en blijven de bits van B onverandert. De opteller berekent dan A+B.

De poortvertragingen bij het berekenen van de carry bits zijn een ernstig performantieprobleem. Zeker indien we brede optelling willen doen. Een oplossing hiervoor is de carry lookahead adder. Hierin worden de uitgaande en inkomende carry bits rechtstreeks berekent uit de operandi. De carry bits moeten dus niet meer telkens doorgegeven worden. Hoewel dit een grotere poortdiepte vereist, is het eindresultaat toch sneller dan de ripple carry adder, zeker bij bredere operandi.

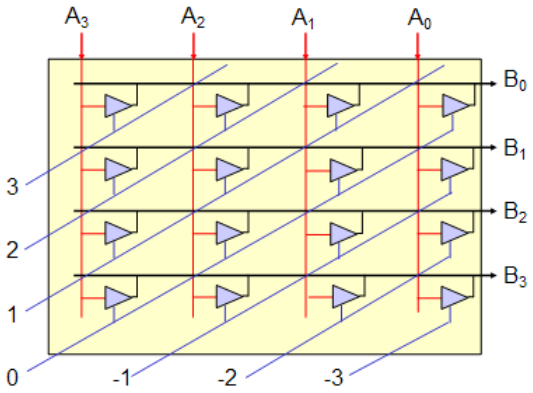
Omwille van technologische fan-in beperkingen kunnen we maar carry lookahead adders realiseren tot 8 bits. Voor optellingen van bredere operandi kan men de carry lookahead adders in cascade plaatsen.

VERSCHUIVERS

We bekijken twee methodes om een bitpatroon te verschuiven.

BARREL SHIFTER

Een barrel shifter kan in 1 poortvertraging een bitpatroon over n willekeurige bits voorwaarts of achterwaarts verschuiven. Men heft hiervoor een matrix van tri-state buffers nodig:



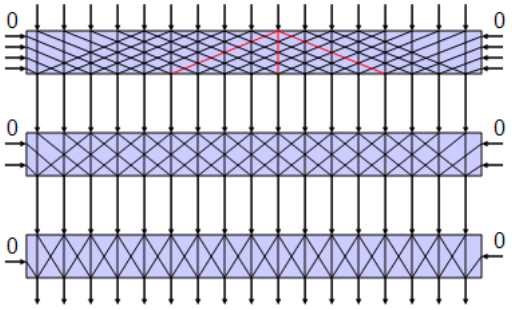
Om een correcte verschuiving uit voeren is nog extra logica voor het links en rechts inschuiven van bits nodig. Dit verandert echter niets aan het concept.

Deze methode is prima voor het verschuiven van kleine bitpatronen, maar voor grotere loopt het aantal buffers en daarmee ook de fan-out van A sterk op.

Bovendien heeft de barrel shifter een verborgen kost. Het genereren van de -n tot n signalen zal een decoder vereisen, deze zal ook een aantal poortvertragingen veroorzaken.

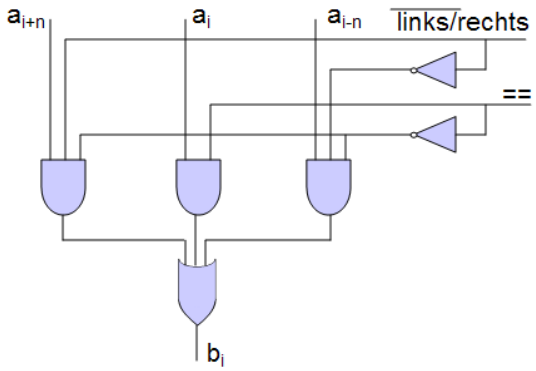
LOGARITMISCHE SHIFTER

Een logaritmische shifter kan een word verschuiven over een willekeurige afstand afhankelijk van de controle-ingangen.



De shifter splitst de verschuiving op in verschillende stappen waarbij elke stap een ander bit van de schuifafstand bekijkt. Elke stap verschuift dus over een macht van 2 posities.

Een cel van de logaritmische shifter ziet er als volgt uit:

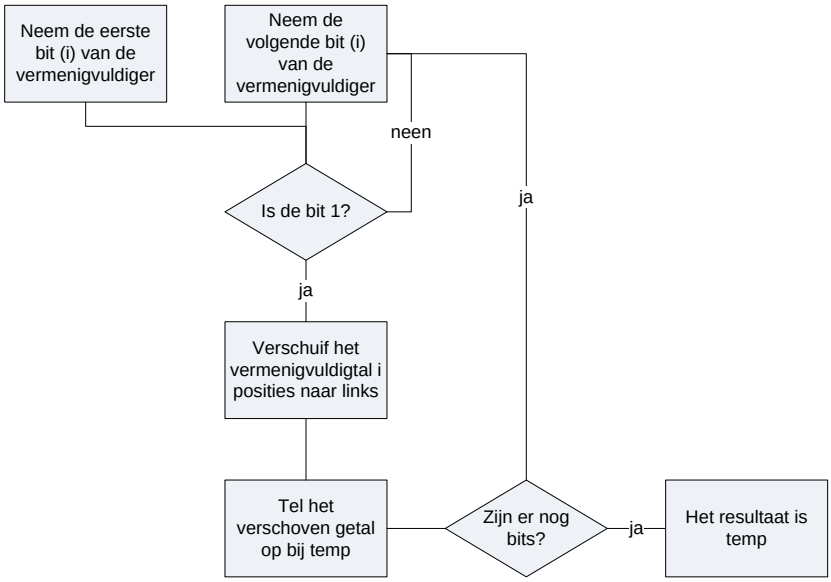


Deze cel vervangt de bit a_i door de bit a_{i+n} of a_{i-n} afhankelijk van de links/rechts controle-input. Indien het == controlesignaal aan staat wordt de bit niet vervangen en treedt er geen verschuiving op. Deze cel wordt op elk niveau herhaalt voor elke bit.

VERMENIGVULDIGERS

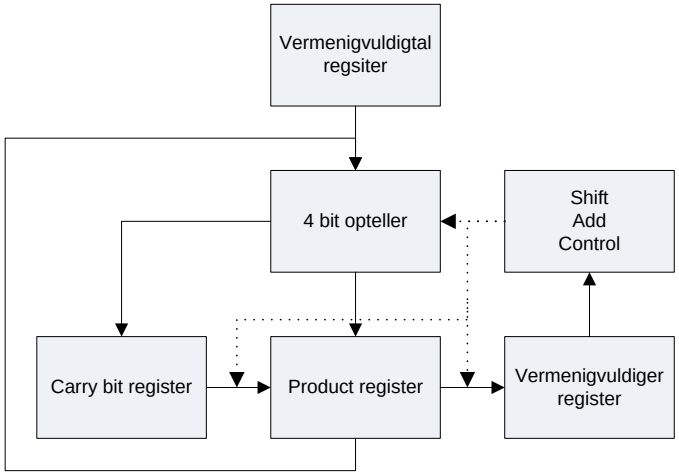
SERIËLE VERMENIGVULDIGER

Een binaire vermenigvuldiging wordt als volgt uitgevoerd:



Het product van 2 getallen van n bits, kan maximaal een 2n bit getal opleveren.

Een hardware-implementatie van een gehele vermenigvuldiging werkt op analoge wijze. De structuur van de implementatie is als volgt:



We beschrijven stap per stap de methode om het product te bekomen:

1. Plaats vermenigvuldigtal en vermenigvuldiger in de juiste registers en zet de andere regiseters op 0
2. Voer n keer uit:
    a. Als de laatste bit in het vermenigvuldiger register 1 is, tel dan de inputs van de opteller op en sla het resultaat op in het product register, ga anders naar b
    b. Schuif het carry bit, product en vermenigvuldiger register 1 bit naar rechts
3. Men vindt het resultaat in de combinatie van het product en het vermenigvuldiger register

Dit algorimte is analoog aan het algoritme voor binaire vermenigvuldiging.

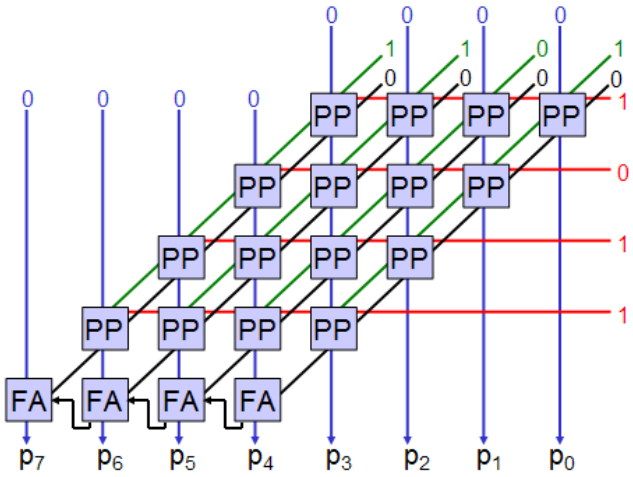
Soms kan men de operatie versnellen door het toepassen van het algoritme van Booth. Het idee is dat indien de vermenigvuldiger bestaat uit een opeenvolging van 1 bits, deze gegroepeerd kunnen worden als een verschil. Concreet zal men bij het uitvoeren de laatste bit die men heeft weggeschoven uit de vermenigvuldiger bijhouden. Bij een overgang van 0 naar 1 begint een rij van 1 bits, op dat moment moet men het verschil maken in plaats van de som. Bij een overgang van 1 naar 0 maakt men de som. Er vanuit gaande dat enkel positieve getallen vermenigvuldigd worden moet men beginnen met een 0 en een leidende 0 toevoegen.

De seriële implementatie die we hier besproken hebben vereist weinig hardware, maar is erg traag. Het vermenigvuldigen van twee n bit getallen is van complexiteit $O(n^2)$. Door het proces te parallelliseren kan men de berekening in 2n stappen uitvoeren.

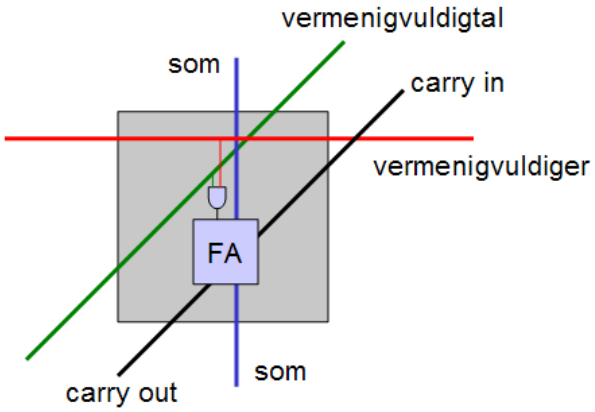
CARRY SAVE VERMENIGVULDIGER

In deze vermenigvuldiger maken we 1 bit producten van elke bit in het vermenigvuldigtal met elke bit in de vermenigvuldiger waarna we de partiële producten in elke rij sommeren.

De realisatie ziet er als volgt uit:



De partiële productelementen zien er als volgt uit:

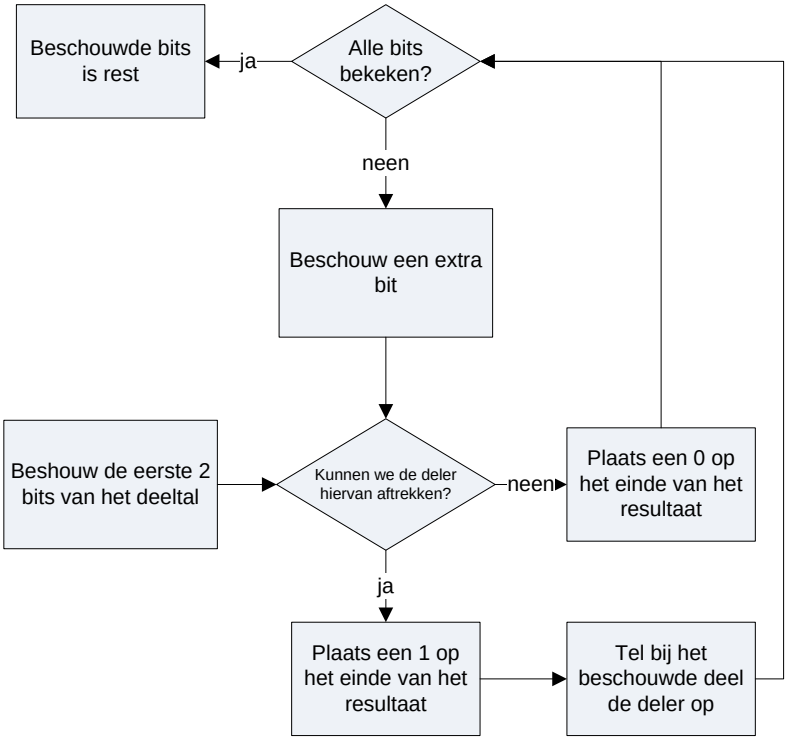


De rijen van partiële producten komen overeen met de producttermen in de seriële methode. De producttermen kan men vinden als de uitgangen van de AND poorten in de partiële producten. De full adders in de partiële producten tellen in cascade de verschillende producttermen op. De onderste rij full adders maakt de cascade optelling compleet.

Qua performantie moet men rekenen op 2 poortvertragingen per extra bit. Hoewel dit al veel beter is dan de seriële methode, kan men dit nog versnellen door het carry lookahead principe toe te passen.

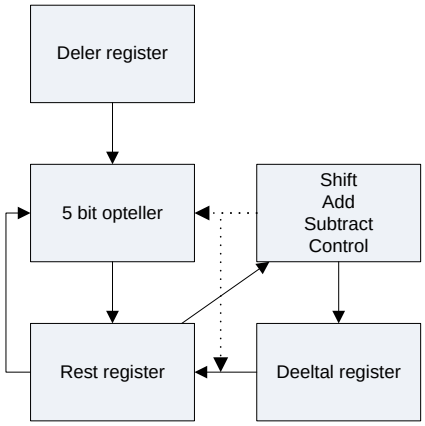
DELERS

Een deling van positieve binaire getallen verloopt analoog als hoe we handmatig een deling uitrekenen:



RESTORING DELING

Dit is een seriële implementatie van een gehele deling, gelijkaardig aan de manuele methode. De structuur is als volgt:



Het algoritme gaat als volgt:

1. Initialiseer de registers met deler, deeltal en de andere 0
2. Voer n keer uit (n is het aantal bits in het deeltal):
    a. Schuif rest en deeltal 1 positie naar links
    b. Bereken het verschil tussen rest en deler
    c. Schrijf in de laatste bit van het deeltal een 0 als het verschil negatief is en een 1 als het verschil positief is
    d. Tel de deler terug op bij de rest, de rest is nu hersteld
3. We vinden de rest in het rest register en het quotiënt in het deeltal register

Als aanpassing op dit algoritme kunnen we indien we merken dat het verschil tussen rest en deler negatief is stap 2.d weglaten en bij de volgende iteratie in 2.b de som nemen in plaats van het verschil. Dit heet de non-restoring deling.

ITERATIEVE DELING

Met behulp van de iteratiemethode voor het bepalen van wortels van veeltermen (Newton) kunnnen we de deling realiseren met enkel de aftrekking en vermenigvuldiging:

$$x_{i+1} = x_i(2 - x_i b)$$

Waarbij b het deeltal is.

Met behulp van lookup-tabellen die reeds 1 bit correct zetten, heeft men 32 bit precisie na 5 iteraties.

HET DATAPAD

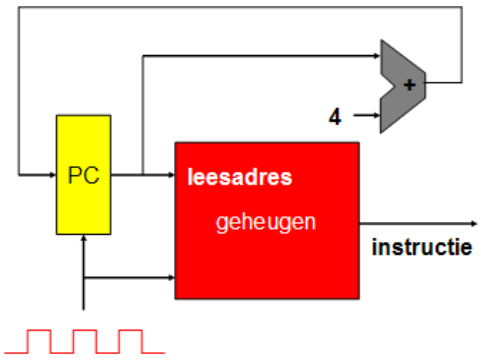
Dit is een van de essentiële onderdelen van de kern van een processor. Het datapad omvat de registers en de ALU. We modelleren een datapad aan de hand van 3 soorten instructies:

ALU-instructies
Geheugeninstructies
Controletransferinstructies

Deze 3 soorten instructies staan model voor alle andere instructies die men in moderne processoren aantreft.

INSTRUCTIES INLADEN

De volgende schakeling zal per klokpuls een instructie ophalen uit het geheugen:



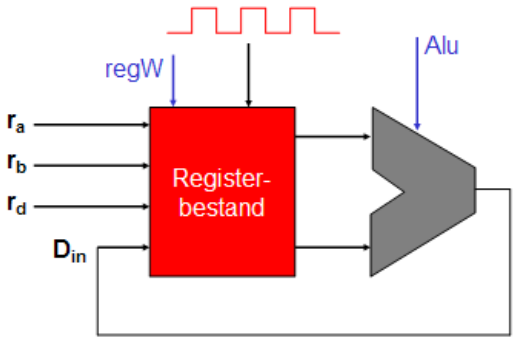
Op de dalende flank van de klok krijgt de program counter een nieuwe waarde die hij meteen doorstuurt naar het geheugen waarop dit als output de aangevraagde instructie levert. De nieuwe waarde van de program counter wordt bovendien naar een ALU gestuurd die er 4 (de veronderstelde lengte van een instructie) bij optelt en terugstuurd naar de program counter die de nieuwe waarde bij de volgende dalende klokflank zal gebruiken. Dit scenario blijft zich herhalen.

ALU-INSTRUCTIES

RRR

Bij het decoderen van een ALU-instructie worden de volgende gegevens geëxtraheerd:

Code voor de ALU-instructie
2 bronregistercodes
1 doelregistercode



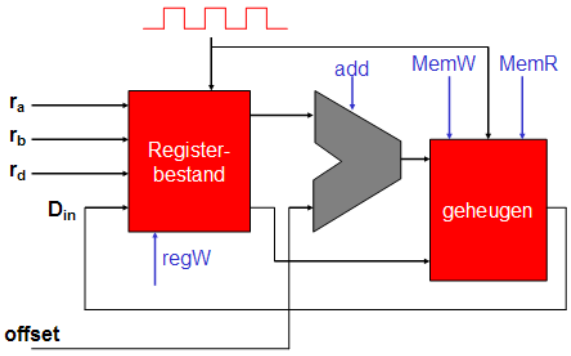
De bronregistercodes worden aangelegd aan leespoorten van het registerbestand en de doelregistercode aan een schrijfpoort. De outputs van de leespoorten gaan naar de ALU die met behulp van de instructiecode de operatie herkent. De output van de ALU wordt teruggemapped naar de input van de schrijfpoort van het doelregister.

Merk op dat bron- en doelregisters gelijk mogen zijn aangezien de flip-flops tegelijk waarden kunnen lezen en schrijven.

RRI

Analoog aan de RRR operatie, maar in plaats van een tweede bronregister op te vragen, wordt een constant rechtstreeks aan de input van de ALU gekoppeld.

GEHEUGENINSTRUCTIES

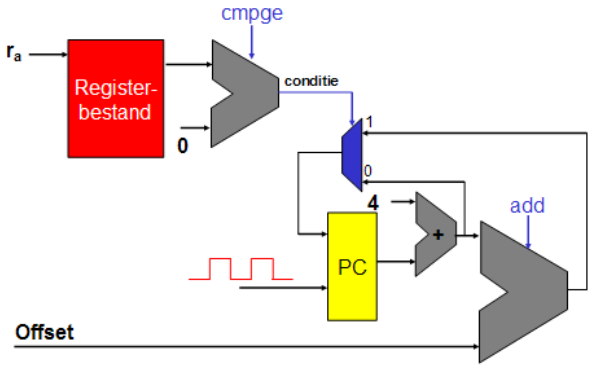


Geheugeninstructies gebruiken de ALU om adresexpressies uit te rekenen. We gaan hier uit van een basisadressering waarbij een constante offset bij de waarde van een register geteld wordt. Het resultaat van de adresberekening wordt aangelegd aan het geheugen.

Afhankelijk of het om een lees- of schrijfoperatie gaat, krijgt het geheugen een memory read of memory write signaal. Bij het lezen wordt de geheugenwaarde gekoppeld aan een leespoort van het registerbestand en bij het schrijven wordt de geheugeningang gekoppeld aan de leespoort.

CONTROLETRANSFERINSTRUCTIES

We beschouwen een controletransferinstructie die eerst een sprongvoorwaarde controleert en daarna indien nodig de adresexpressie uitrekent.



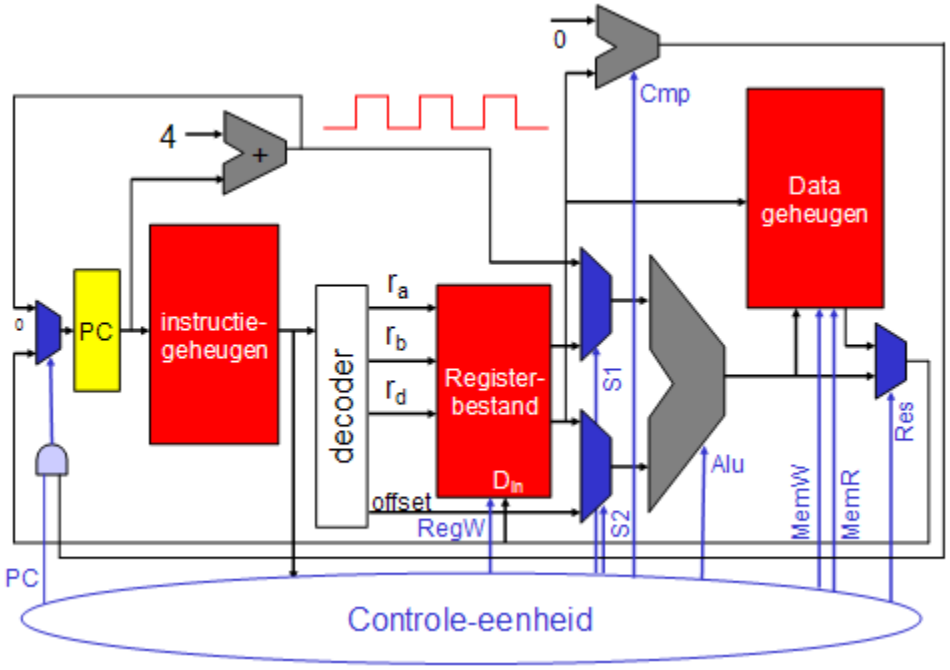
In het bovenstaande voorbeeld is de voorwaarde dat r1 groter is dan 0, in het andere geval gaat de processor verder met het doorvalpad. Als we de sprong nemen, dan wordt een offset opgeteld bij de program counter.

Aangezien de program counter nu ofwel +4 of +4+offset kan zijn, hebben we een multiplexer die we met behulp van de conditie aansturen.

Bij een onconditionele sprong kunnen we gewoon de conditie weglaten ofwel een conditie evalueren die altijd waar is.

EEN CYCLUS-PER-INSTRUCTIEMACHINE

We proberen nu de datapaden voor ALU-instructies, geheugeninstructies en controletransferinstructies samen te voegen tot 1 datapad dat per klokcyclus 1 instructie uitvoert.



Op verschillende plekken zijn multiplexers toegevoegd om te kiezen uit meerdere inputsignalen. Ook is er nu een decoder die de registeroperandi en offset uit de instructie haalt. Ten slotte is er ook een controle-eenheid die aangestuurd door de instructie de controlesignalen naar alle componenten stuurt.

Qua performantie is de een cyclus-per-instructiemachine niet zo interessant. Ze kan niet sneller geklokt worden dan de traagste instructie en sommige componenten zoals de ALU moeten dubbel uitgevoerd worden.

CONTROLE-EENHEID

De controle-eenheid maakt gebruik van een tabel met de verschillende controlepunten. Elke instructie is een rij in de tabel en heeft voor elk controlesignaal een bijhorende waarde.

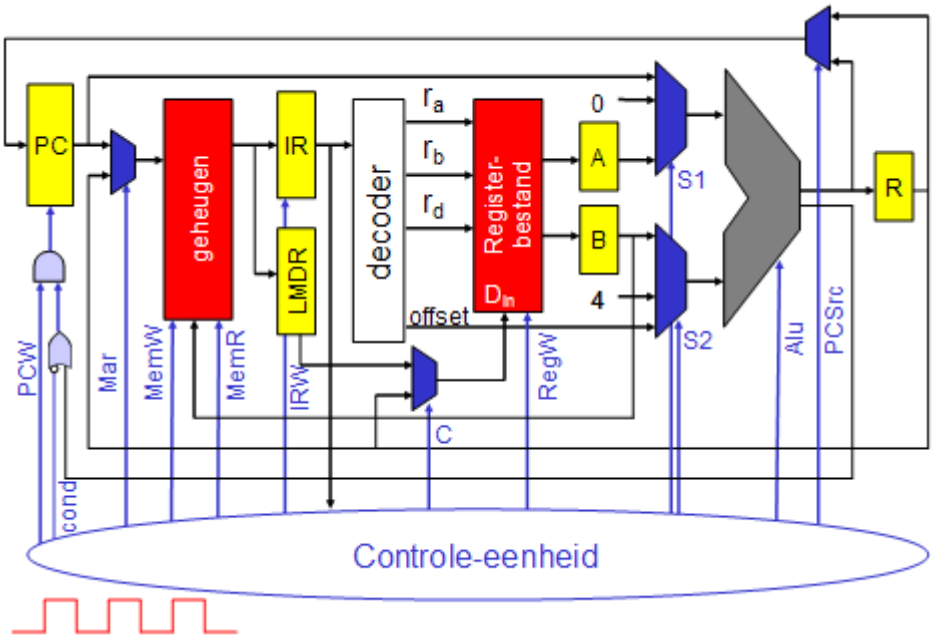
Bijvoorbeeld:

	PC	RegW	S1	S2	Alu	Cmp	MemW	MemR	Res
add	0	1	1	0	001	xxx	0	0	1
addi	0	1	1	1	001	xxx	0	0	1
load	0	1	1	1	001	xxx	0	1	0
store	0	0	1	1	001	xxx	1	0	x
jump	1	0	0	1	001	111	0	0	1
brge	1	0	0	1	001	110	0	0	1

Men kan deze tabel eenvoudig implementeren aan de hand van een geheugen met BAU-cellen. Elke instructie heeft dan een bepaalde index in de tabel en men kan eenvoudigweg de controlesignalen verbinden met de individuele bits in de geselecteerde geheugencel.

MEER CYCLI-PER-INSTRUCTIEMACHINE

Een meer cycly-per-instructiemachine kan als volgt gerealiseerd worden:



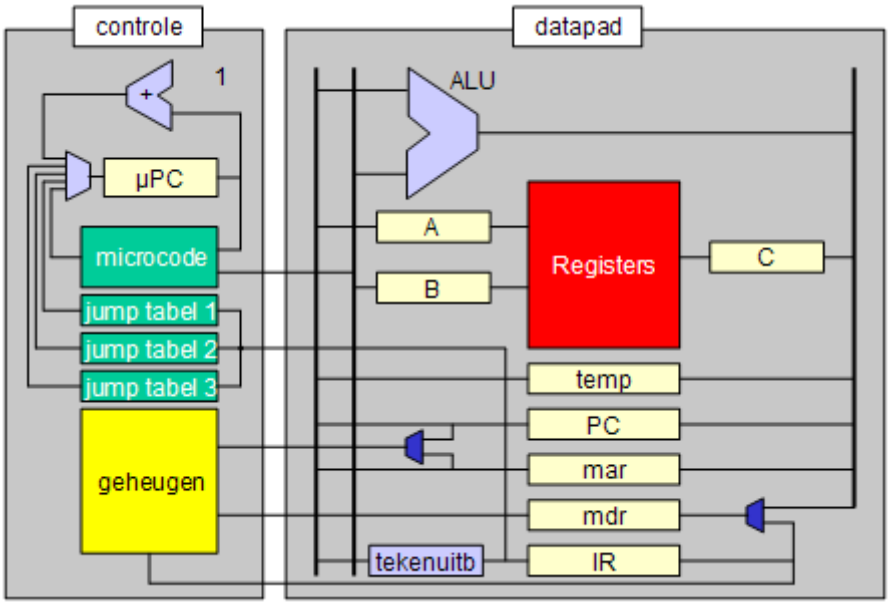
Hier komt elk component maar eenmaal voor. Bijgevolg zullen instructies opgedeeld moeten worden in fasen die sequentieel na elkaar uitgevoerd worden.

CONTROLE-EENHEID

De controle-eenheid bevat nu een toestandstabel die meerdere toestanden per instructive bevat. De controle-eenheid is een toestandsmachine die afhankelijk van de instructie en de huidige toestand elke cyclus een nieuwe toestand bepaalt met de corresponderende controlesignalen.

Men kan de informatie volledig coderen in ROM-geheugen: een controletabel met de controlesignalen voor de toestand en een toestandstabel met de volgende toestand. Om de grote toestandstabel te vermijden kunnen we gebruik maken van een sequencer en ervan uitgaan dat de volgende toestand de volgende rij in de tabel is. Een signaal uit de huidige toestand kan dan via een multiplexer een ander gedrag selecteren, bijvoorbeeld het bepalen van de volgende toestand uit een kleine toestandstabel.

Een andere mogelijkheid is de volgende toestand mee te coderen in de controletabel. We kunnen dan van eender welke toestand naar eender welke andere toestand springen. Indien we dit model gebruiken, kunnen we het datapad iets anders voorstellen:



Men noemt de controletabel nu het microcodeprogramma en de toestandsveranderlijke de microcode-PC. Hieronder ziet u een voorbeeld van een microcodeprogramma:

µPC	Label	ALU	S1	S2	Dest	ExtIR	Const	JCond	Adr	Mem	MAdr	MDest	Regs
0000	Fetch							Mbusy	Fetch	RW	PC	IR	
0001		ADD	PC	Const	PC		4	Jump1					RR
0002	LdSt	ADD	A	IR	MAR	Word		Jump2					
0003	Brge	S1	A					LT	Fetch				
0004	Jump	ADD	PC	IR	PC	Word		True	Fetch				
0005	Add	ADD	A	B	C								
0006								True	Fetch				WF3
0007	Addi	ADD	A	IR	C	Word		True	Wb2				
0008	Load							Mbusy	Load	RW	MAR	MDR	
0009		S1	MDR		C								
000A	Wb2							True	Fetch				WF2
000B	Store	S2		B	MDR								
000C	Store2							Mbusy	Store2	WW	MAR		
000D								True	Fetch				

Opcode	Jump Table 1	Jump Table 2
LD	LdSt	Load
ST	LdSt	Store
ADD	Add	
ADDI	Addi	
BRGE	Brge	
JUMP	Jump	

De controle-eenheid moet bovendien rekening houden met eventuele excepties en onderbrekingen.

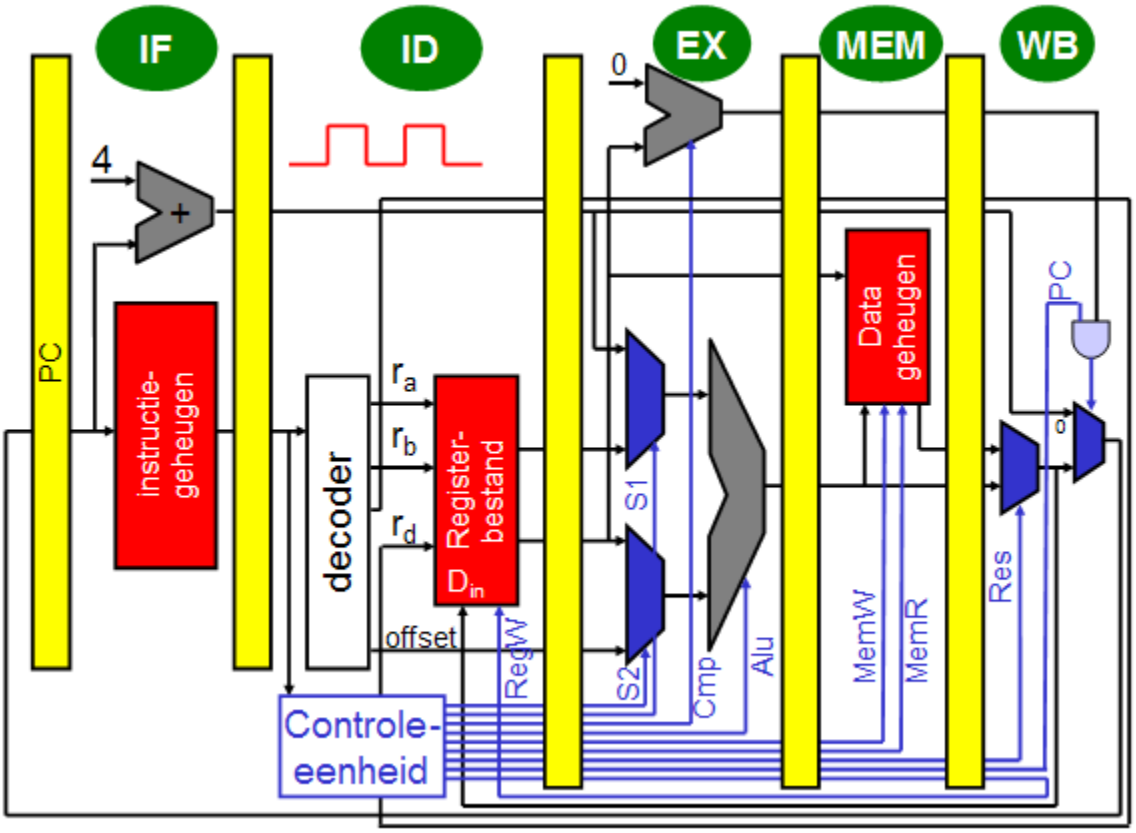
GEPIJPLIJNDE MACHINES

SEQUENTIEEL GEPIJPLIJNDE MACHINE

We kunnen bij een meer cycli-per-instructie machine 5 types toestanden definiëren:

1. Instruction Fetch: het ophalen van de instructie
2. Instruction Decode: het decoderen van de instructie
3. Execute: berekeningen uitvoeren met de ALU
4. Memory: geheugenoperaties uitvoeren
5. Write Back: eventuele resultaten wegschrijven

Willen we deze vijf toestanden in trappen uitvoeren, waar op elk moment in elke trap een instructie zit, dan moeten we extra registers toeveoegen om de toestand te bewaren tussen de trappen. Slaan we in de registers ook de toestandssignalen op, dan kunnen we de architectuur als volgt voorstellen:



Met deze opstelling kunnen we eender welke instructie in 5 trappen uitvoeren. Alle benodigde informatie wordt elke klokcyclus doorgestuurd naar het volgende pijplijnregister. Zoals te zien in het diagram propageren we RegW en r_d ook tot in de WB fase.

Bij gepijplijnde architecturen bepaalt de langstdurende trap de maximale klokfrequentie. De volgende instructie wordt pas begonnen wanneer de vorige afgewerkt is. Trappen blijven dus gemiddeld 4 op de 5 cycli ongebruikt.

PARALLEL GEPIJPLIJNDE MACHINE

Men kan het lage gebruik van elke trap bij de sequentieel gepijplijnde machine vermijden door meerdere instructies parallel na elkaar te verwerken. Tijdens het decoderen van een instructie kan de tweede al ingeladen worden, enzovoort. De enige aanpassing die men hiervoor moet maken is het uitrekenen van de nieuwe PC in de Execute fase te laten doorgaan en het resultaat meteen in het PC register op te slaan.

De principes van parallel gepijplijnde machines zijn:

- Een nieuwe instructie per klokcyclus
- Elke instructie vraagt vijf klokcycli
- Vijf instructies zitten tegelijk in verschillende stadia van uitvoering
- De klokfrequentie is afhankelijk van de trapvertraging
- Men kan de klokfrequentie verhogen door meer trappen in te voeren
- Het aantal instructies per cyclus is over het algemeen kleiner dan 1

Dit levert evenwel problemen op bij controletransferinstructies waarbij men niet weet of de komende instructies wel uitgevoerd moeten worden. Men kan dit oplossen door na controletransferfuncties 2 NOP instructies in te voegen. Deze instructies zullen de pijplijn vullen totdat de bestemming van de sprong bekend is.

Merk op dat de totale tijd voor het uitvoeren van een instructie niet verandert. Men kan echter wel 5 keer meer instructies uitvoeren in dezelfde tijdspanne.

HAZARDS

Hazards zijn afwijkingen van de standard sequentiële interpretatie van de programmatekst. Ze worden veroorzaakt door de overlappende uitvoering van instructies die na elkaar komen.

STRUCTURELE HAZARDS

Deze hazards worden meestal veroorzaakt door schaarse hardware. Stel bijvoorbeeld dat het registerbestand niet simultaan lees- en schrijfbaar zou zijn, dan kan er een conflict ontstaan tussen de Write Back fase en de Instruction Decode fase. In dat geval zou een volgorde moeten afgesproken worden voor toegang tot het registerbestand.

Structurele hazards kunnen vermeden worden door het ontdubbelen van de hardware zodat meerdere operaties tegelijk uitgevoerd kunnen worden.

CONTROLEHAZARDS

Deze hazards worden vooral veroorzaakt door de vertraging waarmee pas bekend wordt of een sprong al dan niet genomen wordt. Zolang de bestemming onbekend is, kan de processor niet verder.

Er zijn een aantal mogelijke oplossingen:

> Verander de instructies door NOP's indien de sprong genomen moet worden, dit is net op tijd om verandering van registers te voorkomen, de omgezette instructies noemen we pijplijnbellen
> Voer de instructies gewoon verder uit, dit verschuift het probleem naar de programmeur die nu rekening moet houden met het feit dat de 2 instructies na een sprong altijd uitgevoerd worden, dit heet vertraagde controletransfer
> Voorspel de sprong, indien later de gok fout blijkt te zijn kan men van de instructies NOP's maken

Het gevaar kan bovendien verkleint worden door de berekening van de nieuwe PC te verhuizen naar de Instruction Decode stap. Daar is de PC al van bij het begin beschikbaar en de offset kort erna. De comparator die de voorwaardelijke sprong berekent moet echter wel wachten op het registerbestand. Meestal lukt dit echter en kan men het aantal probleeminstructies terugbrengen naar 1.

Moest het voorgaande niet lukken kan men nog altijd proberen om de sprong te voorspellen. Een sprongvoorspeller probeert uitgaande van het voorbije gedrag van het programma te voorspellen wat de volgende in te laden instructie zal zijn. Er zijn veel verschillende soorten en kunnen meer dan 95% nauwkeurigheid halen bij realistische programma's.

Een sprongvoorspeller voorspelt in de Instruction Fetch fase of een sprong al dan niet genomen zal worden. De generatie van het adres waarnaar moet gesprongen worden gebeurt in de Instruction Decode trap. Bij een machine met vijf trappen levert dit geen voordelen, maar bij andere machines waar de uitkomst pas in de vijfde trap bekend zou zijn levert dit een winst op. Aangezien men zelfs met een voorspeller pas in de

Instruction Decode fase de sprong kan voorspellen, heeft men nog 1 instructie over die men kan gebruiken voor vertraagde controletransfer.

Door in de sprongtabellen ook het sprongadres bij te houden kunnen we reeds in de Instruction Fetch fase weten waarnaar gesprongen moet worden.

We bespreken enkele types van sprongvoorspellers:

Type	Methode	Uitleg
Statisch	Niet-genomen	De normale pijplijn laadt de instructies na de sprong in, dit komt neer op het voorspellen dat de sprong niet genomen zal worden. Onvoorwaardelijke sprongen moeten natuurlijk wél als genomen voorspeld worden. Men kan van dit gedrag gebruik maken door in het programma de voorwaardelijke sprong in het begin te plaatsen.
	Backward taken	Hier is de voorspelling afhankelijk van de richting van de sprong. We kunnen van deze methode gebruik maken in code door de voorwaarde achteraan te plaatsen.
Dynamisch	1 bit voorspeller	Houdt in een tabel bij of de huidige sprong bij de vorige uitvoering uitgevoerd werd of niet. Indien de voorspelling fout blijkt te zijn wordt de sprongtabel aangepast. Nadelen: De eerste en de laatste uitvoering van een lus worden fout voorspeld De sprongtabel is kleiner dan het programma, sprongen zullen elkaars informatie dus overschrijven
	2 bit voorspeller	Het nadeel dat de laatste sprong fout voorspeld wordt bij de 1 bit voorspeller kan opgevangen worden door meerdere bits bij te houden. Uit de praktijk blijkt 2 bit te volstaan. Elke keer een sprong genomen wordt, verhoogt de teller met 1 (tot de maximale waarde) en in het andere geval verlaagt de teller met 1 (tot de minimale waarde). Indien de teller op 00 of 01 staat, voorspelt men dat de sprong niet genomen wordt en indien de teller op 10 of 11 staat voorspelt men dat de sprong wél genomen wordt.
	Lokaal	Hierbij baseren we de voorspelling op de geschiedenis van uitkomsten. Per uitgevoerde sprong wordt in het schuifregister een bit bijgeschoven. Met dit patroon wordt een element uit een tweede tabel, de sprongtabel, opgevraagd dat de voorspelde waarde geeft voor dit patroon.
	Globaal	Analoog aan de lokale voorspeller, maar in plaats van een geschiedenistabel hebben we nu één enkel veld voor de geschiedenis. Om te vermijden dat verschillende programma's elkaars geschiedenis verstoren, kan men de geschiedenis XOR'en met een aantal bits van de PC.

	Brang Target Buffer	Indien we reeds in de Instruction Fetch fase willen weten waarnaar de sprong zal springen, moeten we ook het doeladres bijhouden. We hebben dus een tabel nodig met de volgende gegevens: PC van de sprong Doeladres van de sprong Een aantal voorspelbits Sprongen die voorspeld worden om niet door te gaan moeten niet opgenomen worden in de tabel.

DATAHAZARDS

Worden veroorzaakt door het feit dat de resultaten pas in de vijfde cyclus weggeschreven worden, maar dat andere instructies ondertussen al van deze waarden gebruik hadden willen maken.

Wanneer 2 opeenvolgende instructies hetzelfde object lezen of schrijven zijn er 4 mogelijke afhankelijkheden:

Read after Write: de eerste instructie schrijft en de tweede leest, de leesoperatie moet wachten op het voltooien van de schrijfoperatie
Write after Read: de eerste instructie leest en de tweede schrijft, de schrijfoperatie moet wachten op het voltooien van de leesoperatie maar kan evengoed zijn resultaat elders kunnen wegschrijven
Write after Write: de eerste instructie schrijft en de tweede ook, de tweede moet wachten op het voltooien van de eerste maar kan evengoed zijn resultaat elders wegschrijven
Read after Read: de eerste instructie leest en de tweede ook, de operaties mogen in een willekeurige volgorde uitgevoerd worden maar moeten op elkaar wachten

Er zijn drie mogelijke oplossingen:

Plaats de instructies die dezelfde gegevens gebruiken ver genoeg uit elkaar
Blokkeer de pijplijn bij het optreden van een datahazard
Gebruik forwarding, deze techniek maakt gebruik van waarden die al berekent zijn in verdere stappen maar nog nie weggeschreven zijn in het registerbestand, dit kan vanuit de Execute en de Memory fases

ARCHITECTUUREVOLUTIE

RISC VERSUS CISC

In de jaren 70 was de CISC (Complex Instruction Set Computer) zeer populair. Deze microgecodeerde instructies waren echter zeer complex en de compilers gebruikten maar een klein aantal van de instructies. Daarom werd de RISC ontwikkeld (Reduced Instruction Set Computer).

Vele studies toonden aan dat het grootste deel van de gebruikte instructies vrij eenvoudig was en nieuwe complexe instructies eigenlijk overbodig waren. Het is duidelijk dat men de snelheid van de computer meer kan verbeteren door zich te focussen op de meest uitgevoerde instructies, dan op de complexe zeldzame instructies.

Men kan een RISC architectuur herkennen aan:

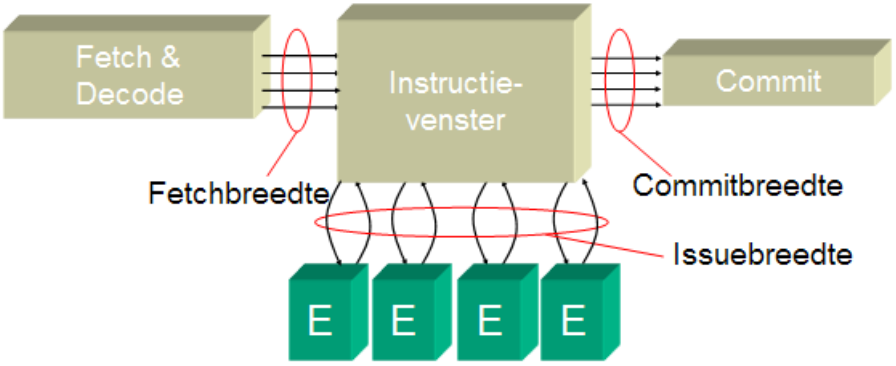
Voornamelijk noodzakelijke instructies

Eenvoudig decodeerbare instructies: instructies hebben een vaste lengte en zijn gealigneerd
Load/store architectuur met veel registers
Geen complexe adresseermodes

Intel processoren vertonen nog een aantal CISC sporen, maar hebben intern heel veel RISC kenmerken.

SUPERSCALAIRE ARCHITECTUREN

Indien we per klokcyclus slechts 1 instructie inladen, blijft het aantal instructies per klokcyclus kleiner dan 1. Om meerdere instructies per cyclus uit te voeren, kunnen we meerdere instructies inladen en uitvoeren in dezelfde trap. We moeten dan wel toezien dat de sequentiële semantiek behouden blijft en rekening houden met de afhankelijkheden.



Bij superscalaire architecturen laadt en decodeert men asynchroon instructies die in het instructievenster komen. Daar wachten instructies tot al hun argumenten beschikbaar zijn en voeren dan uit op een van de beschikbare eenheden. Deze uitvoering kan out of order gebeuren. Na de uitvoering verplaatsen de instructies naar de commit trap waarbij ze de processor terug in volgorde verlaten en hun resultaten wegschrijven.

VLIW PROCESSORS

In VLIW of Very Long Instruction Word processors bevat elke instructive een aantal operaties die parallel uitgevoerd kunnen worden ofwel NOP's.

Een voorbeeld is de Philips Trimedia processor.

EPIC PROCESSORS

EPIC of Explicitly Parallel Instruction Computing processors krijgen operaties in bundles waarbij een aantal template bits aangeven welke daarvan parallel uitgevoerd kunnen worden. Bovendien bevat EPIC de mogelijkheid om instructies conditioneel uit te voeren zonder sprongen.

Een voorbeeld is de Itanium processor van Intel.

EMBEDDED SYSTEMS

De personal computers beslaan maar een klein deel van de markt. Het overgrote deel van de verkochte processors zijn 4 of 8 bit architecturen. Deze worden gebruikt in embedded systems of ingebedde systemen.

Processoren worden tegenwoordig in enorm veel apparaten gebruikt, waarbij we evolueeerden van mainframes naar kleine handheld devices. Performantie is hier lang niet het enige criterium, ook prijs, autonomie, grootte, gewicht en functionaliteit zijn belangrijke factoren.

Bij embedded systems is een relatief kleine, eenvoudige processorchip ingebouwd in het apparaat. Hij voert steeds hetzelfde programma uit dat vaak met beperkte hardware in real-time resultaten moet leveren.

De vereisten waaraan een ingebed systeem moet voldoen zijn meestal niet mals:

Draagbaar en dus klein en licht
Grote autonomie
Bestand tegen vocht, hitte, vorst, schokken, straling, …
Real-time
Betrouwbaar
Zeer goedkoop
Produceerbaar in grote volumes

Er worden ook vaak DSP of Digital Signal Processors gebruikt. Deze processoren zijn gespecialiseerd in het verwerken van digitale signalen zoals audio, video, … Ze kunnen vaak zeer complexe transformaties uitvoeren. Door de grote markt hiervoor is het interessant om hier gespecialiseerde processoren voor te bouwen.

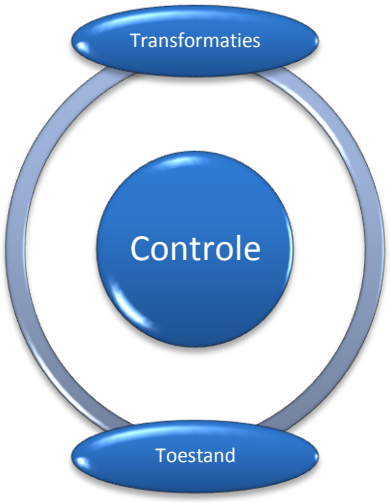
APPENDIX A: DEFINITIES, AFKORTINGEN EN BELANGRIJKE WOORDEN

AGP

Advanced Graphics Port. Een standaard om videokaarten aan te sluiten op de computer. Meer informatie is te vinden bij moderne computers.

ALGORITME

Een algoritme is een sluitende beschrijving om via een opeenvolging van transformaties van gegevens een eindresultaat te berekenen.



Startende vanuit een gekende verzameling begingegevens, worden een aantal transformaties uitgevoerd. De controle bepaalt – volgens het algorimte – welke transformatie moet uitgevoerd worden en houdt bij waar we ons bevinden in het algoritme.

ALU

Arithmetic and Logic Unit, ook wel de functionele eenheid of het operatorgedeelte genoemd. Dit is de conceptuele eenheid die effectieve berekeningen uitvoert.

BESTURINGSSYSTEEM

De “manager” van het computersysteem. Biedt interfaces aan voor toegang tot randapparaten en andere bronnen.

BINAIRE COMPATIBILITEIT

Wanneer verschillende machines dezelfde machinetaal ondersteunen en dus zonder hercompileren programma’s kunnen uitvoeren geschreven in die machinetaal.

Bijvoorbeeld de x86 reeks of de IBM 360 reeks.

BRONCODECOMPATIBILITEIT

Wanneer voor een hoog-niveauprogrammeertaal compilers bestaan in verschillende machinetalen, dan is de programmeertaal broncodecompatibel tussen deze machinetalen.

COMPUTER

Deze term is afkomstig van het Engelse woord voor rekenaar, iemand die berekeningen uitvoert.

DANGLING POINTER

Een niet-geïnitialiseerde pointer verwijst naar een random geheugenlocatie. Dit is een dangling pointer.

GEHEUGENCEL

Een geheugencel is een collective van logische poorten die een stabiele ingang kan bewaren zonder dat de ingangen actief moeten zijn. De uitgang van een geheugencel wordt bepaald door de huidige ingangen en de geschiedenis van ingangen. Een geheugencel kan gebruikt worden om één enkele informatiebit op te slaan en dient als bouwblok voor computergeheugen.

IDE

Integrated Drive Electronics. Een standaard om harde schijven en optische drives aan de computer te koppelen. Meer informatie is te vinden bij moderne computers.

INTERPRETERS

Dit zijn programma’s die runtime de vertaling uitvoeren tussen een hoog-niveauprogrammeertaal en machinetaal. Het programma wordt dus elke uitvoering gecompileert.

ISA

Industry Standard Architecture. Een standaard om randapparatuur aan te sluiten op de computer. De voorloper van PCI. Meer informatie is te vinden bij moderne computers.

LATCH

Een geheugencel van 1 bit.

ONDERBREKING

Onderbrekingen zijn special subroutines. Ze kunnen ofwel aangeroepen worden door de programmeur ofwel door een externe factor. Bijvoorbeeld door de CVE of randapparatuur om een bepaalde situatie op te lossen.

Speciaal aan de oproepen is dat het doeladres via een geheugenindirectie wordt verkregen. Elke onderbreking heeft een nummer en het corresponderende doeladres is opgeslaan in een vectortabel in het geheugen met het nummer als index.

Men gebruikt onderbrekingen om fouten op te vangen of bepaalde diensten te vragen aan het besturingssysteem. Door het updaten van de vectortabel is het eenvoudig om de functionaliteit van het systeem aan te passen.

OPWAARTSE COMPATIBILITEIT

Dit begrip houdt in dat een systeem zo ontworpen wordt dat ook toekomstige versies ervan dezelfde ondersteuning zullen bieden voor de afhankelijke systemen.

OUT-OF-BOUNDS EXCEPTION

Wanneer een index van een array buiten de grenzen van deze array valt, treedt een out-of-bounds exception op.

PCI

Peripheral Component Interconnect. Een standaard voor randapparaten in de computer. Meer informatie is te vinden bij moderne computers.

PCMCIA

Personal Computer Memory Card Industry Association. Een standard om randapparaten aan te sluiten op laptops. Compact, maar ook duurder dan de alternatieven ISA en PCI bij dekstop computers. Heeft een bandbreedte van 50 MB/s.

PLATFORM

De combinatie van hardware, machinetaal en besturingssysteem.

POINTER

Een geheugencel die het adres van een andere geheugencel bevat.

SCSI

Small Computer Systems Interface. Een standaard om harde schijven en optische drives aan de computer te koppelen. Meer informatie is te vinden bij moderne computers. Een alternatief voor IDE, vooral in servers wordt hier vaak de voorkeur aan gegeven.

SEQUENTIËLE LOGICA

Terwijl combinatorische schakelingen inputs omzetten naar outputs, passen sequentiële logica of finite state machines een bepaalde toestand aan aan de hand van inputs en de vorige toestand.

USB

Universal Serial Bus. Een zeer veel gebruikte standard om allerlei soorten externe randapparatuur aan te sluiten op de computer. Bijvoorbeeld invoerapparaten, printers, s

anners, memory sticks, …