

odisee

Processorarchitectuur

Donné, J.

2 EO/ ICT

€ 6,81



Editie 2015-2016

www.odisee.be

Processorarchitectuur

Johan Donné

2^e fase PBA Electronica-ICT

Inhoudsopgave

Hoofdstuk 1: Overzicht werking processor	1-1
1.1 Uitvoeren van programmacode.....	1-1
1.2 Het formaat van een processorinstructie.....	1-5
1.3 Fetch – Decode - Execute	1-6
1.3.1 De ‘instruction fetch’	1-6
1.3.2 Von Neuman architectuur versus Harvard architectuur	1-7
1.4 Instruction Set Architecture (ISA).....	1-8
1.4.1 Omschrijving	1-8
1.4.2 De 6502 architectuur.....	1-9
1.4.3 De Intel IA-32 architectuur.....	1-11
1.4.4 De ARM-architectuur	1-14
Hoofdstuk 2: basisarchitectuur van een processor	2-1
2.1 Interne architectuur van een processor	2-1
2.2 Basisschema processor	2-1
2.3 Control-Unit met hardware decoder	2-4
2.4 Microprogrammatie	2-5
2.5 Complex Instruction Set Computers (CISC)	2-7
2.6 Reduced Instruction Set Computers (RISC).....	2-8
Hoofdstuk 3: Pipelining	3-1
3.1 Model van een pipelined RISC architectuur.....	3-4
3.2 Pipeline ‘hazards’	3-5
3.2.1 Controlehazard.....	3-5
3.2.2 Oplossingen voor controlehazards.....	3-7
3.2.3 Speculative execution	3-8
3.2.4 Datahazard	3-8
3.2.5 Oplossingen voor datahazards	3-9
Hoofdstuk 4: Superscalaire processoren	4-1
4.1 Superscalaire architecturen	4-1
4.2 Een eenvoudig superscalair model.	4-2
4.3 Out of Order execution	4-3
4.4 Window of execution.....	4-4
4.5 Afhankelijkheden.....	4-5
Hoofdstuk 5: Vergelijking P4 ‘Netburst’ - Athlon	5-1
5.1 Pentium 4 - Blokschema.....	5-1
5.2 P4 Front Side Bus	5-2
5.3 Execution Trace Cache	5-2
5.4 Hyperpipeline	5-3
5.5 Out of order execution	5-3
5.6 Execution Units: ‘Rapid Execution Engine’	5-3
5.7 Besluit	5-4
5.8 Athlon-Blokschema	5-4

Hoofdstuk 6: ‘Very Large Instruction Word’ processoren	6-1
6.1 VLIW	6-1
6.2 De Intel Itanium	6-4
6.3 De Transmeta-Crusoë	6-5
Hoofdstuk 7: Andere vormen van parallelisme	7-1
7.1 Indeling architecturen: SISD, SIMD MISD, MIMD	7-1
7.2 Vectorinstructies (SIMD)	7-1
7.2.1 MMX	7-1
7.2.2 3DNow!	7-2
7.2.3 SSE (Streaming SIMD Expressions)	7-2
7.2.4 Altivec (Velocity engine, VMX)	7-2
7.3 Thread level Parallelisme (TLP): Hyperthreading	7-3
7.4 SMP: Symmetric multiprocessing	7-4
7.5 Multicore-processoren	7-6
7.6 Asymmetric multiprocessing (AMP)	7-6
7.7 Case-study: de Cell-processor	7-7
7.7.1 Overzicht.....	7-7
7.7.2 general purpose core (Power PC)	7-7
7.7.3 Synergistic Processor Elements (SPE)	7-8
7.7.4 Element Interface Bus (EIB)	7-9
7.7.5 Modellen voor gebruik van de Cell	7-9

Hoofdstuk 1: Overzicht werking processor

1.1 Uitvoeren van programmacode

Het hart van elk computersysteem is de processor (CPU) die zorgt voor het verwerken van de instructies waaruit het besturingssysteem en de toepassingen zijn opgebouwd.

Een processor is echter enkel in staat om een beperkt aantal relatief eenvoudige instructies te verwerken. Die instructies moeten onder de vorm van binaire informatie opgeslagen zijn in het geheugen van de computer.

Daarbij is het zo dat elke processorfamilie zijn eigen specifieke instructieset heeft. Een Intel x86 processor zal dus niet in staat zijn code uit te voeren die bedoeld is voor een ARM-processor en vice versa.

De programmacode die je als ontwikkelaar schrijft, moet op de één of andere manier uiteindelijk uitgevoerd worden door de processor, maar dat gaat dus niet zomaar. Afhankelijk van de taal en omgeving waarin je programmeert zal dat op een andere manier gebeuren.

Een kort overzicht van de meest voorkomende mogelijkheden:

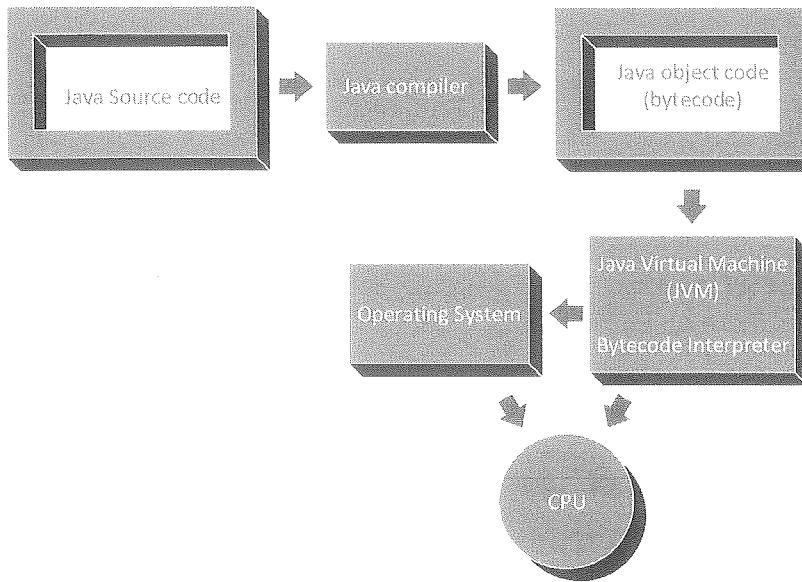
- ‘Managed’ code: Java, C#, VB.Net

De programmacode die je als ontwikkelaar schrijft, wordt in eerste instantie door de compiler omgevormd naar een tussencode: In het geval van Java wordt dit ‘bytecode’ genoemd, bij talen zoals C# en VB.Net spreekt men van ‘Common Intermediate Language (CIL)’.

Een dergelijke tussencode is gemeenschappelijk voor de gebruikte programmeertaal, en is onafhankelijk van het besturingssysteem of de specifieke processor waarop je toepassing uitgevoerd zal worden.

Wanneer je je toepassing laat uitvoeren, wordt de tussencode geïnterpreteerd binnen een speciaal afgeschermd omgeving (‘sandbox’): de ‘Java virtual machine (JVM)’, de ‘.Net common language runtime (CLR)’...

De toepassing die de tussencode interpreteert (de ‘interpreter’) zal op basis van de inhoud van de tussencode beslissen welke instructies de CPU moet uitvoeren en daarbij eventueel gebruik maken van services (methoden) van het besturingssysteem.



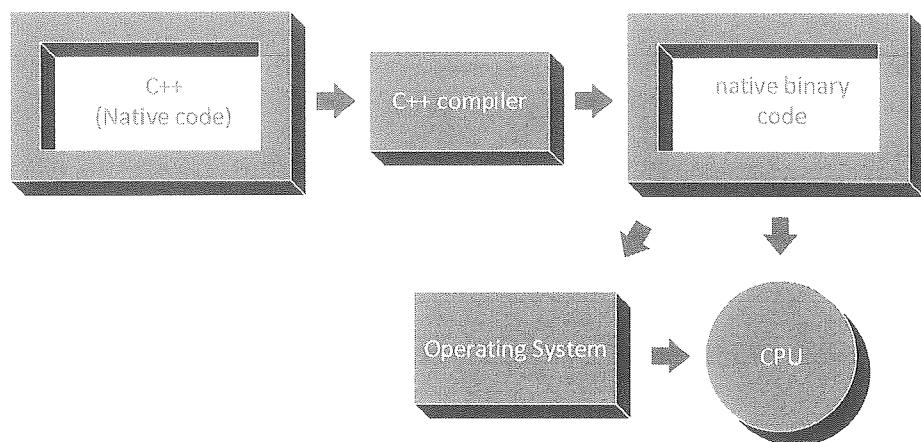
Deze manier van werken heeft enkele belangrijke gevolgen:

- De afgeschermde omgeving waarbinnen de tussencode uitgevoerd wordt zorgt voor een extra bescherming tegen kwaadwillige code of programmeerfouten. Zo hoeft de programmeur zich bv. niet veel zorgen te maken over het geheugemanagement. Een slecht geheugenbeheer is in klassieke ‘native’ talen (zie verder) één van de belangrijkste oorzaken van fouten en problemen als ‘memory leaks’. Bij ‘managed’ code zoals Java en .Net zal in de afgeschermden omgeving een ‘garbage collector’ regelmatig zorgen dat het geheugen dat niet meer nodig is voor je code, weer vrijgegeven wordt.
- De source code en de daaruit gegenereerde tussencode kunnen niet rechtstreeks uitgevoerd worden op de CPU. De interpreter binnen de afgeschermden omgeving zal bv. voor elke java/bytocode-instructie de nodige processorinstructies uitvoeren. Dit betekent uiteraard dat die interpreter zelf niet in managed code geschreven kan zijn: dat is een toepassing die uiteindelijk bestaat uit ‘native’ processorinstructies voor de specifieke CPU waarop de toepassing uitgevoerd wordt.
- Het hierboven beschreven mechanisme heeft een belangrijk nadeel: elke instructie van de tussencode die meermaals uitgevoerd wordt (bv. in een lus of een hulpmethode) zal bij elke uitvoering telkens opnieuw geïnterpreteerd moeten worden en aanleiding geven tot dezelfde reeks native processorinstructies. Die tussenstap zou de uitvoering van de toepassing sterk vertragen. In werkelijkheid wordt er gewerkt met ‘Just In Time’ compilatie (JIT): de eerste keer dat een instructie uit de tussencode uitgevoerd wordt, zal de daarvoor gegenereerde native processorcode opgeslagen worden. Voor elke volgende uitvoering van die tussencode zal rechtstreeks die opgeslagen native code uitgevoerd worden en valt de tussenstap van interpretatie van de tussencode weg. Door het mechanisme van JIT-compilatie zal Managed code in praktijk nauwelijks trager zijn dan klassieke ‘unmanaged’ code zoals die verder beschreven wordt.

- ‘Native’ code, unmanaged code: C, C++, Fortran, Pascal...

Hier schrijft de ontwikkelaar de programmacode in een ‘High Level Language’ die door een compiler vertaald wordt naar een binair bestand met native processorinstructies, aangevuld met de nodige informatie over hoe het binair bestand door het besturingssysteem in het geheugen geladen moet worden om het uit te voeren. Een dergelijk bestand wordt een ‘executable’ genoemd.

Om een dergelijke toepassing uit te voeren, zijn er geen tussenstappen meer (na de éénmalige compilatie):



In tegenstelling tot het programmeren in ‘managed’ code wordt de toepassings hier rechtsreeks uitgevoerd op de processor, binnen het besturingssysteem. Dat betekent in de eerste plaats dat de programmeur hier ook technische aspecten als geheugenmanagement helemaal zelf moet verzorgen.

Daarenboven is er geen enkele afscherming tussen de toepassingscode en de processor/het besturingssysteem. Hierdoor is de kans veel groter dat ‘native’ code een veiligheidsrisico vormt of de oorzaak is van een instabiel systeem.

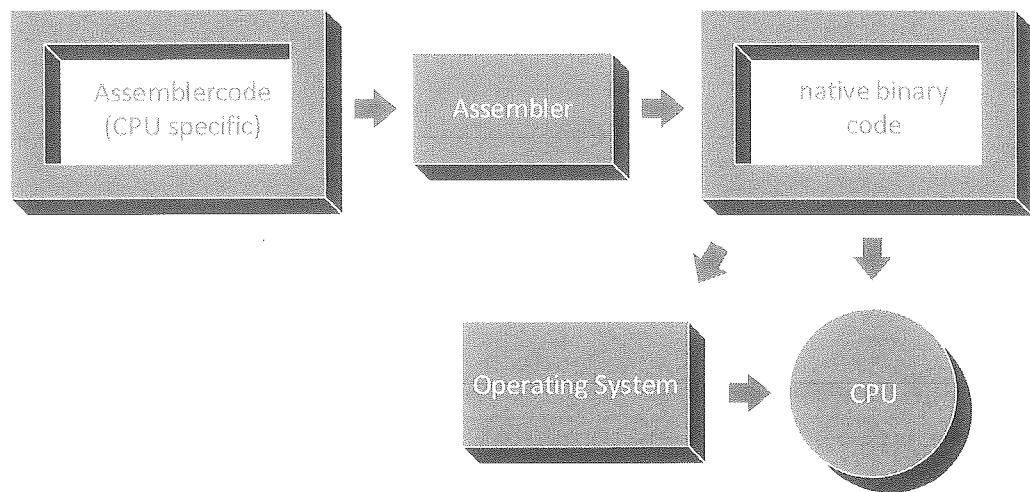
- Assembly language

Alle ‘High Level’ programmeertalen (zowel managed als unmanaged) zijn vooral ontworpen om toe te laten dat een ontwikkelaar op een efficiënte en productieve manier programmacode kan schrijven.

Een processor is echter niet in staat om dergelijke programmacode rechtsreeks uit te voeren. De compiler die uiteindelijk native processorinstructies genereert, moet garanderen dat de uitvoering daarvan het effect heeft dat de programmeur wou bereiken met de geschreven ‘source’ code.

Daarbij moet die native processorcode zo efficiënt mogelijk zijn (compact en snel in uitvoering). Moderne compilers slagen er meestal goed in om geoptimaliseerde code te genereren.

Toch zijn er situaties waar geprogrammeerd wordt in ‘assembly language’. Dit is een ‘low level’ programmeertaal, specifiek voor elke processor.



Als programmeur schrijf je eigenlijk programmacode met de native instructies zoals de processor die kent, enkel doe je dat in een voor mensen leesbare tekst en niet in binaire code.

Voorbeeld: x86 assembler code fragment:

```
DSEG SEGMENT
    blok1  DB 100H DUP ('*')
    blok2  DB 100H DUP (?)
DSEG ENDS

SSEG SEGMENT
    DW 1000H DUP (?)
SSEG ENDS

CSEG SEGMENT
ASSUME DS:DSEG, SS:SSEG, CS:CSEG

    Mov AX,DSEG           ; initialiseer DS
    Mov DS,AX
    Mov ES,AX               ; en ES
    Mov SI, offset blok1   ; start van blok1 in SI
    Mov DI, offset blok2   ; start van blok2 in DI
    Mov CX, 100H            ; 256 byte te copiëren
    CLD                    ; starten op laagste adressen
    REP MOVSB              ; copieer blok1 naar blok2
CSEG ENDS

END
```

Dit codefragment zal een blok van 256 bytes kopiëren van één locatie in het geheugen naar een andere. De source code bevat wat informatie over de indeling van het geheugen en verder een lijst met processorinstructies (‘Mov’...).

De assembler zal deze tekst omvormen naar de overeenkomende binaire code, waarbij voor elke instructielijn in de source exact één binaire instructiecode gegenereerd zal worden.

Als assembler programmeur werk je dus rechtstreeks met de instructies zoals de processor ze zal uitvoeren. Er wordt ook wel eens gesproken over ‘Low level’ programmatie. Omdat de

processorinstructies op zich zeer beperkt zijn, zal je in assembler ook veel meer instructies nodig hebben dan je voor een gelijkaardig programma in een hogere taal zou gebruiken.

Programmeren in assembly language heeft het voordeel dat je absolute controle hebt over de binaire code en je dus de mogelijkheid krijgt om zeer compacte en efficiënte code te schrijven. Dit veronderstelt dan wel dat je zeer goed op de hoogte bent van de interne structuur en werking van de processor. Bovendien zal je ook een goede programmeur met heel wat ervaring moeten zijn vooraleer je het beter zal doen dan een degelijke moderne compiler.

De term ‘assembler’ wordt zowel gebruikt voor de programmeertaal (en is dan een alternatief voor ‘assembly language’) als voor de toepassing die je source code in tekstvorm omzet naar de overeenkomstige binaire programmacode. Laat je hierdoor niet in verwarring brengen. De term ‘machinetaal’ wordt dan weer soms gebruikt voor de uiteindelijke binaire code zoals de processor die uitvoert.

1.2 Het formaat van een processorinstructie

Zoals hiervoor beschreven zal de processor uiteindelijk binair gecodeerde instructies inlezen en verwerken. Die instructies zullen dan ook in het geheugen van de processor opgeslagen zijn.

Een binaire instructie bestaat dus uit een aantal bits, maar die hebben wel een vaste structuur.

Elke instructie bevat een ‘opcode’ (afgekort voor ‘operation code’): dat is een binaire combinatie die aangeeft welke bewerking de processor moet uitvoeren (gegevens verplaatsen, een optelling uitvoeren, naar een andere instructie springen...).

Soms volgen dan nog een aantal bits die aangeven waar de data die nodig zijn voor de bewerking zich bevinden. Dit worden de ‘operanden’ genoemd;

Een voorbeeldje van een x86 instructie:

```
mov ah, 3fh ; stockeer de waarde 3F in het 8-bit AH-register
```

Deze instructie zal binair voorgesteld worden door 2 bytes:

B4 3F (hexadecimaal weergegeven)

De opcode ‘mov’ betekent dat de processor een byte moet verplaatsen naar het AH-register. Er zijn twee operanden: ‘ah’ is de bestemming van de verplaatsing (in dit geval het AH-register), ‘3fh’ is de waarde die verplaatst moet worden.

In de binaire voorstelling van deze instructie bevat B4 de opcode en de code voor de eerste operand, de tweede operand wordt gewoon in het 2^e byte van de instructie opgeslagen.

Een instructie bestaat niet noodzakelijk 2 bytes:

‘inc bx’ wordt binair gecodeerd als één byte: 43 (hexadecimaal).
‘mov bx,1’ wordt gecodeerd als: BB 00 01 (hexadecimaal)

Bij dit laatste voorbeeld wordt de operand ‘1’ gecodeerd in twee bytes omdat het hier over een 16-bit getal gaat (bx is een 16-bit register).

Bij het uitvoeren van een instructie zal de processor steeds eerst de opcode inlezen. Op basis van de verschillende bits waaruit de opcode bestaat zal de processor dan beslissen welke bewerking er moet uitgevoerd worden en of er nog operanden bij de instructie horen. In dat laatste geval moeten die ook eerst opgehaald worden uit het geheugen.

1.3 Fetch – Decode - Execute

Om zinvol werk te leveren, zal de processor opeenvolgende instructies moeten uitvoeren. Er wordt daarbij gesproken over de ‘Fetch - Decode - Execute’ cyclus, die telkens opnieuw doorlopen wordt voor de opeenvolgende instructies:

Alvorens een instructie uit te kunnen voeren moet deze uit het geheugen opgehaald worden : de **instruction fetch**. Na de instruction fetch wordt de binaire waarde van de instructie binnen de processor opgeslagen in een buffergeheugen (**instruction register - IR**).

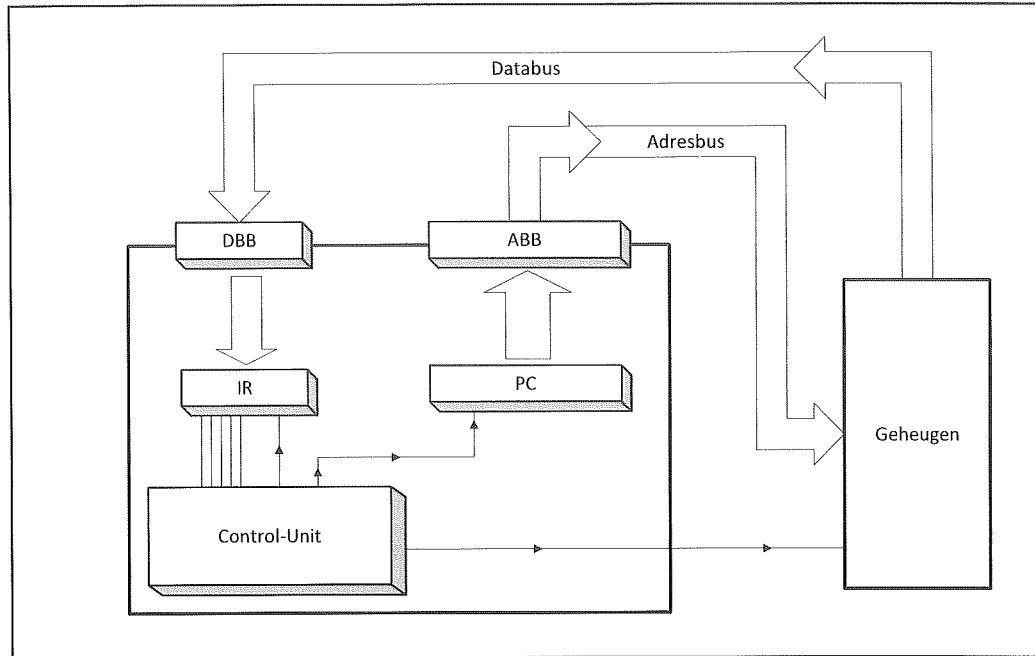
In een volgende fase van de instructieverwerking zal het besturingsgedeelte (de **control unit**) van de processor de instructie ontleden, om na te gaan welke elementaire bewerkingen nodig zijn om de instructie uit te voeren : dit noemt men de **decodefase**. Hier bepaalt de processor of er nog operanden opgehaald moeten worden en wat er nu precies moet gebeuren.

Tenslotte zal in de uitvoeringsfase (**execution-fase**) de control unit ervoor zorgen dat deze verschillende elementaire bewerkingen binnen de processor uitgevoerd worden door de nodige controlesignalen te genereren. Deze controlesignalen besturen op hun beurt dan weer de andere componenten van de processor en de rest van de computer (zoals bv. de read- en write-signalen).

Wanneer een instructie uitgevoerd is, zal de processor opnieuw een fetch-cyclus uitvoeren en zo de verwerking van de volgende instructie starten.

1.3.1 De ‘instruction fetch’

De verbinding van de adres- en databus met het inwendige van de processor gebeurt via de **adresbusbuffer (ABB)** en de **databusbuffer (DBB)**. Deze buffers dienen om de externe signalen voldoende vermogen te geven, en eventueel de spanningsniveaus aan te passen aan de externe elektronica.



Om bij het ophalen van de instructies het juiste adres ervan te kunnen genereren, bevat elke processor een telregister dat het adres bevat van de instructie die uitgevoerd moet worden : de **programmateller (Program Counter, Instruction pointer)**. Deze programmateller wordt bij elke instruction fetch automatisch verhoogd, zodat bij elke instructiecyclus een volgende instructie uitgevoerd wordt.

Het verloop van de instruction fetch ziet er dan als volgt uit :

- De inhoud van de programmateller (PC) wordt via de Adresbus buffer (ABB) op de adresbus gezet.
- De controle unit (CU) maakt het externe Read-signal actief zodat het geheugen geactiveerd wordt.
- Het geheugen plaatst de gevraagde instructie op de databus.
- De controle-unit vergrendelt de instructie-code via de databusbuffer (DBB) in het instructieregister (IR).
- De programmateller wordt verhoogd.

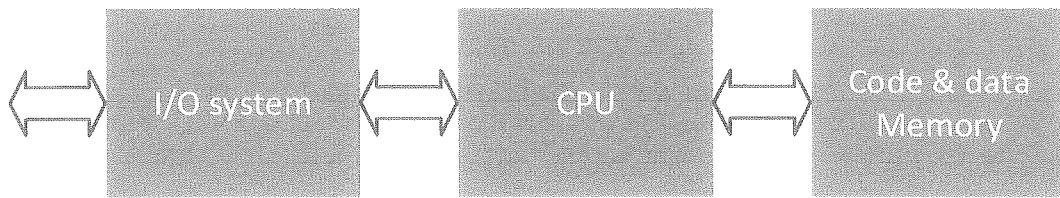
Indien de instructie uit meerdere geheugenwoorden bestaat (vanwege de extra operanden) zal de processor in de execute fase die extra operanden ook inlezen en de programmateller verder verhogen, zodat die uiteindelijk verwijst naar de opcode van de volgende instructie.

1.3.2 Von Neuman architectuur versus Harvard architectuur

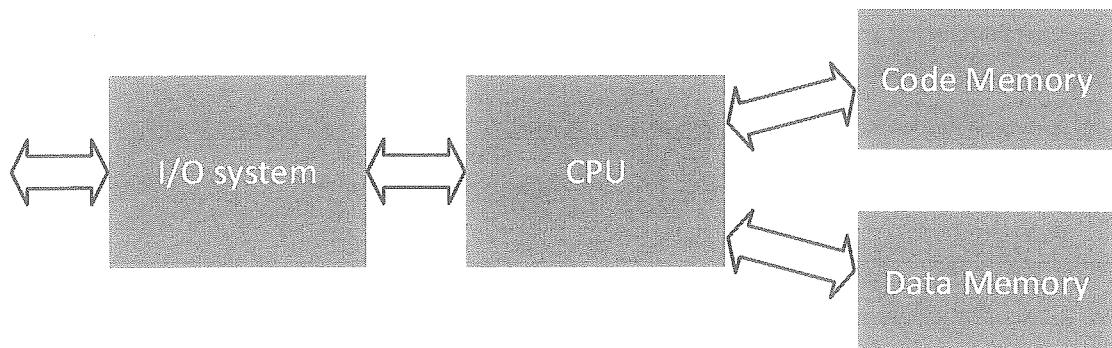
Zoals uit het voorgaande blijkt, zal de uit te voeren programmacode opgeslagen zijn in het geheugen van de computer.

Daarnaast is er ook geheugen nodig om gegevens op te slaan, zoals beelden, muziek, informatie uit een gegevensbank...

In de klassieke ‘Von Neumann’ architectuur bevatt de computer één geheugenstructuur waarbinnen zowel de programmacode als de andere gegevens opgeslagen worden.



Een alternatief voor deze opbouw is de ‘Harvard’ architectuur. Daarbij zijn er twee afzonderlijke geheugensystemen voor respectievelijk de programmacode en de gegevens:



Het voordeel van deze aanpak is dat het lezen van instructies en van data tegelijk kan gebeuren wat de computer sneller maakt. Daar staat dan wel een veel complexer geheugensysteem tegenover.

De Harvard architectuur zal dan ook vooral gebruikt worden in microcontrollers met on-chip geheugen: daar is de extra complexiteit relatief gemakkelijk en goedkoop te realiseren.

Bijna alle computers voor algemeen gebruik (zoals desktops, servers, notebooks...) zijn opgebouwd volgens de Von Neumann architectuur. Het gebruik van extern geheugen met een grote busbreedte (128 bit, 256 bit) maakt het gebruik van de Harvard architectuur onaanvaardbaar duur. Het nadeel van gemeenschappelijk geheugen voor programmacode en data wordt dan weer gemakkelijk gecompenseerd door een geschikte cachesysteem.

1.4 Instruction Set Architecture (ISA)

1.4.1 Omschrijving

Om in praktijk code te kunnen schrijven voor een specifieke processor moet je als programmeur uiteraard concrete informatie hebben over een aantal concrete aspecten van die processor. De belangrijkste hiervan zijn:

- Welke assemblerinstructies bestaan er voor de processor?
- Welke registers zijn er beschikbaar voor de programmeur?
Registers zijn interne geheugenplaatsen in de processor die zeer snel toegankelijk zijn en soms ook speciale functies of beperkingen hebben (zie verder in de voorbeelden).
- Hoe is het geheugen voor de processor georganiseerd en hoe kan het gebruikt worden vanuit de assemblerinstructies? Welke ‘adresseermethodes’ zijn er?
- Hoe gebeurt communicatie met randapparatuur: via een afzonderlijke adresbus en gespecialiseerde instructies (isolated I/O) of via hardware die door de processor ‘gezien’ wordt als gewone geheugenplaatsen (memory mapped I/O).
- Welke voorzieningen zijn er voor bescherming tussen verschillende processen onderling en tegenover het besturingssysteem.

Al deze informatie samen wordt aangeduid met de term ‘Instruction Set Architecture (ISA)’, niet te verwarren met de ISA-bus zoals gebruikt in de originele IBM PC.
In het Nederlands wordt ook wel de term ‘programmeermodel’ gebruikt.

In de eerste processoren was deze ‘instruction set architecture’ ook onmiddellijk de beschrijving van de interne hardware van de processor. Naarmate de processortechnologie evolueerde ontstonden er echter families van processoren die door de programmeur op dezelfde manier te programmeren zijn (dus met dezelfde ISA), maar waarbij verschillende processoren uit die familie intern heel anders opgebouwd kunnen zijn om bv. een betere performantie te halen. Gekende voorbeelden van dergelijke families zijn: de Intel IA-32 architectuur, de AMD64 architectuur, de ARM-processoren...

Hieraan volgt voor enkele architecturen een beknopte beschrijving die door plaatsgebrek noodzakelijk ook onvolledig is.

1.4.2 De 6502 architectuur.

De 6502-processor is in 1975 ontworpen door het bedrijf MOS-technologies. Het was samen met de Intel 8080 en de daarvan afgeleide Zilog Z80 één van de belangrijkste 8-bit processoren. Het was tegelijk een goedkope en zeer efficiënte processor. Bovendien was het vrij eenvoudig om op basis van de 6502 de hardware voor een compleet computersysteem te ontwerpen.

Deze eigenschappen verklaren het grote succes: hij werd onder andere gebruikt in de eerste ATARI-computers, in de Apple II, in de Acorn BBC computer en in de Commodore PET, VIC20, 64, 128 en zelfs in de eerste Nintendo spelconsole.

Programmeermodel:

- De 6502 heeft een 8-bit ALU en databus en een 16-bit adresbus.
- Op basis van de 16-bit adresbus kan de 6502 in totaal 65536 bytes (64 kB) geheugen adresseren. Daarbinnen moet zowel het noodzakelijke ROM-geheugen, RAM-geheugen als de gebruikte I/O-hardware een plaats vinden; Er wordt dus gewerkt met memory mapped I/O;

- **Registers:**
 - **A:** er is één algemeen 8-bit rekenregister: A (accumulator). Berekeningen en logische bewerkingen kunnen enkel in de accumulator gebeuren.
 - **X, Y:** en er zijn twee bijkomende 8-bit indexregisters (X, Y) voor adresberekeningen (zogenaamde geindexeerde adressering).
 - **SR:** het statusregister bevat de klassieke statusvlaggen: Zero (Z), Carry (C), Overflow (V), Negative (N), Decimal (D), Interrupt (I)
 - **PC:** de program counter (instruction pointer) is het enige 16-bit register in de 6502 en bevat het adres van de volgende instructie die uitgevoerd zal worden.
 - **SP:** de ‘stack pointer’. De stack is bij de 6502 slechts 256 bytes groot en bevindt zich in het RAM-geheugen in het gebied 0\$100..\$01FF).
- Er zijn **instructies** voor de klassieke binair bewerkingen (AND, OR, SHIFT, ROL), rekenkundige bewerkingen (geen vermenigvuldiging of deling), het verplaatsen van of naar de registers, (voorwaardelijke) spronginstructies en nog enkele gespecialiseerde instructies voor het gebruik van de stack en de statusvlaggen. De 6502 kan in hardware enkel 8-bit integer bewerkingen uitvoeren.
- Belangrijkste **adresseermethodes:**
 - **Implicit:** de instructie legt impliciet de operand(en) vast.
vb.: TAX ; copy value from accumulator to X-register
 - **Immediate:** de operand wordt rechtsreeks in de instructie gegeven.
vb.: LDA #\$10 ; Load hex. value 10 in A-register
 - **Absolute:** in de instructie wordt het geheugenadres opgegeven van de operand.
vb.: ADC \$2100 ; Add content from address \$2100 to accumulator
 - **Indexed:** het opgegeven adres wordt verhoogd met de inhoud van het gebruikte index-register (X of Y).
vb.: STA \$1000,X ; store Accumulator to address (\$1000 + (X))
 - **Indirect:** het te gebruiken geheugenadres staat zelf in het geheugen op de plaats die in de instructie opgegeven wordt. Bij de 6502 wordt indirecte adressering altijd gebruikt in combinatie met Indexed adressering en enkel voor adressen in de eerste 256 bytes van het geheugen..
vb.: CMP (\$B0,X) ; compare A to byte found at address in (\$B0 + X)
- **Interrupts:**

De 6502 heeft 3 mogelijke interrupts:

- **NMI:** de ‘non maskable interrupt’ kan niet uitgeschakeld worden. Wanneer de NMI-pin actief wordt, zal de processor de lopende instructie afwerken, de PC en

het SR op de stack plaatsen en dan springen naar het adres dat gevonden wordt op locatie \$FFFA (de ‘NMI interrupt vector’). Op \$FFFA en \$FFFF moet dus het adres van de interrupt handler voor NMI staan.

- **IRQ:** gelijkaardig als NMI, maar kan uitgeschakeld worden door de I-vlag in het statusregister op 1 te zetten. De interrupt vector voor IRQ wordt gelezen op adressen \$FFFE en \$FFFF.
- **BRK:** een software-interrupt. Via de BRK instructie wordt hetzelfde effect verkregen als een actief signaal op de IRQ-pin.

1.4.3 De Intel IA-32 architectuur

Als opvolger van de 8-bit 8080 processor bracht Intel in 1978 de 16-bit 8086 processor uit. Dit was een processor met een 16-bit databus en een 20-bit adresbus (waarmee 1MiB geheugen te adresseren was). Toen IBM in 1981 de eerste PC op de markt bracht, gebruikte het daarin een vereenvoudigde versie van de 8086, namelijk de 8088 waarbij de externe databus slechts 8-bit breed was (het lezen van elk 16-bit woord gebeurde door twee 8-bit leescycli na elkaar uit te voeren).

Na het enorme succes van de eerste generatie van de IBM PC was er al snel nood aan meer rekenkracht, meer bruikbaar geheugen en extra faciliteiten voor beschermingsmechanismes noodzakelijk voor multiprocessing. In 1985 bracht Intel daarom de 80386 processor uit die niet enkel compatibel was met de originele 8086 in ‘Real’ mode, maar die ook een eigen 32-bit ‘protected mode’ kende.

Later kreeg deze 32-bit architectuur de benaming ‘IA-32’. In de praktijk wordt er ook vaak over de ‘x86’ familie gesproken.

In 2004 voegde Intel nog een uitbreiding naar 64-bit toe onder de benaming ‘EMT64’. Dit was in feite een rechtstreekse implementatie van de AMD64 uitbreiding die AMD zelf had toegevoegd aan zijn eigen x86 compatibele processoren.

De belangrijkste uitbreidingen zijn een grote adresbereik (theoretisch 64 bit adressen) en ook kan de processor nu rechtstreeks 64-bit berekeningen maken.

Deze ‘64-bit mode’ van de IA-32 architectuur is niet te verwarren met de ‘IA-64’ architectuur: daarmee bedoelt Intel namelijk een heel andere 64-bit architectuur, beter gekend als de ‘Itanium. Deze komt later in deze cursus aan bod.

Het overgrote deel van de desktops, laptops en servers gebruikt processoren met de IA-32/AMD64 architectuur.

De complete IA-32 instruction set architecture wordt door Intel beschreven in enkele downloadbare handboeken.

Hier dan ook slechts enkele typische kenmerken:

- Bij het inschakelen of na een reset start de processor steeds op in ‘real mode’ en werkt dan net zoals een heel snelle 8086. Daarbij is er slechts 1 Mib geheugen beschikbaar en zijn er geen beveiligingsmechanismen.
Het is de bedoeling dat na de nodige initialisatie overgeschakeld wordt naar ‘protected

mode' waar de volledige capaciteiten van de processor beschikbaar zijn.

- Er wordt gewerkt met een 32-bit rekeneenheid die eveneens 8-bit en 16-bit bewerkingen kan uitvoeren. De externe databus is 32-bit breed net zoals de adresbus. Hiermee kan 4 GiB geheugen geadresseerd worden.
- **Randapparatuur** wordt meestal aangesloten via een 'Isolated I/O' mechanisme. Hier voor gebruikt de IA-32 een ruimte van 64KiB adressen en afzonderlijke instructies.
- **Geheugenorganisatie:** bij de IA-32 familie wordt het geheugen opgedeeld in afzonderlijke 'segmenten'. Dit zijn geheugenblokken met specifieke eigenschappen (grootte, locatie, type...). Zo zal een proces minstens een codesegment, een datasegment en een stacksegment hebben. Deze segmenten kunnen om het even waar in het geheugen zitten en eventueel overlappen indien gewenst.
De IA-32 heeft voorzieningen voor virtueel geheugen waarbij de programmeur het maximale theoretische geheugen kan gebruiken, ook als er minder fysiek geheugen beschikbaar is in de computer.
- Een overzicht van de belangrijkste **Registers** (32-bit versie):

EAX	AH	AL	AX	Accumulator
EBX	BH	BL	BX	Base
ECX	CH	CL	CX	Count
EDX	DH	DL	DX	Data
EBP	BP			Base-pointer
ESI	SI			Source-Index
EDI	DI			Destination Index
ESP	SP			Stack Pointer
	CS			Code segment
	SS			Stack segment
	DS			Data Segment
	ES			Extra segment
	FS		" "	" "
	GS		" "	" "
EIP		IP		Instruction Pointer
EFLAGS		FLAGS		

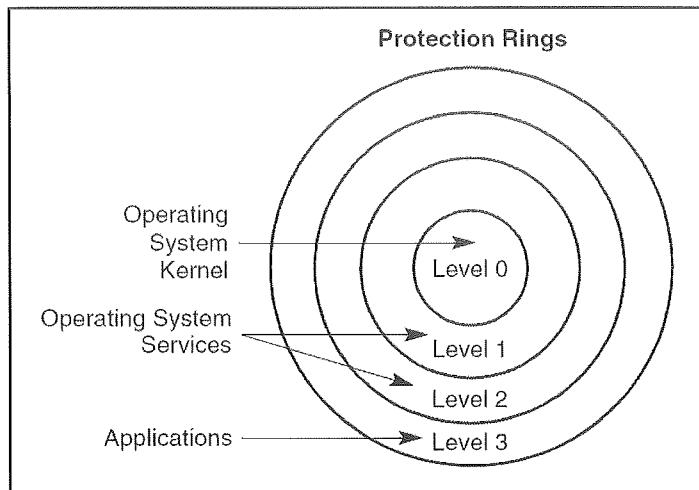
- Algemene rekenregisters: EAX, EBX, ECX, EDX: hierin kunnen bewerkingen uitgevoerd worden. Ze kunnen gebruikt worden als 32-bitregister (EAX), als 16-bit register (AX) of als 8-bit register (AH en AL).

CX en BX hebben bij sommige instructies een specifieke betekenis.

- Registers voor adresberekeningen: BP, SI, DI, SP. Deze kunnen enkel in speciale situaties bij adresberekeningen gebruikt worden. Er kunnen geen bewerkingen in gebeuren en ze kunnen niet als 8-bit register gebruikt worden.
 - De segmentregisters (CS, SS, DS, ES, FS, GS) bevatten informatie over de geheugensegmenten waarmee de processor werkt.
 - De instruction pointer (EIP): bevat het adres van de volgende uit te voeren instructie.
 - Het statusregister bevat onder andere de klassieke vlaggen: carry, parity, zero, sign, interrupt enable, overflow ...
 - Tenslotte zijn er ook nog een hele reeks controleregisters waarmee de werking van de processor geconfigureerd kan worden: werking in real/protected/64-bit mode, gebruik van virtueel geheugen, cache geheugen...
- **Instructies:** uiteraard kent de IA-32 architectuur de klassiek bewerkingen zoals je die in elke processor verwacht: verplaatsingen berekeningen (inclusief vermenigvuldiging en deling), binaire logische bewerkingen (and, or, xor, shift, rotate...), voorwaardelijke spronginstructies. Bewerkingen kunnen zowel in de rekenregisters als rechtstreeks in het geheugen gebeuren.
De processor bevat sinds de Intel 486 ook hardware ondersteuning voor floating point bewerkingen met afzonderlijke registers en instructies.
Verder zijn er in de loop van de tijd een aantal instructies bijgekomen voor speciale toepassingen zoals bijvoorbeeld de SSE-instructies voor de parallelverwerking van gegevens (zie hoofdstuk 7).
- Ook de IA-32 architectuur voorziet een hele reeks **Adresseermethodes:** register, immediate, direct, register indirect, register relative, base + index, base relative plus index, scaled indexed. Die bieden heel wat meer mogelijkheden dan bij de eenvoudige 8-bit processoren. Een aantal van die adresseermethodes zijn speciaal voorzien ten behoeve van compilers om het werken met arrays en velden binnen objecten eenvoudiger te maken.
 - Er is een uitgebreid mechanisme voor verwerking van **interrupts** en andere **exceptions**. De interrupt vector table bevat de adressen voor 256 verschillende exceptionhandlers voor zowel de externe hardware interrupts als de interne exceptions zoals een ‘divide by zero’, een ‘general protection failure’, een software interrupt via de ‘INT’-instructie...
 - Zoals de naam ‘**protected mode**’ aangeeft, zijn er ook een aantal voorzieningen die beveiliging ondersteunen van de werking van de processor/computer. Onder andere is het mogelijk er voor te zorgen dat processen enkel toegang hebben tot het geheugen dat ze zelf toegewezen kregen vanuit het besturingssysteem, dat ze niet zomaar toegang hebben tot alle aanwezige hardware en I/O-poorten, dat ze systeemroutines kunnen gebruiken zonder dat ze ook zelf over de privileges van het besturingssysteem beschikken enzovoort.

In het bijzonder voorziet de architectuur 4 beveiligingsniveaus die toegekend kunnen worden aan een proces.

Als een proces in ‘ring 0’ uitgevoerd wordt (ook vaak ‘kernel mode’ genoemd), kan het de hele werking van de processor/computer controleren en in principe toegang krijgen tot alle geheugen en hardware. Voor de andere niveaus (ring 1 tot ring 3) kan vanuit de systeemsoftware geconfigureerd worden welke toegang er mogelijk is.



In praktijk gebruiken bijna alle besturingssystemen slechts twee van de 4 mogelijke niveaus: Ring 0 voor de meest kritische delen van het besturingssysteem (de ‘kernel’: scheduler, memory manager en sommige hardware drivers) en Ring 3 voor alle toepassingsprogramma’s en de delen van het besturingssysteem die niet noodzakelijk extra privileges nodig hebben (dit wordt dan ook vaak aangeduid als ‘user mode’).

1.4.4 De ARM-architectuur

‘Acorn computers’ was een klein computerbedrijf uit Cambridge (Groot Brittannië) dat in 1981 een wedstrijd won voor het ontwerp van een bruikbare homecomputer voor de BBC. Deze 8 bit ‘BBC computer’ (op basis van de 6502-processor) was een groot succes voor het bedrijf dat vervolgens startte met het ontwerp van een eigen 32-bit processor op basis van de RISC-principes (zie hoofdstuk 2). Deze processor kreeg de naam ‘Acorn RISC Machine (ARM)’. Origineel was ze bedoeld voor desktop computers maar net op dat moment kwam de grote doorbraak van de IBM-PC op basis van de x86 architectuur. Er werd dan ook een andere afzetmarkt gezocht in de wereld van embedded control (in bv. laserprinters en andere hardware).

In praktijk bleek de ARM-architectuur zeer geschikt voor toepassingen waar een relatief grote rekenkracht nodig was bij een zeer laag energieverbruik. Daarbij bevat een ARM-core vanwege zijn RISC-architectuur typisch relatief weinig transistoren en is dus klein en dus goedkoop te produceren.

Een andere bijzonderheid is dat de originele ontwerpers er voor kozen om zelf geen ARM-processor te fabriceren. De ARM-architectuur werd daarentegen gecommercialiseerd door licenties te verkopen voor gebruik van de architectuur in ontwerpen van processoren. Het ontwerp wordt dus aangeleverd als een VHDL-bibliotheek (‘Intellectual Property’ of ‘IP’) die de klant – na het betalen van de nodige licenties – kon gebruiken bij het ontwerp van een eigen processor door er extra hardware (bijv. geheugen, I/O) aan toe te voegen.

Zowel de rekenkracht, de efficiëntie als het verkoopsmodel maakten de ARM een ideale kandidaat als processor in mobiele devices zoals smartphones en tabletcomputers. In 2013 werd het marktaandeel van CPU's voor mobile devices op basis van de ARM-core geschat op ruim 90%. In 2014 werden er wereldwijd zo'n 12 miljard CPU's op basis van de ARM-core verkocht!

De belangrijkste aspecten van het programmeermodel:

- De ARM was origineel een 32-bit processor, maar achteraf zijn er ook 64-bit versies uitgebracht.
- De ARM is ontworpen op basis van de RISC-principes zoals die in hoofdstuk 2 beschreven worden. Het is een load/store architectuur (zeer typisch voor een RISC processor). Dit betekent dat bewerkingen enkel in de interne CPU-registers kunnen gebeuren. De gegevens moeten daarvoor steeds uit het geheugen opgehaald worden en achteraf teruggeschreven worden.
- Om de load/store architectuur efficiënt te maken zijn er minimaal 16 registers (waaronder de program counter en het statusregister) van 32/64 bit beschikbaar die allemaal op gelijkaardige manier met alle instructies gebruikt kunnen worden.
- In tegenstelling tot de meeste andere processoren kunnen bijna alle instructies conditioneel uitgevoerd worden: er kan een conditie meegegeven worden op basis van de toestand van de statusvlaggen. Zo is het dikwijls mogelijk om een IF...THEN... ELSE constructie (die in klassieke processoren met conditionele sprongen gerealiseerd worden) te vervangen door één enkele conditionele instructie. Niet alleen maakt dit de code compacter en efficiënter, het heeft ook nog eens een zeer positief effect op de werking van de processor pipeline (zie hoofdstuk 3).
- Alle instructies zijn exact 32 bits lang. Dit maakt een heel efficiënte interne organisatie van de processor mogelijk.
- Het aantal instructies is beperkt, maar bijna alle instructies worden in één klokcyclus uitgevoerd.
- Er zijn uitgebreide en zeer krachtige adresseermethodes voor de load- en store instructies.
- Ook hier zijn er een aantal voorzieningen voor beveiliging aanwezig: de processor kan werken in ‘user mode’, in ‘supervisor mode’, in ‘system mode’ en in nog geenaantal toestanden specifiek voor de verwerking van interrupts.
- Mits goed geschreven assemblercode, kan de ARM-processor zeer efficiënt werken. Het vraagt echter een goede en ervaren programmeur om efficiënte code te schrijven. Op zich is dit niet zo'n groot probleem omdat er ondertussen goede compilers zijn die High-level talen als C en C++ omzetten naar zeer efficiënte assemblercode.

Hoofdstuk 2: basisarchitectuur van een processor

2.1 Interne architectuur van een processor

In vorig hoofdstuk werd uitgelegd dat de ‘Instruction Set Architecture’ de werking van de processor beschrijft ten aanzien van de programmeur/compiler.

Uiteraard moet de processor de nodige schakelingen bevatten om ook effectief de programmacode op de juiste manier uit te voeren.

In de periode van de 8-bit processoren was de basisarchitectuur van de meeste processoren zeer gelijkaardig. Toen lag de nadruk vooral op het ontwerpen van processoren die op een foutloze manier de instructies verwerkten.

Later kwamen er enkele grote ontwikkelingen:

- Het comfort verhogen voor de assembler programmeurs door het krachtiger maken van de instructies en adresseermethodes. Zo heeft de programmeur minder instructies nodig om een bepaalde toepassing te schrijven. Hierdoor wordt de programmeur productiever maar bovendien zal de code ook minder fouten bevatten, simpelweg omdat er minder instructies zijn.
- Het verhogen van de performantie/de rekenkracht van de processor. Dit gebeurt enerzijds door het verhogen van de klokfrequentie waarmee de processor werkt. Anderzijds probeert men de efficiëntie van de interne architectuur te verhogen: als er minder klokperiodes nodig zijn om een instructie uit te voeren, zal de processor ook bij gelijk blijvende klokfrequentie de programma’s sneller uitvoeren.
- Het invoeren van extra voorzieningen ter ondersteuning van multitasking. Bij multitasking worden door de processor meerdere processen/threads tegelijk uitgevoerd (via time sharing en/of vanuit verschillende processorcores). In zo’n situatie is het belangrijk dat elk proces voldoende geïsoleerd is van de andere processen en dus bijvoorbeeld niet in staat is om de informatie van een ander proces te lezen of te wijzigen.

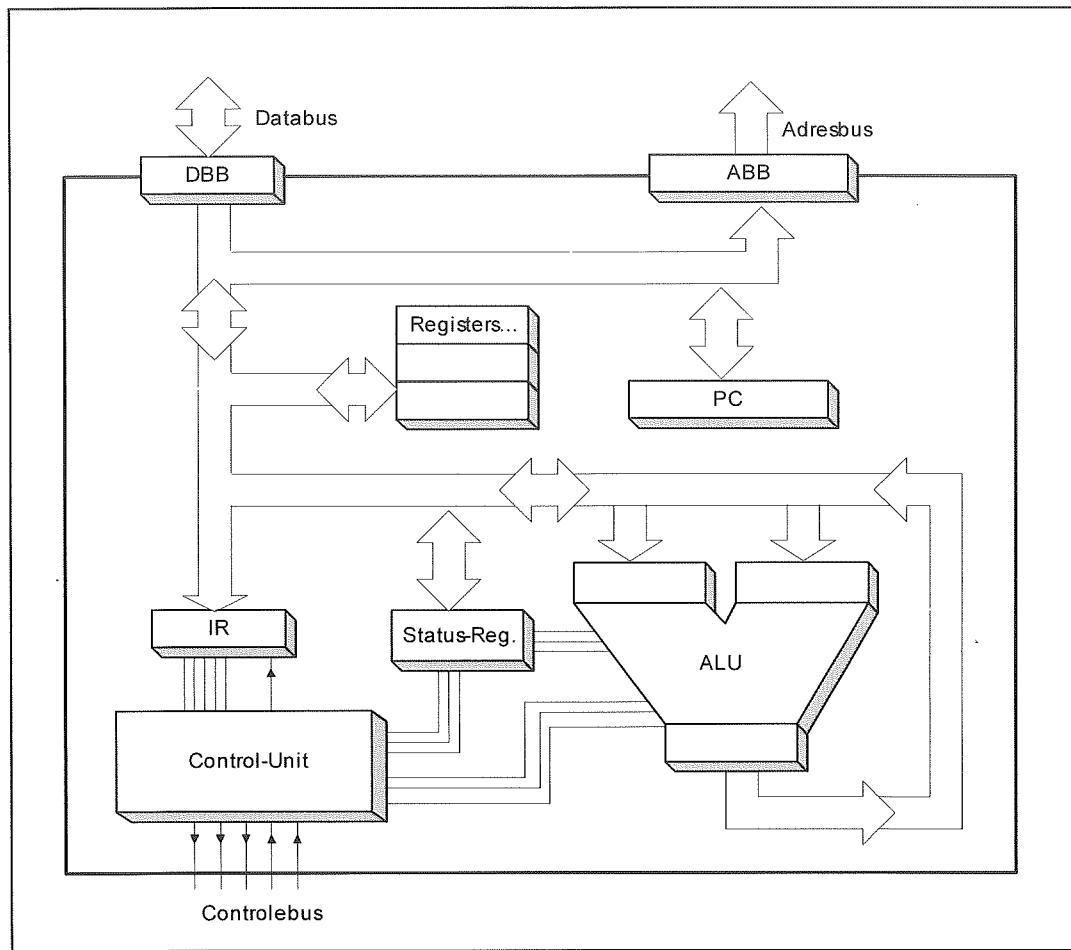
2.2 Basisschema processor

In dit hoofdstuk bespreken we de interne basisarchitectuur van een typische processor.

De belangrijkste elementen daarvan zijn:

- Registers (oa IP/PC en statusregister)
- Rekeneenheid
- Controle-unit
- Interne bussen
- Bus-buffers

Interne controlesignalen sturen de werking van de verschillende componenten.



Een processorregister is eenvoudigweg een kleine, snelle geheugenplaats (typisch opgebouwd uit een aantal D-flipflops) in de processor waarin één geheugenwoord van 8, 16, 32, 64... bits opgeslagen kan worden.

Elke processor heeft enkele gespecialiseerde registers:

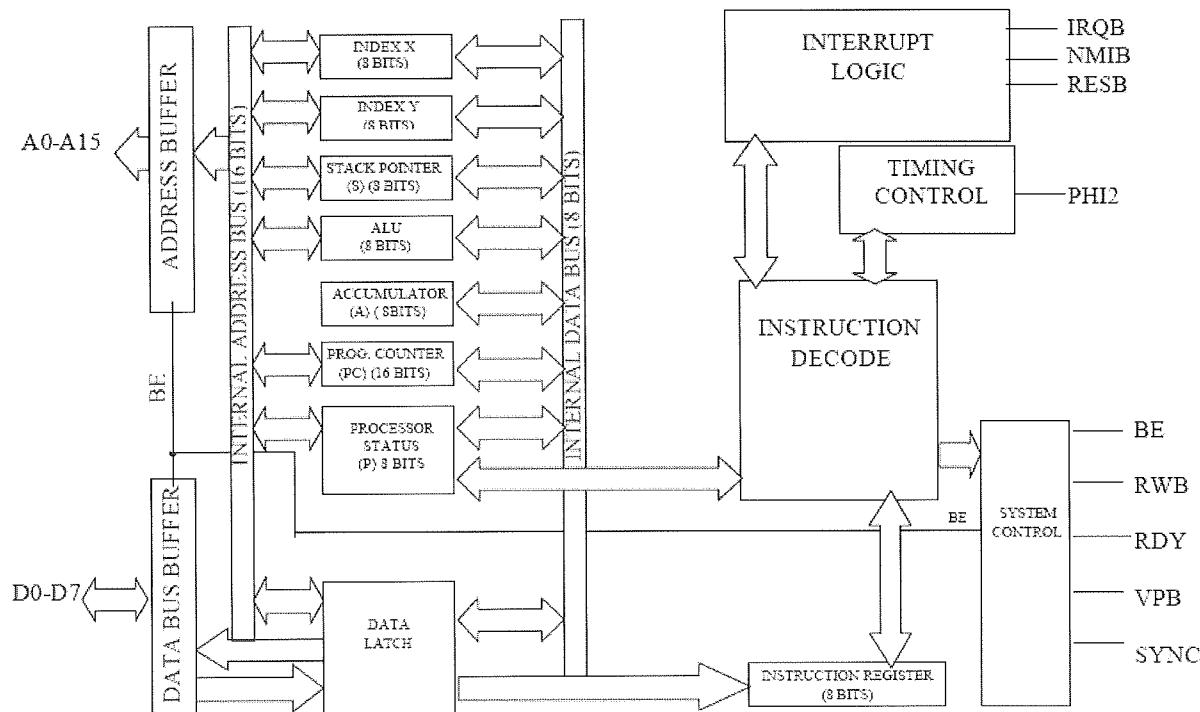
- De Program Counter/Instruction Pointer (PC/IP): bevat het geheugenadres van de volgende instructie die uitgevoerd moet worden. Dit register wordt bij elke fetchcyclus van een instructie automatisch verhoogd naar het adres van de volgende instructie.
- Het Instruction Register (IR): wanneer de operation code (opcode) van een instructie bij een fetchcyclus ingelezen wordt uit het extern geheugen, wordt deze code opgeslagen in het instructieregister. Vandaar kan de control unit de verschillende bits uit de opcode lezen bij het decoderen van de instructie.
- Het Status Register (SR): bevat een aantal vlaggen (bits) waarin de huidige toestand van de processor weergegeven wordt. Elke processor heeft zijn eigen verzameling statusvlaggen, maar een aantal daarvan komen bijna overal voor:
 - Zero vlag (Z): staat op 1 als het laatst berekende resultaat nul was.

- Carry vlag (C): bevat een eventueel carry-bit na een optelling/aftrekking.
- Sign vlag (S, N): bevat het hoogste bit (MSB) van het laatst berekende resultaat. Als er gerekend wordt met 2's complement getallen (signed integers), dan zal de Sign vlag op 1 staan als het laatst berekende resultaat negatief is.
- Overflow vlag (O, V): indien er gerekend wordt met 2's complement getallen en het resultaat van een bewerking kan niet correct voorgesteld worden binnen de gebruikte woordlengte van het register, dan zal de overflow vlag op 1 staan.
Bijvoorbeeld: in een 8-bit register kunnen 2's getallen voorgesteld worden in het bereik van -128 tot +127. Als nu in zo'n 8-bit register de waarde 120 opgeteld wordt met de waarde 10, wordt het resultaat 130, wat binair voorgesteld wordt als 10000010. Vermits het hoogste bit op 1 staat wordt dit in 2's complement notatie foutief geïnterpreteerd als een negatief getal: -126. In een dergelijk geval zal de overflow vlag dus aangevend at het resultaat niet zomaar gebruikt mag worden.

De statusvlaggen kunnen meestal gebruikt worden om te beslissen of een conditionele sprong uitgevoerd moet worden.

De rekeneenheid (Arithmetic and Logic unit , ALU) is de schakeling waar de bewerkingen effectief in uitgevoerd worden. Het gaat dan minimaal over optellingen, aftrekken van integer getallen en binaire bewerkingen (and, or, xor, shift, rotate). Afhankelijk van de processor komen daar ook nog vermenigvuldiging en deling bij, eventueel bewerkingen op BCD of floating point getallen.

Soms bestaat ook de mogelijkheid om in één stap een combinatie van verschillende bewerkingen te laten uitvoeren: Multiply and Accumulate (MAC) wordt vaak gebruikt bij digitale signaalverwerking.



Architectuur 6502-processor

De Control Unit (CU) coördineert de werking van de processor. Hij zorgt voor het genereren van de nodige controlesignalen (intern en extern) voor het ophalen van een instructie, decodeert de instructie en zal op basis daarvan de nodige controlesignalen genereren om de verwerking van de instructie te laten gebeuren.

De controle unit is met voorsprong de meest complexe schakeling in elke processor.

Naast de externe adres-, data- en controlebus zijn er ook intern in de processor de nodige bussen om de instructies en data binnen de processor te verplaatsen waar nodig.

Om technische redenen zijn er buffers nodig voor de koppeling tussen de interne en de externe bussen:

- Meestal wordt binnen in de processor met veel lagere spanningen gewerkt dan op de externe bussen. Intern wordt er bij moderne processoren gewerkt met spanningen tussen 0,8V en 1,5V terwijl extern geheugen met spanningen werkt tussen 1,3V en 5V (oudere generaties).
- Om de externe bussen in de computer aan te sturen is veel meer vermogen nodig dan voor de interne bussen op de processorchip.

2.3 Control-Unit met hardware decoder

In de allereerste processoren gebeurde het decoderen van de instructies na een instruction-fetch gewoon door de **hardware**.

De verwerking van een assembler-instructie (opgeslagen als een opcode met zijn verschillende operanden) gebeurt intern in een aantal opeenvolgende stappen. Bv. ADD AX,BX. Elk van die stappen gebeurt intern door het aan- of uitschakelen van een aantal controlesignalen.

Met elke mogelijke instructie komt dus een bepaalde opeenvolging van controlesignalen overeen. In de eerste microprocessoren werd die sequentie door een hardwareschakeling gegenereerd:

Verschillende bits van de operation code zijn digitale inputs voor een complexe digitale sequentiële schakeling. Elke verschillende combinatie van die bits heeft een eigen specifieke sequentie van de interne controlesignalen tot gevolg die overeenkomt met de verwerking van de gelezen instructie.

De decoder binnen in de control unit is dus eigenlijk een sequencer voor de digitale controlesignalen, waarbij elke opcode als ingang van de sequencer een eigen specifieke volgorde van de controlesignalen zal veroorzaken.

Het effect van de instructies is dus eigenlijk ingebakken in het ontwerp van de sequencer. Er wordt dan ook gesproken van een '**hardware decoder**'.

Naarmate er meer bits en complexere instructies kwamen, groeide ook de complexiteit van de decoder wat het ontwerp van de hardware erg moeilijk maakte.

Voordeel:

- Vermits de decoder een hardware schakeling is, zal deze zeer snel werken: de snelheid van het decoderen zelf en de uitvoering van elke stap in de verwerking van een instructie wordt enkel beperkt door de schakeltijd van de gebruikte technologie.

Nadeel:

- De control unit wordt een zeer complexe schakeling naarmate er meer en ingewikkelde instructies moeten voorzien worden.
- Het is erg moeilijk een dergelijke complexe schakeling te ontwerpen. Het ontwerp is ook moeilijk aanpasbaar, om een fout in het ontwerp op te lossen moet een volledig nieuw hardwaredesign gebeuren.
- Zeker in de beginjaren van de microprocessoren waren er geen krachtige systemen voor simulatie of ontwerp (CAD) van de hardware beschikbaar.

2.4 Microprogrammatie

Met het complexer worden van de 8-bit processoren en zeker bij de opkomst van de 16-bit processoren kwam vanuit de mainframewereld een technologie overgewaaid die het procesorontwerp drastisch veranderde: **microprogrammatie**.

Een korte uitleg:

De interne opbouw van een processor zoals tot nu toe beschreven, kan ook anders bekeken worden. Die bestaat namelijk uit twee grote afzonderlijke delen:

- Het ‘**datapath**’ is dat deel van de processor waar de gegevens effectief in opgeslagen, verplaatst of verwerkt worden. Het omvat de interne bussen, registers, ALU, buffers.
- De ‘**control unit**’ stuurt in een aantal opeenvolgende stappen de verschillende blokken van het datapath bij het verwerken van een programma-instructie als bijv. MOVE, ADD. Hierbij gebruikt de controle-unit interne controlesignalen waarmee bijv. gegevens uit interne registers op de interne bussen geplaatst kunnen worden.

Een hardware decoder zal op elk moment van de uitvoeringssequentie voor elk controlesignaal in de processor een specifieke toestand (hoog, laag) aanleggen. In plaats van die sequentie te laten genereren door een sequentiële hardware schakeling, kunnen de verschillende opeenvolgende toestanden voor elke instructie ook in een intern geheugenblok opgeslagen worden.

Om een instructie uit te laten voeren door de processor volstaat het dan op de juiste plaats in dat intern geheugen te starten en dan de opeenvolgende geheugenplaatsen naar buiten te brengen waar ze dan (ter vervanging van de hardware sequencer) de controlesignalen in de processor aansturen.

Elke locatie in dat intern geheugen wordt dan een ‘micro-instructie’ genoemd.

Op een dergelijke manier kan de uitvoering van een klassieke instructie (de ‘macro’ instructie zoals de programmeur die schrijft en die uit het extern geheugen opgehaald wordt), opgesplitst worden in een opeenvolging van een aantal **micro-operaties** of **micro-instructies**.

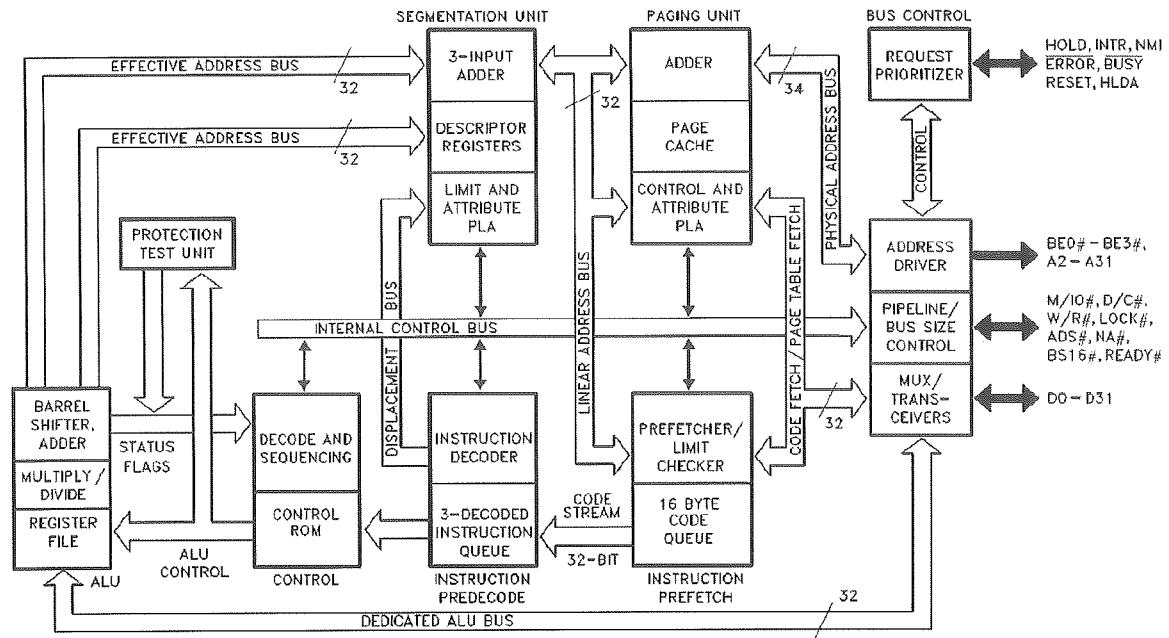
In tegenstelling tot de hardware decoder werkt men dan met een intern geheugen (de **control store**) waarin voor elke macro-instructie een opeenvolging van micro-instructies opgeslagen is.

De decoder zal dan op basis van de opgehaalde (macro)instructiecode een startadres in de control store kiezen en het uitvoeren van een macro-instructie gebeurt dan door een aantal opeenvolgende waarden uit die control-store te lezen en daarmee de controlelijnen aan te sturen. Er wordt in dat geval gesproken van ‘**micropogrammatie**’ en ‘**microcode**’.

Het gebruik van micropogrammatie heeft een aantal gevolgen:

- Het wordt gemakkelijk(er) om **nieuwe instructies** toe te voegen aan de Instruction Set Architecture van een processor, zonder de complexiteit van de processor hardware noodzakelijk te verhogen. Hiervoor moet enkel de control store uitgebreid worden met de microcode voor de nieuwe instructie (bijv. MUL op de 68000).
- Omgekeerd is het mogelijk de interne hardware te wijzigen en bv; krachtiger te maken zonder de ISA te wijzigen: enkel de microcode die de gewijzigde hardware aanstuurt moet aangepast worden. De processor zal dan helemaal compatibel blijven met vorige versies maar de code wel sneller uitvoeren.
- Eventuele **fouten** bij het ontwerp kunnen achteraf gemakkelijk verholpen worden door de microcode te herschrijven (bijv. Pentium-bug). Bij heel wat processoren is het tegenwoordig mogelijk bij het opstarten de interne microcode in ROM te vervangen door een nieuwe versie die dan in herschrijfbaar geheugen opgeslagen wordt. Zo is het mogelijk dat fouten in de ingebakken microcode van een processor bij het opstarten vervangen worden door goed werkende microcode. Hierdoor kan de fabrikant dus eigenlijk ‘bug fixes’ beschikbaar maken. Voor heel wat ontwerpfouten in een processor is het dan ook niet langer nodig om processoren om te wissen (een zeer kostelijke operatie voor de fabrikant).
- Het **processorontwerp** wordt eerder een kwestie van software (microcode) dan van hardware. Dit vergemakkelijkt het ontwikkelen en testen.
- Het wordt mogelijk om verschillende processoren te ontwerpen die toch **compatible** met elkaar zijn (de microcode van de verschillende processoren moet enkel dezelfde macro-instructies ondersteunen). Voorbeelden hiervan zijn de MC68xxx familie en de Intel x86 familie. Dit heeft een enorme boost gegeven aan de evolutie van de processoren in de zoektocht naar steeds hogere performantie.
- Bij gelijke klokfrequentie zal de decoder op basis van microcode trager werken dan een hardware decoder: het initialiseren en uitlezen van de verschillende microcode instructies zal (veel) meer tijd vragen dan de schakeltijd tussen twee opeenvolgende uitgangen van den hardware sequencer.

Tegenwoordig gebruiken alle moderne processoren een combinatie van hardware decoder en microcode. De meest gebruikte en eenvoudigste instructies worden dan in hardware gedecodeerd, terwijl meer complexe instructies via microcode uitgevoerd worden.



Bron: Intel Corporation

2.5 Complex Instruction Set Computers (CISC)

Met de jaren werd het aantal instructies dat een processor had sterk opgevoerd en deze instructies zelf werden steeds complexer en krachtiger. De bedoeling was het programmeren gemakkelijker te maken voor de assembler programmeur. Ook het ontwerp van een compiler werd hierdoor eenvoudiger.

Men ging dan ook spreken van CISC-processoren (CISC = Complex Instruction Set Computer).

De efficiëntie van de processor steeg daardoor echter niet noodzakelijk:

Een typisch kenmerk van een dergelijke processor is dat er een grote hoeveelheid instructies en adresseermethodes bestaan (wat impliceert dat er een grote hoeveelheid microcode aanwezig moet zijn), waarbij echter een aantal van de beschikbare complexe instructies slechts zelden gebruikt wordt.

Daarbij komt nog dat dergelijke complexe instructies meestal relatief traag uitgevoerd worden omdat er veel micro-instructies voor nodig zijn.

Ter illustratie: het aantal klokcycli nodig om een optelling en een vermenigvuldiging uit te voeren (16-bit) tussen twee registers in twee processoren die niet over een hardware multiplier beschikten:

# klokcycli/instructie	ADD	MUL
Motorola MC68000	4	70 (maximum)
Intel 8086	3	118

2.6 Reduced Instruction Set Computers (RISC)

Tegen ongeveer 1980 werd duidelijk dat de CISC-architectuur intrinsiek zware beperkingen had bij het zoeken naar hogere performantie. Zo kwamen er steeds meer gespecialiseerde instructies en adresseringsmethodes met als gevolg steeds meer microcode en ondersteunende hardware schakelingen.

In praktijk maakte dit het processorontwerp complexer maar zorgde dit er echter niet altijd voor dat de processor ook effectief sneller programmacode uitvoerde.

Tegelijk kende ook de technologie van het hardwareontwerp een evolutie waardoor er een veel grotere complexiteit haalbaar was dan in de beginperiode van de microprocessor.

Als gevolg hiervan ontstond de **RISC**-filosofie: de ‘Reduced instruction Set Computer’ gaat uit van een processor zonder microcode met een ‘hardwired’ decoder. Om die decoder snel en efficiënt te houden heeft een dergelijke processor slechts een zeer beperkt aantal elementaire instructies en adresseermethodes, die dan wel zeer snel en efficiënt uitgevoerd worden.

De redenering die daarbij gevuld werd: indien een processor minder en eenvoudigere instructies heeft, zal de programmeur meer instructies moeten gebruiken om hetzelfde effect te krijgen als bij een CISC processor met heel krachtige instructies. Maar als de RISC processor zeer efficiënt is, kan het sneller zijn om veel eenvoudige instructies uit te voeren dan enkele complexe.

Voor een succesvolle RISC processor is het dus belangrijk dat instructies zo efficiënt mogelijk uitgevoerd worden. Een belangrijke indicator daarvoor is het aantal instructies dat gemiddeld per klokcyclus uitgevoerd kan worden. Deze indicator wordt aangeduid als ‘Instructions per clockcycle (IPC)’. Als de uitvoering van een instructie gemiddeld 4 klokcycli in beslag neemt, heeft de processor een IPC van 0,25. Uiteraard is het de bedoeling deze IPC zo hoog mogelijk te krijgen met een geschikte processorarchitectuur.

Om dit te bereiken heeft een RISC architectuur enkele typische kenmerken:

- Een **beperkte, eenvoudige instructieset** maakt het mogelijk zonder microcode te werken en alle instructies in hardware te decoderen.
- Een **vaste instructielengte**. Bij een CISC processor zullen verschillende instructie vaak een verschillende lengte hebben (afhankelijk van mee te geven operanden of geheugenadressen). Zo kan in de IA-32 architectuur (ontstaan in de CISC-filosofie) de lengte van een instructie variëren van 2 tot 16 bytes. Bij de klassieke ARM-processor (RISC) is elke instructie 32 bits lang.
- Bewerkingen kunnen enkel in de processorregisters gebeuren en niet in het extern geheugen. Bewerkingen in interne registers zijn namelijk veel sneller dan in het extern geheugen. Daarbij bevat een RISC processor typisch veel algemene registers om efficiënte code mogelijk te maken. Zo zal een lusteller bijvoorbeeld meestal enkel in een register bijgehouden worden. Samengevat: **veel processorregisters** (enkel load/store operaties in het externe geheugen, geen bewerkingen).
- Het doel is om een IPC van 1 te bereiken.

Het gevolg van die filosofie is dat de programmeur meer code nodig heeft om hetzelfde werk te doen als bij een CISC-processor, maar dit wordt gecompenseerd door een hogere verwerkingsnelheid.

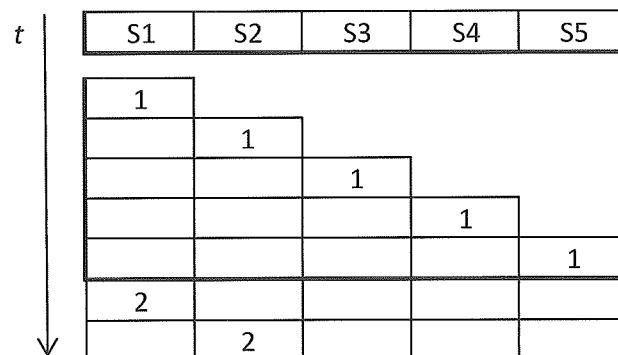
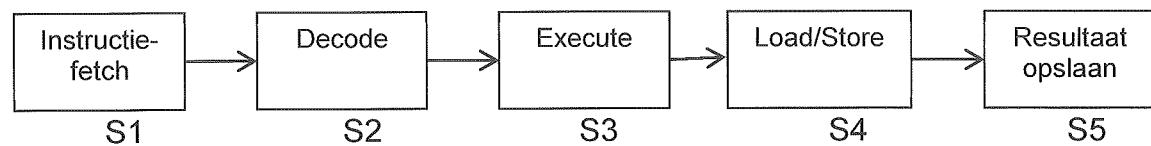
Een bijkomend gevolg van de vaste instructielengte is vaak dat de gemiddelde instructielengte bij een RISC processor groter is dan bij een CISC processor. Een gelijkaardig programma zal op een RISC processor dan ook dikwijls groter zijn dan op een CISC-processor.

Bekende voorbeelden van RISC-processoren zijn de PowerPC van IBM (door Apple gebruikt in oudere modellen van de MAC), de Alfabprocessor, de SPARC-processor en de ARM.

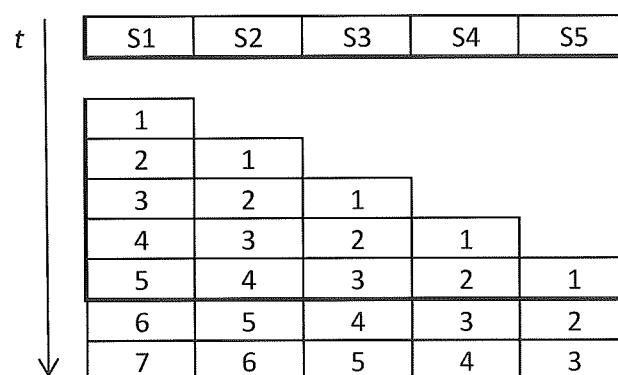
Tegenwoordig worden vaak aspecten van beide strekkingen samengebracht in een processor. Zo bevatten de moderne varianten van de IA-32 familie zowel een RISC-kern voor de meest gebruikte eenvoudige instructies, als microcode voor de complexere instructies.

Hoofdstuk 3: Pipelining

De verschillende stappen in de uitvoering van een instructie vereisen ook verschillende stukken hardware in de processor: het ophalen van de opcode, het decoderen, ophalen van operands, het uitvoeren van de bewerking, eventuele load/store naar extern geheugen het weg-schrijven van de gegevens. Dat heeft tot gevolg dat in de eerste processoren steeds slechts een beperkt deel van de processor effectief aan het werk was.



Bij de RISC-filosofie is het belangrijk dat instructies zo snel mogelijk uitgevoerd worden. Om die reden wordt het principe van een **pipeline** ingevoerd: verschillende instructies worden tegelijk uitgevoerd, elk in een ander deel van de processor (de verschillende ‘stages’):

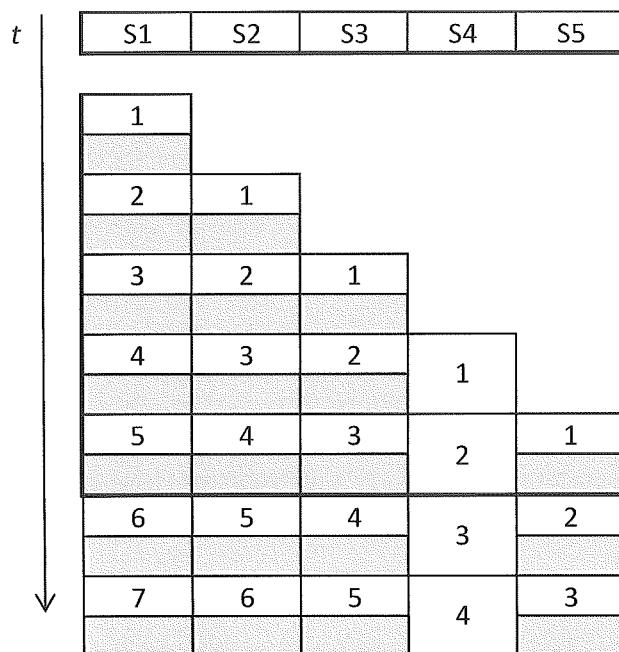


Door deze werkwijze verwerkt de processor elke klokcyclus een instructie, hoewel de verwerking van een individuele instructie toch 5 cycli in beslag neemt. Men spreekt ook wel van ‘instruction-level parallelisme’.

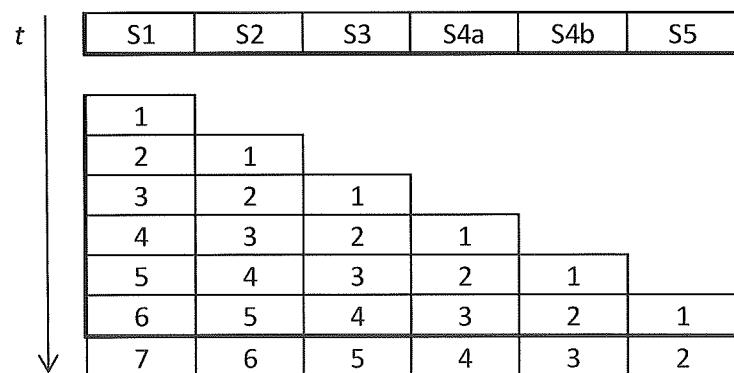
In praktijk zal uiteraard niet elke stap in de uitvoering even lang duren en kan de tijd nodig voor het afwerken van een stap afhankelijk zijn van de instructie die op een bepaald moment verwerkt wordt. Zo zal er in de ‘Execute’ stap uiteraard verschil zijn tussen instructies. En zal de load/store stap overbodig zijn voor instructies die enkel met interne registers werken.

Een belangrijk punt hierbij is de vraag hoe ver de **klokfrequentie** kan opgedreven worden. Dit hangt namelijk af van de tijdsduur die nodig is om de traagste fase van de vijf volledig af te kunnen werken (alle ‘stages’ worden synchroon geklokt).

Veronderstel dat S1, S2, S3 en S5 elk 4 nsec nodig hebben terwijl S4 8nsec in beslag neemt, dan zal de klokperiode gelijk zijn aan 8 nsec (klokfrequentie= 125 MHz). Dit betekent dus dat de maximale kloksnelheid onder meer bepaald wordt door de traagste ‘stage’ in de pipeline.



Omgekeerd kan de klokfrequentie ook opgedreven worden door de verwerking van een instructie op te splitsen in steeds meer ‘stages’: als in bovenstaand voorbeeld S4 opgesplitst kan worden in twee afzonderlijke delen (S4a en S4b) van elk 4 nsec, dan zal daardoor de klokfrequentie opgedreven kunnen worden tot 250 MHz (periode = 4 nsec).



Waar de eerste pipelines in processoren typisch uit een 5-tal stages bestonden wordt er tegenwoordig met meer (en kleinere, snellere) stages gewerkt:

	# stages
Intel 486	5
Intel Pentium	5 (8 voor FP-bewerkingen)
Intel Pentium III	10
AMD Athlon	10 - 17
Intel Pentium IV	20 - 31
Motorola PowerPC G4	7
Pentium M (Centrino)	12
Core 2 Duo	14
Nehalem	16
Sandybridge .. Haswell	14 - 19
ARM Cortex A9	8 - 11
ARM Cortex A15	15 - 24

Op basis van het voorgaande zal er een duidelijk verband zijn tussen de klokfrequentie en de lengte van de processor pipeline: hoe langer de pipeline, hoe hoger de klokfrequentie die gehanteerd kan worden (vanwege de kortere uitvoeringstijd van de afzonderlijke, kleinere stappen).

Toch zijn er nog een aantal andere beperkingen voor de klokfrequentie:

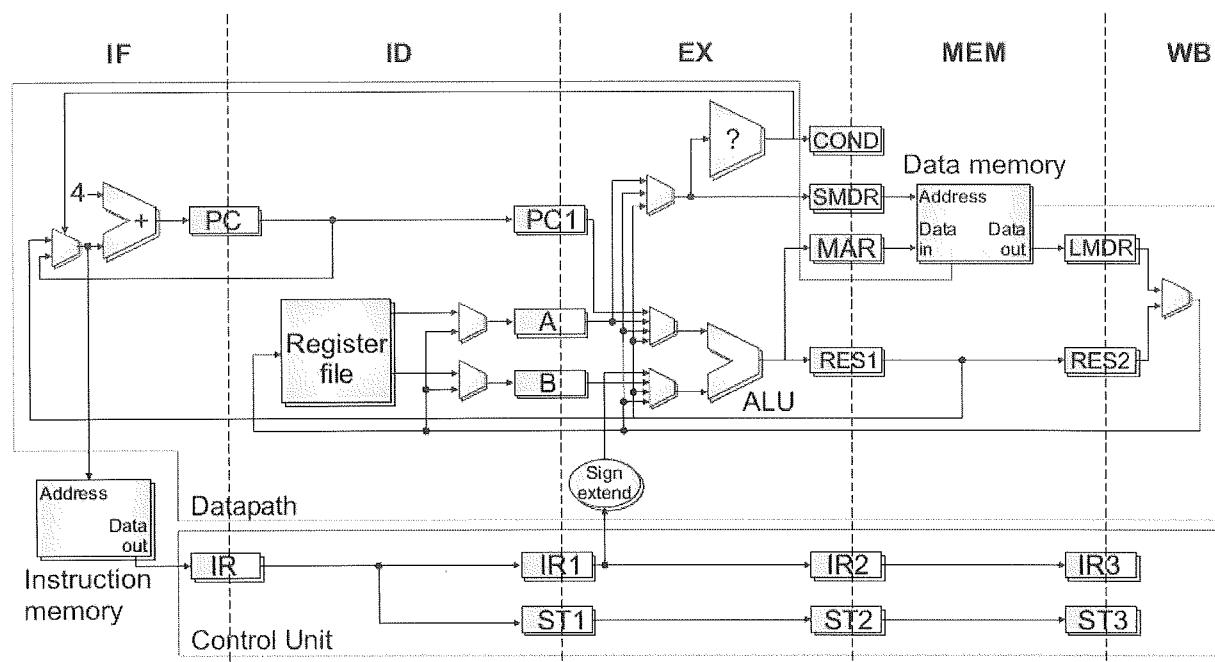
- De gebruikte technologie heeft een specifieke typische schakeltijd voor de digitale schakelingen (zie digitaaltechniek). Uiteraard kan ook die een beperking zijn voor de bruikbare klokfrequentie.
- Naarmate de klokfrequentie stijgt zal ook het als warmte gedissipeerde vermogen stijgen. In praktijk gebeurt dit echter niet noodzakelijk lineair.
- De voortplantingstijd van de elektronische signalen in de chip is eindig. Vermits we in de klassieke processoren typisch met synchrone schakelingen werken (waar de hele chip synchroon geklokt wordt door een centraal kloksignaal), zal ook de lengte van de interne verbindingen de klokfrequentie beperken.
Om deze beperking te verzachten kan met kleinere schakelingen en kortere afstanden gewerkt worden, maar dat heeft dan weer een negatief effect op het probleem van de warmteverliezen.

3.1 Model van een pipelined RISC architectuur

Ter illustratie bekijken we een theoretisch model van een pipelined RISC-architectuur zoals die beschikbaar is in de 'ESCAPE' processor-simulator, ontwikkeld aan de Universiteit Gent.

Er wordt gewerkt met eenvoudige instructiepijplijn met 5 'stages':

- **Instruction Fetch:** ophalen van de instructie uit het geheugen, en het aanpassen van de PC voor de volgende instructie.
- **Instruction Decode:** het decoderen van de instructie (bepalen wat er moet gebeuren ter uitvoering van de instructie) en het uitlezen van de operanden uit het registerbestand voor de ALU bewerking.
- **Execute:** uitvoering van de relevante berekening in de ALU
- **Memory:** indien de instructie een geheugenoperatie vereist, wordt die in deze stap uitgevoerd.
- **Write back:** het terugschrijven van het resultaat van de ALU operatie of geheugenlees-operatie naar het registerbestand.



In de figuur is het bovenste lichtgrijze blok het datapad met daarnaast rechts het data-geheugen. Onderaan staat links het programmageheugen en rechts daarvan de controle-unit.

Opmerking: in het schema staan data- en programmageheugen afzonderlijk getekend. In werkelijkheid zouden die ook dezelfde component kunnen zijn zodat we een normale Van Neumann-architectuur krijgen.

- **Fetch:** leest een instructie uit het geheugen, en berekent de volgende waarde voor PC. In geval van een sprong kan de volgende PC uit RES1 genomen worden, volgens de berekende sprongvoorwaarde cond.
- **Decode:** het decoderen van de instructie vertaalt IR in ST1 wat de gedecodeerde versie van IR is, en de aan te sturen controlepunten beschrijft. Voorts worden de operanden uit het registerbestand gelezen en klaargezet voor de ALU in A en B.
- **Execute:** uitvoering van de relevante berekening in de ALU. Behalve uit het registerbestand, kunnen operanden ook uit de PC of als constante uit de instructie komen. Ook wordt een eventuele sprongconditie hier berekend. Het resultaat van de ALU komt in res1, het resultaat van de sprongconditie in cond.
- **Memory:** indien de instructie een geheugenoperatie vereist, wordt die in deze stap uitgevoerd. Het geheugenadres werd in de ALU berekend, de te schrijven data (ingeval van schrijfoperatie) wordt uit het registerbestand doorgegeven via B en SMDR. Het uitvoeren van een geheugenoperatie kan verschillende cycli in beslag nemen.
- **Write back:** het terugschrijven van het resultaat van de ALU operatie (in RES2) of geheugen-lees-operatie (in LDMR) naar het registerbestand.

Hierbij zijn er een (beperkt) aantal instructies:

- ALU-instructies, bv. ADD r1, r2, r3 of ADDI r1, 19, r3
- Load/Store: bv. LDW r1, 0x100(r2)
- Jump: bv. brge r1, 0x10(pc)

3.2 Pipeline ‘hazards’

In een vorige paragraaf werd uitgelegd dat het opdrijven van het aantal ‘stages’ in de pipeline tot gevolg heeft dat er met een hogere klokfrequentie gewerkt kan worden. Toch heeft deze techniek ook nadelen: klassieke pipelining werkt immers enkel efficiënt als instructies uit een daarvoor geschikt programma zuiver sequentieel uitgevoerd worden.

In praktijk zijn er twee soorten mogelijke problemen: ‘controlehazards’ en ‘datahazards’:

3.2.1 Controlehazard

Bij een controlehazard bestaat het risico dat er instructies (gedeeltelijk) uitgevoerd worden waar dit eigenlijk niet zou mogen. Een dergelijk risico bestaat typisch bij spronginstructies.

Een symbolisch voorbeeld:

1
 2
BNE 7
 4
 5
 6
 7

Pas op het ogenblik dat de BNE-instructie uitgevoerd wordt (in S3) kan de processor beslissen welke instructie als volgende uitgevoerd moet worden.

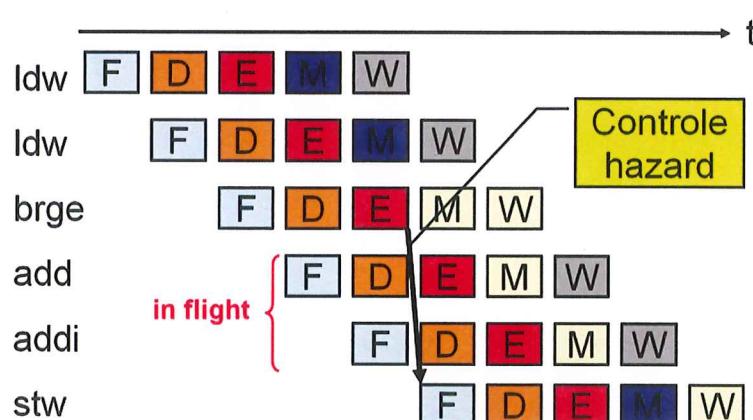
Op dat ogenblik ziet de inhoud van de verschillende plaatsen in de pipeline er als volgt uit:

<i>t</i>	S1	S2	S3	S4	S5
1					
2	1				
3	2	1			
4	3	2	1		
5	4	3	2	1	
7	5	4	3	2	
...	7	5	4	3	2

Als de sprong niet uitgevoerd wordt, is er niets aan de hand en werkt de processor op volle snelheid door.

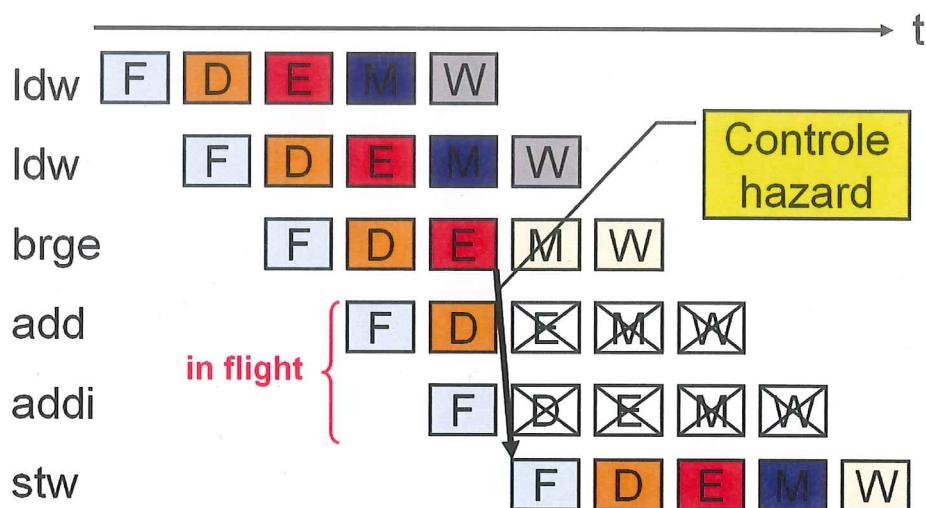
In het ander geval is er een probleem: de volgende instructie die moet uitgevoerd worden is 7, maar die is zelfs nog niet opgehaald, terwijl de instructies die wel al in de pipeline zitten (4, 5) helemaal niet mochten uitgevoerd worden.

Een praktisch voorbeeld:



3.2.2 Oplossingen voor controlehazards

Een eerste mogelijke aanpak houdt in dat pipeline ‘**geflushed**’ wordt: de overtollige instructies worden intern gewoon genegeerd (bv. in de pipeline vervangen door ‘NOP’-instructies), waarna de processor pas verder kan wanneer de instructie waarnaar gesprongen is in de pipeline in de execution-stap is terechtgekomen. Op die manier worden er gedurende een aantal kloocycli geen instructies verwerkt. Men spreekt van een ‘**stall**’ van de pipeline (die valt eventjes stil).



Een tweede oplossing bestaat erin te beschrijven dat de processor bv. de 2 instructies volgend op een spronginstructie steeds effectief uitvoert. De programmeur moet er dan rekening mee houden dat een spronginstructie pas effectief uitgevoerd wordt na de 2 erop volgende instructies: er wordt gesproken over een ‘**delayed branch**’.

Het komt er dan op aan dat de programmeur geschikte instructies vindt om na de spronginstructies te zetten. Lukt dat niet dan kan men nog steeds NOP-instructies gebruiken.

Bij een derde oplossing probeert de processor bij het ophalen van instructies op voorhand te voorspellen of een sprong al dan niet genomen wordt. Men spreekt dan van ‘**Branch prediction**’.

Er zijn verschillende technieken voor branch prediction. In het eenvoudigste geval wordt een tabel bijgehouden met daarin voor elke (recent uitgevoerde) spronginstructie het adres van de volgende uitgevoerde instructie. Telkens een spronginstructie ingelezen wordt, gebruikt de prefetcher dan voor de volgende instructie het adres uit die tabel.

Dit komt er op neer dat de processor veronderstelt dat als een sprong de vorige keer genomen werd, dit de volgende keer ook zo zal zijn. Die techniek werkt goed bij bv. een lus.

In praktijk zijn de Branch predictors heel wat gesofisticeerder en worden hitrates van 95% gehaald.

Zo had de AMD Athlon ruimte om informatie over 2048 sprongen op te slaan. De Intel P4 had een BTB (Branch Target Buffer) van 4096 plaatsen, maar een veel langere pipeline (Intel spreekt zelf van ‘Hyper-pipelining’) waardoor een eventuele fout in de voorspelling veel zwaardere gevolgen had.

3.2.3 Speculative execution

In het geval van een spronginstructie, voorspelt de processor met behulp van branch-prediction met welke instructies de pipeline verder gevuld zal worden. Het is echter nooit helemaal zeker of die voorspelling wel juist was, tot de spronginstructie zelf in de verwerkingsfase van de pipeline geraakt is.

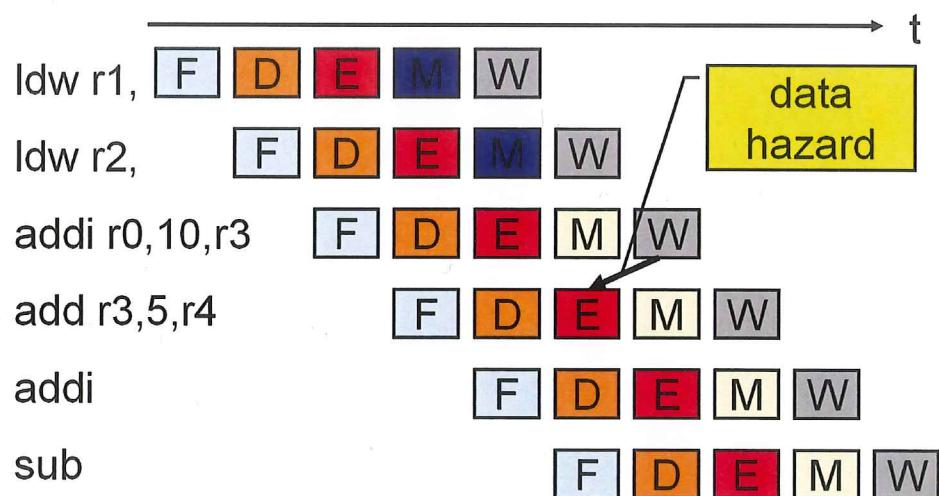
Men spreekt over ‘speculative execution’ van de instructies die op basis van de voorspelling opgehaald en al gedeeltelijk uitgevoerd zijn.

Voor het geval de voorspelling verkeerd was, moet de processor deze instructies op een speciale manier behandelen. Hiervoor zijn twee technieken:

- **roll back:** Bij een foutieve voorspelling worden alle effecten van de ten onrechte uitgevoerde instructies opnieuw te niet gedaan.
- **kladregisters:** De resultaten van de speculatief uitgevoerde instructies worden nog niet echt gebruikt of opgeslagen in het geheugen, maar bijgehouden in kladregisters, die gewoon gewist worden als blijkt dat de instructies niet mochten uitgevoerd worden.

3.2.4 Datahazard

Een datahazard ontstaat wanneer een instructie afhankelijk is van het resultaat dat in een vorige instructie berekend is: een zogenaamde ‘dependency’



In het voorbeeld worden de operanden voor de tweede ADD-instructie opgehaald en klaargezet in de ‘Decode’-stap van die instructie. Op dat moment is de voorgaande add-instructie nog maar in de ‘execute’-stap. Het resultaat van die eerste optelling zal pas twee stappenverder weggeschreven worden naar register r3.

Als hier geen rekening mee gehouden wordt, zal de tweede add-instructie dus uitgevoerd worden met een foutieve startwaarde voor r3.

Wanneer 2 opeenvolgende of nabije instructies hetzelfde object lezen of schrijven, ontstaat er een afhankelijkheid.

Deze bestaat in 4 soorten, afhankelijk van de normale uitwerking:

- **RAW** (*read after write*): het object wordt normaal eerst geschreven, dan gelezen. Dit is een echte afhankelijkheid. De leesoperatie moet wachten op de schrijfoperatie.
- **WAR** (*write after read*): het object wordt normaal eerst gelezen, dan geschreven. Dit is een anti-afhankelijkheid. De schrijfoperatie moet wachten op de leesoperatie, maar zou zijn resultaat in sommige situaties ook elders kunnen wegschrijven en op deze manier het wachten kunnen vermijden.
- **WAW** (*write after write*): het object wordt 2 maal geschreven. Dit is een anti-afhankelijkheid. De tweede schrijfoperatie mag niet gebeuren vóór de eerste schrijfoperatie, maar zou zijn resultaat in sommige situaties soms ook elders kunnen wegschrijven en op deze manier het wachten kunnen vermijden.
- **RAR** (*read after read*): het object wordt 2 maal gelezen. In dit geval is er eigenlijk geen afhankelijkheid. De leesoperaties in een willekeurige volgorde uitgevoerd worden zonder het resultaat van het programma te wijzigen.

3.2.5 Oplossingen voor datahazards

De anti-afhankelijkheden uit voorgaand lijstje kunnen vermeden worden door een verstandige registerkeuze door de programmeur (of compiler). Dit kan namelijk door voor de tweede instructie een doelregister te gebruiken dat niet door de eerste instructie gebruikt wordt.

Ook de processor kan dit in bepaalde omstandighedetecteren en zelf beslissen om een ander register te gebruiken. Er wordt dan gesproken over '**register renaming**'.

Sommige processoren hebben speciaal voor dit doel een aantal reserveregisters die niet rechtstreeks toegankelijk zijn voor de programmeur.

In het geval van de 'Read after Write' die een echte afhankelijkheid is, moeten er andere oplossingen gezocht worden:

Men kan die instructies die van elkaar afhankelijk zijn, ver **genoeg uit elkaar zetten**.

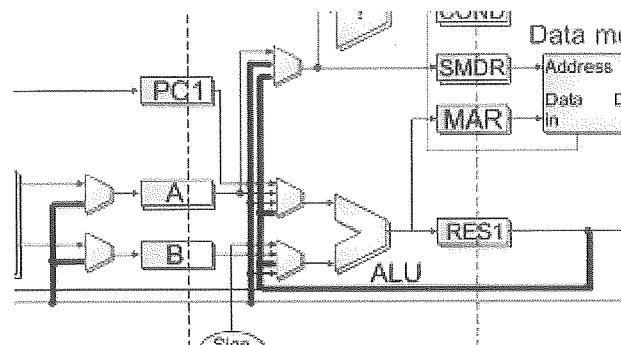
Wat precies ver genoeg is, hangt af van de architectuur (in het bijzonder van de pipeline). Dit is een statische methode: de wijziging gebeurt in de programmacode, en dus buiten de processor. Dit maakt de code zeer specifiek voor de machine, en ook weinig efficiënt indien er om voldoende afstand tussen instructies te bereiken NOP-instructies moeten bijgevoegd worden.

Alternatief kan men de **pijplijn blokkeren** tijdens de uitvoering, telkens zich een echte afhankelijkheid voordoet. Men stelt dan de uitvoering van de tweede instructie uit tot de eerste instructie afgelopen is, of althans ver genoeg gevorderd. Dit laat toe dat de programmacode niet bijzonder moet ingesteld zijn voor de pijplijn (correctie wordt dynamisch gewaarborgd), maar verhelpt niets aan het gebrek aan efficiëntie.

Tot slot kan men de **forwarding**-techniek introduceren in het datapad. Dit betekent dat van zodra dat het resultaat bestaat op de processor, het beschikbaar is voor volgende instructies

die dit resultaat gebruiken, zelfs al is het resultaat nog niet weggeschreven in het registerbestand.

Forwarding kan gebeuren vanuit de output van de executetrap, of vanuit de output van de geheugentrap.



Om **forwarding te realiseren**, is het nodig dat er een aantal extra verbindingen worden gemaakt in het datapad: zo wordt RES1 aangeboden aan de ALU (terugkoppeling over afstand 1: memory-trap naar execute-trap) en wordt RES2 / LMDR aangeboden aan de ALU (terugkoppeling over afstand 2: van write-back-trap naar execute-trap).

De terugkoppeling over afstand 3, write-back-trap naar decode-trap (van LMDR/RES2 naar A, B) kan als forwarding gezien worden, of als normale werking van het datapad of registerbestand. Het betekent alvast dat in eenzelfde klokcyclus de waarde die in het registerbestand wordt ingeschreven (door een write-back-fase), er ook kan worden uitgelezen door een decode-fase.

Opmerking:

De bedoeling van pipelining is om meerdere instructies tegelijk te laten verwerken en hogere klokfrequenties mogelijk te maken. Theoretisch kan er maximaal één instructie per klokcyclus afgewerkt worden. In praktijk zal vanwege control- en datahazards de performantie beneden de 1 IPC (instructions per cycle) liggen.

Om de die grens te doorbreken zal het nodig zijn extra vormen van parallelisme in te voeren.

Hoofdstuk 4: Superscalaire processoren

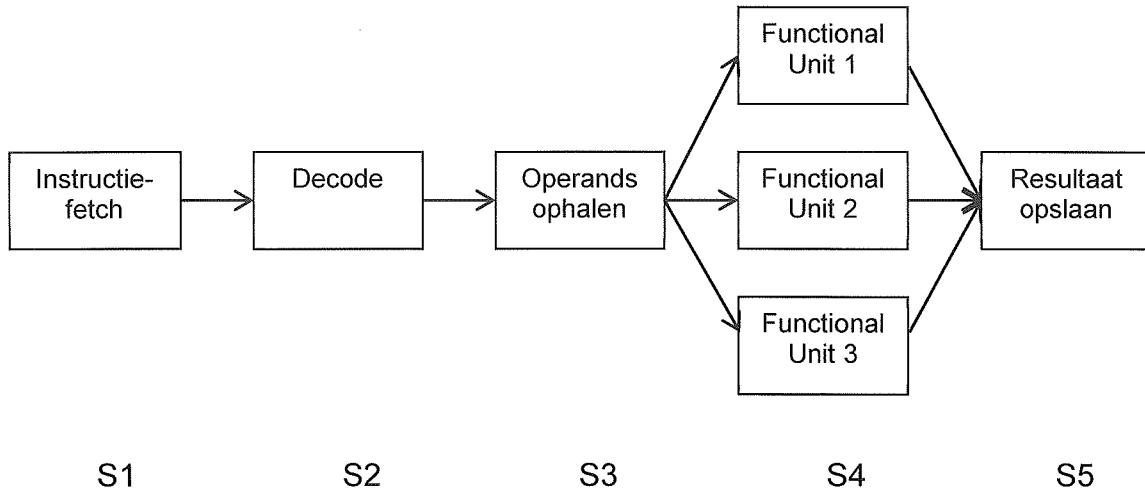
4.1 Superscalaire architecturen

In de zoektocht naar het nog sneller maken van processoren was een eerste stap het gebruik van meerdere pipelines. Toch heeft ook dit zijn nadeel: de processor wordt er niet enkel sneller door, maar ook in dezelfde mate complexer.

Tegelijk blijft er nog een belangrijk nadeel van het pipeline-principe bestaan: niet alle ‘stages’ kunnen even snel gemaakt worden en het is ook niet altijd mogelijk om de meest complexe ‘stages’ op te splitsen in meerder kleinere (zie ook hoofdstuk 2).

Een oplossing voor beide nadelen krijg je als enkel de meest kritische (traagste) stukken in de pipeline ontdubbeld worden.

In praktijk gebruikt men in de moderne processoren bijna overal verschillende **functional-units** (voor de verwerking van de instructies) in parallel.



Hierbij zijn de delen S1, S2, S3 en S5 in staat elke klokcyclus een instructie op te halen, terwijl elke functional unit meerdere klokcycli nodig heeft (en dikwijls intern weer als een pipeline is opgebouwd).

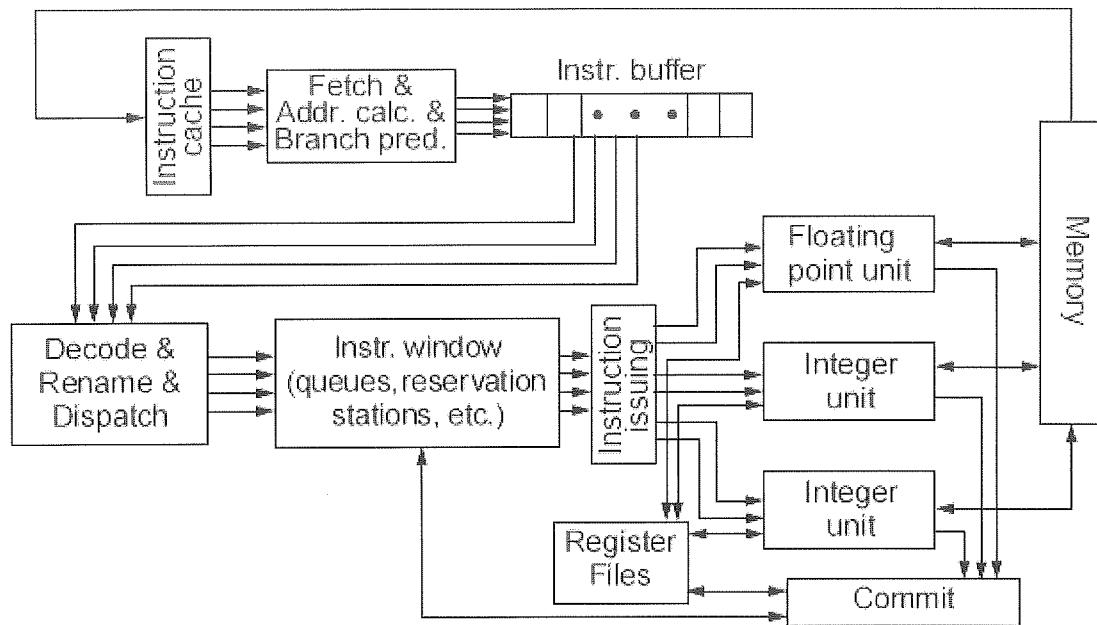
Dit soort ontwerp noemt men een ‘**superscalaire architectuur**’. Daarbij is het zeker niet altijd zo dat de verschillende parallelle units identiek of gelijkwaardig zijn. Zo zou het bijvoorbeeld kunnen dat een processor een snelle kleine ALU heeft voor integer bewerkingen, en parallel daarmee een tweetal grotere, tragere rekeneenheden voor floating point bewerkingen.

Om de processor zo snel mogelijk te maken (IPC zo groot mogelijk) worden in praktijk meerdere instructies tegelijk opgehaald en gedecodeerd, meestal vanuit een cache-hiërarchie, rekening houdend met een eventuele sprongvoorspelling.

4.2 Een eenvoudig superscalair model.

In het volgende voorbeeld kunnen twee integer operaties en één floating Point operatie tegelijk gestart worden.

Elke verwerkingsseenheid bestaat ook intern uit een pipeline, zodat elke unit tegelijk aan verschillende instructies kan werken.



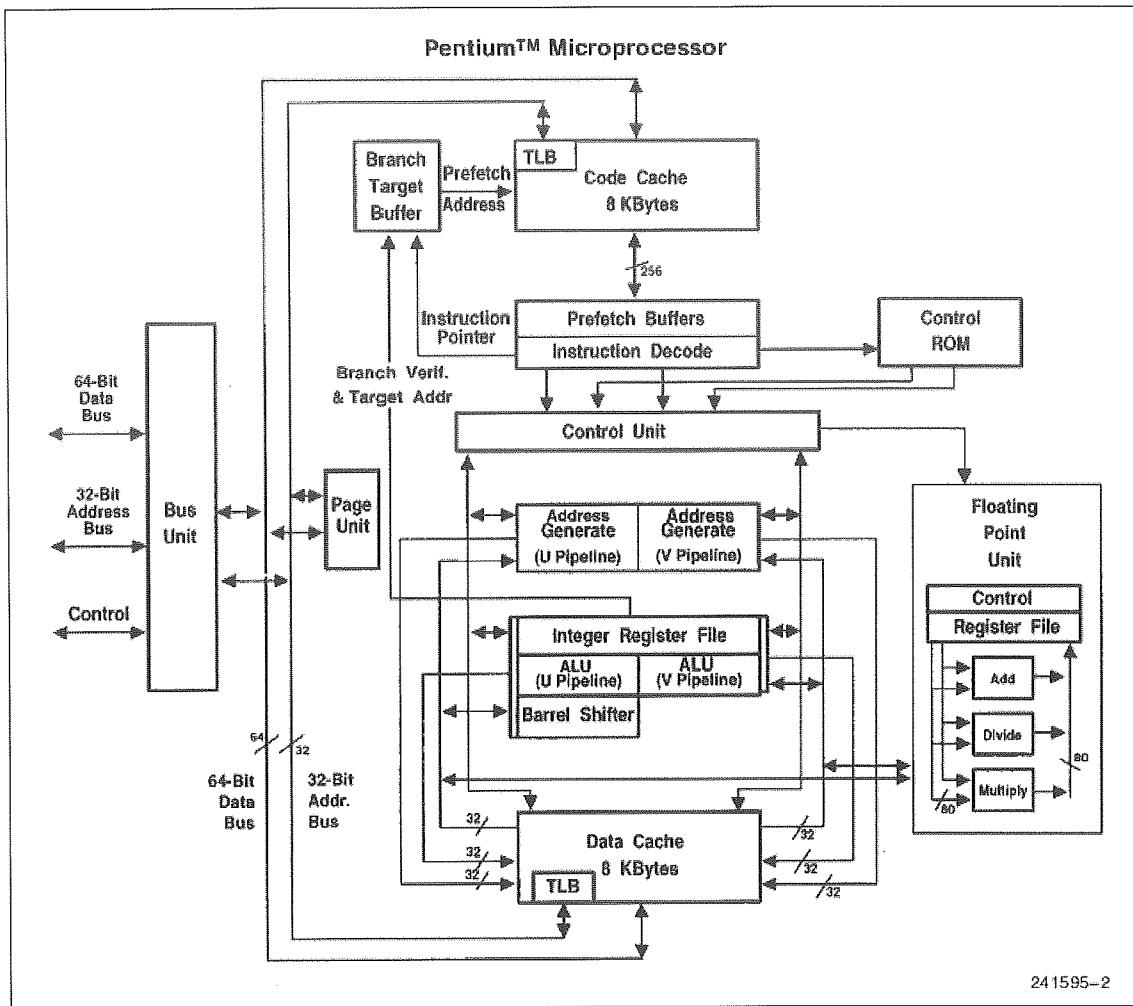
Bij het fetchen worden tegelijk meerdere opeenvolgende instructies opgehaald, die in parallel gedecodeerd worden. Waar nuttig wordt ‘Register Renaming’ toegepast (zie ook verder).

Wanneer een instructie klaar is voor verwerking wordt deze klaargezet (de ‘**Dispatch**’) in een wachtrij (het ‘**reservation station**’).

Zodra er een geschikte executing unit beschikbaar is, wordt de verwerking van de instructie gestart (‘**Issuing**’).

Het superscalair werken heeft nog een neveneffect: Bij een gewone pipeline is er een eenvoudige koppeling tussen de execution unit en de processorregisters. Deze laatsten hebben een ‘READ’-port langs waar bijvoorbeeld de ALU de inhoud kan lezen en een ‘WRITE’-port om de resultaten terug te schrijven (eigenlijk zijn dit respectievelijk uit- en ingangen van een reeks eenvoudige D-flipflops).

Door meerdere functional-units te gebruiken moet er extra logica zijn om de koppeling te voorzien en eventuele conflicten te vermijden. De verschillende registers worden dan ook verzameld in een ‘**register file**’ die je kunt vergelijken met een klein stukje RAM-geheugen met eigen data- en adresbussen.



Bron: Intel Corporation

4.3 Out of Order execution

Door het invoeren van een superscalaire architectuur, ontstaat de mogelijkheid dat meerdere instructies tegelijk uitgevoerd worden.

De eenvoudigste aanpak hiervoor is dat de verwerking van instructies opgestart wordt in de volgorde waarin de instructies in het programma staan ('**in order issuing**'). Daarbij is de kans echter niet zeer groot dat verschillende opeenvolgende instructies ook geschikt zijn om tegelijk uitgevoerd te worden. Dit kan enkel als

- die opeenvolgende instructies in verschillende execution units uitgevoerd kunnen worden.
- er geen afhankelijkheden zijn tussen die instructies (zie pipelining en verder).

In praktijk zullen de verschillende execution units dus zeker niet altijd allemaal permanent aan het werk zijn.

Om de beperkte efficiëntie van ‘in order issuing’ te verbeteren wordt dan ook meestal gebruik gemaakt van ‘**dynamic instructions scheduling**’, of met een andere naam ‘**out of order execution**’.

Hierbij wordt voor de verwerking van verschillende instructies enkel rekening gehouden met de afhankelijkheden tussen instructies en de beschikbare execution units, en niet met de oorspronkelijke volgorde van de instructies.

Het uiteindelijke resultaat van het programma moet uiteraard wel identiek zijn aan wat verkregen zou worden bij een zuiver sequentiële verwerking.

Na de ‘out of order’ verwerking in de verschillende parallelle takken komt er ook nog een extra stap. Die zorgt ervoor dat de effecten van instructies naar buiten toe gezien worden alsof de instructies in de normale volgorde uitgevoerd zijn met de in het programma vermelde registers. Dit gebeurt in de ‘**reorder buffer**’ of ‘**retire unit**’ (in het schema aangeduid met ‘commit’).

De eerste processor uit de x86 familie met ‘out of order execution’ was de Pentium II.

4.4 Window of execution

Bij de keuze van de uit te voeren instructies moet dus goed rekening gehouden worden met eventuele afhankelijkheden tussen die instructies.

Gevolg hiervan is dat er steeds slechts een beperkt aantal instructies kandidaat zijn voor parallele verwerking.

Om de efficiëntie van de processor te verhogen is het dus belangrijk dat het aantal instructies waartussen gekozen kan worden voldoende groot is. De instructies waartussen gekozen kan worden bij de ‘issuing’ wordt het ‘**window of execution**’ genoemd. Dit is liefst zo groot mogelijk, maar wordt bepaald door twee factoren:

- De processor hardware: capaciteit om snel instructies op te halen, te decoderen en op te slaan in het reservation station.
- De structuur van het programma zelf (spronginstructies!):

Een voorbeeld:

```
for (i=0; i<last; i++)
{
    if (a[i] > a[i+1])
    {
        temp = a[i];
        a[i] = a[i+1];
        a[i+1] = temp;
        change++;
    }
}
```

Een mogelijke assemblervertaling hiervoor:

.....
r7: address of current element ($a[i]$)
r3: address for access to $a[i]$, $a[i+1]$
r5: change; r4: last, r6: i
.....

L2	move	r3,r7		basic block 1
	lw	r8,(r3)	$r8 \leftarrow a[i]$	
	add	r3,r3,4		
	lw	r9,(r3)	$r9 \leftarrow a[i+1]$	
	ble	r8,r9,L3		
	move	r3,r7		basic block 2
	sw	r9,(r3)	$a[i] \leftarrow r9$	
	add	r3,r3,4		
	sw	r8,(r3)	$a[i+1] \leftarrow r8$	
	add	r5,r5,1	change++	
L3	add	r6,r6,1	i++	basic block 3
	add	r7,r7,4		
	blt	r6,r4,L2		

Door gebruik te maken van ‘branch prediction’ kan het ‘window of execution’ groter gemaakt worden dan de afzonderlijke basic blocks.

Omdat er natuurlijk geen garantie is dat de sprongen correct voorspeld worden, zal er gebruik gemaakt worden van ‘speculative execution’: de instructies van het voorspelde pad worden voorlopig uitgevoerd. Pas nadat een spronginstructies berekend is en de voorspelling correct blijkt, worden de resultaten permanent en zichtbaar voor de rest van de processor.

Men zegt dat er een ‘commit’ gebeurt (net zoals bij een transactie op een databank): de berekende waarden (in de reorder buffer of retire unit) worden effectief gebruikt voor opslag in het geheugen/registers en/of voor verdere berekeningen.

4.5 Afhankelijkheden

Bij de invoering van de pipeline en de mogelijke parallelle uitvoering van instructies is het probleem van de afhankelijkheden opgedoken.

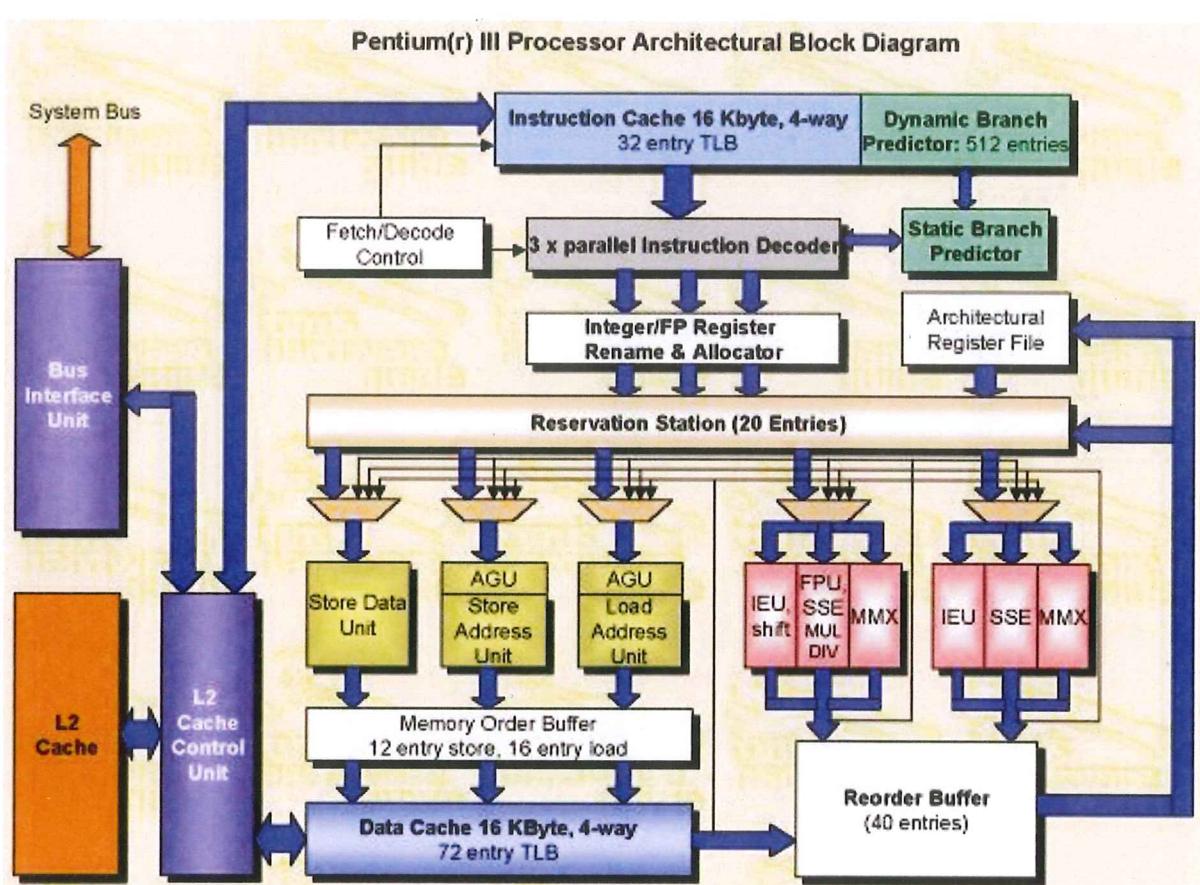
Het gebruik van een superscalaire architectuur verhoogt de kans op parallele uitvoering en dus ook de kans op problemen met afhankelijkheden.

De anti-afhankelijkheden zoals vermeld in vorig hoofdstuk geven geen problemen bij een stricte ‘in order execution’, maar wel mogelijk bij ‘out of order execution’.

De oplossing daarvoor is zoals vroeger vermeld de techniek van ‘register renaming’. Om een efficiënte werking bij een groot ‘window of execution’ mogelijk te maken is het belangrijk dat er voldoende registers beschikbaar zijn voor de processor.

De ‘echte’ afhankelijkheden (read after write) zijn nu iets minder problematisch: naast de techniek van forwarding was het mogelijk dat de pijplijn stilgelegd moest worden om op een resultaat van een voorgaande berekening te wachten. Vanwege de out of order execution, bestaat nu de mogelijkheid dat die ‘verloren tijd’ gebruikt kan worden om andere instructies uit het reservation station te verwerken.

De Pentium III Voorbeeld van een superscalaire processor met branch prediction, register renaming en out of order execution.



Bron: www.tomshardware.com

Hoofdstuk 5: Vergelijking P4 'Netburst' - Athlon

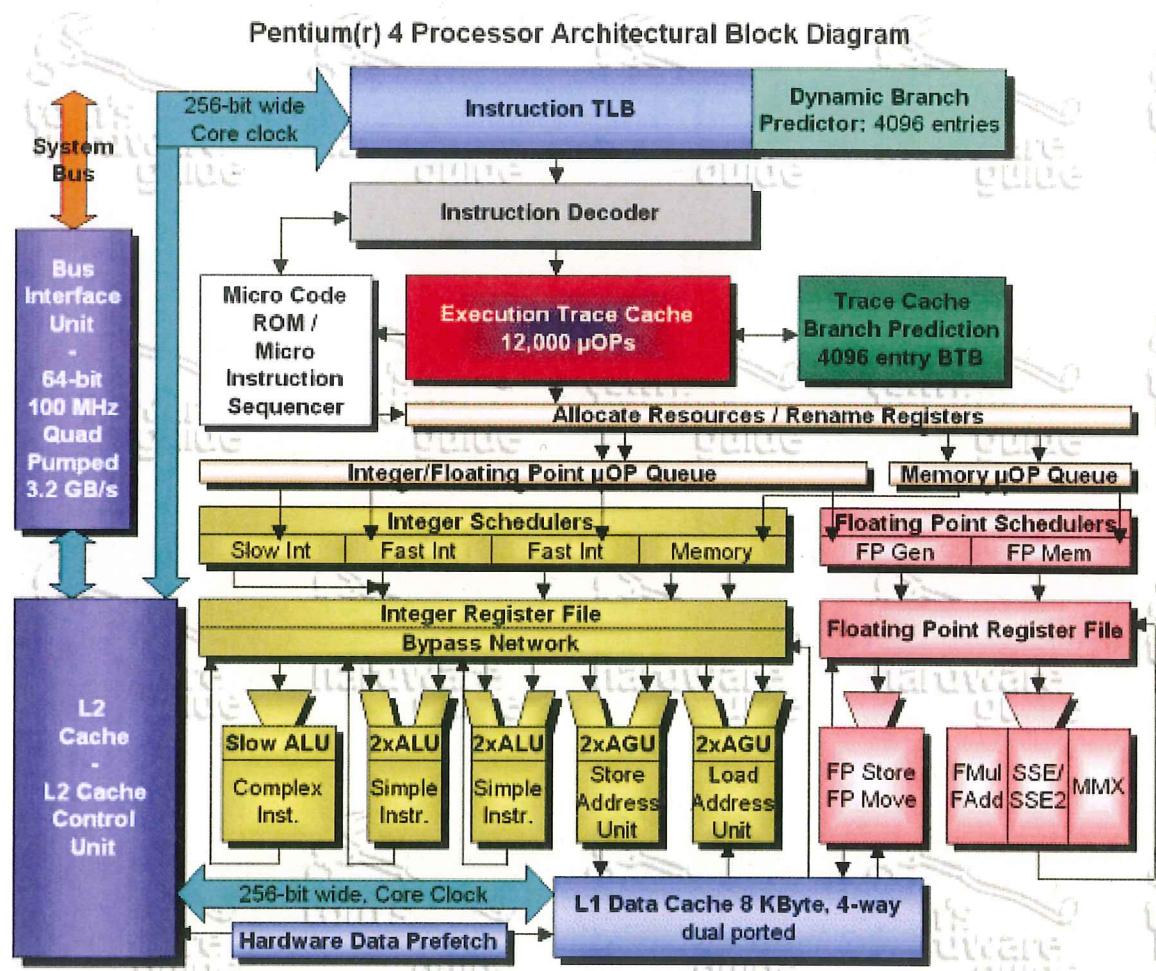
zie ook:

<http://www.tomshardware.com/2000/11/20/intel/index.html>

<http://www.anandtech.com/showdoc.aspx?i=1301&p=1>

Zie ook blokschema van de PIII vroeger in deze nota's.

5.1 Pentium 4 - Blokschema



Bron: Tomshardware

Terminologie:

- **Instruction TLB:** Buffer met de instructies zoals ze uit de L2 cache geprefetched zijn.
- **BTB:** Branch Target Buffer. Een tabel met de adressen waar naar gesprongen wordt in het programma.
- **μ OP: micro-operation.** Dit zijn (gedecodeerde) instructies die rechtstreeks door de execution units uitgevoerd kunnen worden. De decoder vertaalt elke x86 instructie in één of meer ‘fixed length’ μ OP, behalve wanneer de x86 instructie zo complex is dat er via de microcode ROM een complete sequentie μ OP’s wordt gegenereerd. Gemiddeld wordt een x86-instructie vertaald naar een 2-tal μ OP’s, maar complexe instructies als goniometrische FP-instructies kunnen vertaald worden in honderden μ OP’s.
- **AGU - Address Generation Unit.** Berekent de adressen waar operanden gelezen of opgeslagen moeten worden.

5.2 P4 Front Side Bus

De originele P4 had een 64-bit brede front side bus, geklokt aan 100 MHz (Quad Data Rate). Dit resulteerde in een debiet van $8 \times 4 \times 100 \text{ MHz} = 3,2 \text{ GB/s}$. Later werd de klokfrequentie opgedreven naar 133 MHz met als resultaat een debiet van 4,2 GB/s.

Deze FSB is gekoppeld met een On-chip L2 8 way Set associative cache van 2 MByte (origineel 256 kByte). De L2-cache is gekoppeld met de processorcore (L1 Data cache en instruction prefetcher) via een bus van 256 bits breed en met dezelfde klok als de processorcore (L2 cache latency is 7 klokcycli).

Voor data gebruikt de P4 een L1 cache van slechts 8 kB (4-way SA) met een zeer lage latency (2 klokcycli). Deze cache is bovendien dual-ported en maakt dus tegelijk één Load en één Store per klokcyclus toe.

5.3 Execution Trace Cache

De P4 maakt geen gebruik van een klassieke L1 cache voor de instructies. In de plaats worden de instructies eerst gedecodeerd en vertaald naar μ OP’s en deze laatsten worden in een ‘cache’ geplaatst, de ‘**execution trace cache**’.

Bij een klassieke L1 instruction cache moeten de instructies telkens opnieuw gedecodeerd worden als ze uit de cache gelezen worden? Dat is hier dus niet meer het geval. Vooral bij het gebruik van complexe instructies of wanneer een lus volledig in de execution trace cache past, geeft dit een grote snelheidswinst.

Overigens past de P4 ook binnen deze cache van 12.000 μ OPs branch prediction toe.

Dit mechanisme beperkt ook het risico dat de pipeline stilvalt door een situatie waarbij de decoders een bottleneck vormen.

Vanuit de execution trace cache worden gemiddeld per klokcyclus 3 µOps doorgegeven naar de volgende stap in de pipeline.

5.4 Hyperpipeline

Eén van de best gekende aspecten van de P4 Netburst architectuur is de extreem lange pipeline van 20 stappen (tegenover 10 voor de PIII en 11 voor de Athlon).

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP	TC Fetch	Drive Alloc		Rename	Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive			

De bedoeling van dit groot aantal stappen is om een zo hoog mogelijke klokfrequentie mogelijk te maken.

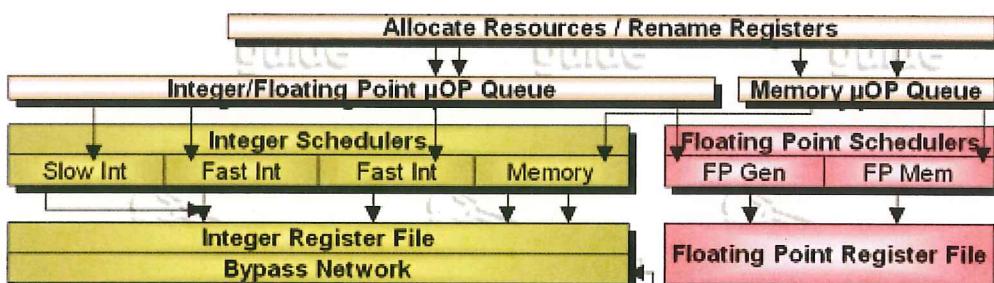
Uiteraard betekent dit ook dat een probleem in de pipeline (stall,...) grote gevolgen heeft: de P4 pipeline kan tot 126 instructies ‘in flight’ hebben.

Vanwege de dubbele branch-prediction zou (volgens Intel) een stall van de pipeline slechts zelden voorkomen. Het gedeelte tot en met de execution Trace cache beslaat de eerste 5 stappen van de pipeline.

5.5 Out of order execution

Bij het schedulen van de µOP’s voor uitvoering gebeuren volgende stappen:

- Toekenning van resources (execution units, registers)
- Bepalen van afhankelijkheden
- Register renaming (er zijn 128 registers fysiek beschikbaar ipv de 8 theoretische)
- µOP’s klaarzetten in de µOP queue
- De verwerking van de µOPs starten zodra hun execution unit beschikbaar is.



Er kunnen maximaal 6 µOP’s tegelijk gestart worden per klokcyclus.

5.6 Execution Units: ‘Rapid Execution Engine’



In het schema zie je de verschillende execution units van de P4. De kern hiervan wordt gevormd door de ‘rapid execution engine’.

Deze ‘rapid execution engine’ bestaat uit 2 integer ALU’s en 2 AGU’s die per klokcyclus elk 2 µOP’s kunnen verwerken. Gevolg hiervan is dat een aantal eenvoudige instructies slechts een halve klokcyclus nodig heeft.

De andere integer µOP’s (complexere bewerkingen) worden uitgevoerd in de tragere ALU (waarvan er maar één is). Gelukkig bevat de meeste assembler-code hoofdzakelijk eenvoudige instructies die dus zeer snel uitgevoerd worden.

Voor FP bewerkingen is er één load-unit, één store unit, één FP-ALU, één MMX-unit en één SSE-unit.

5.7 Besluit

De P4 Netburst-architectuur heeft als voornaamste doelstelling om een zo hoog mogelijke kloksnelheid mogelijk te maken door een lange pipeline, snelle execution units en de nodige logica om de problemen hierbij te beperken (branch prediction, execution trace cache, veel registers voor renaming,...). De IPC-waarde ligt hierbij lager dan bij andere architecturen zoals de PIII of de Athlon.

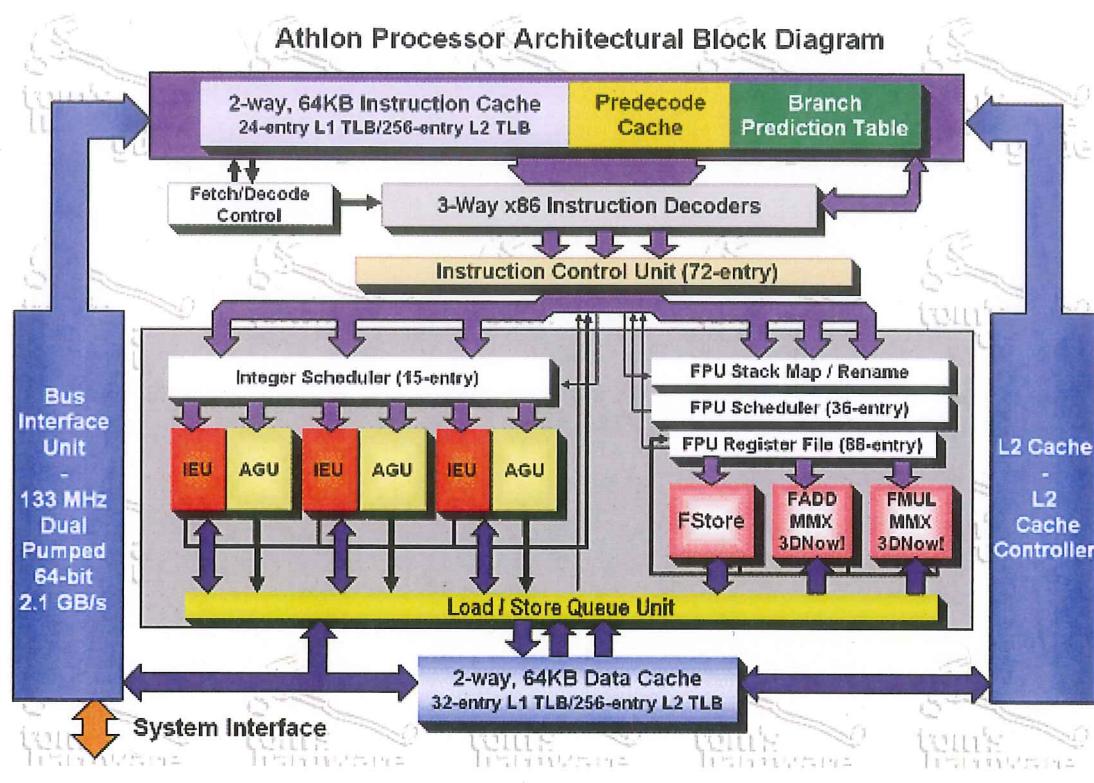
De P4 moet zijn performantie dus hoofdzakelijk uit zijn hoge kloksnelheid halen.

Wanneer de P4 vergeleken wordt met zijn voorganger (de PIII) aan een identieke kloksnelheid, blijkt de P4 in heel wat situaties trager te zijn!

Naast de interne logica is er nog een vereiste om de nadelen van een lange pipeline zo veel mogelijk te beperken: ook het externe geheugen moet data aan een voldoende hoge snelheid kunnen aanleveren om de processor op volle snelheid te laten werken. Vandaar het belang van de hoge throughput van de front-side bus, en het gebruik van geheugen dat snel genoeg is.

5.8 Athlon-Blokschema

Ter vergelijking het blokschema van de originele Athlon-processor (rechtstreekse concurrent van de P4).



Enkele opmerkelijke verschillen ten opzichte van de P4:

- Een klassieke zij het grote L1 cache ipc de execution trace cache.
- 3 parallel decoders: dit geeft de Athlon een grotere capaciteit voor het decoderen van instructies (compenseert gedeeltelijk de execution trace cache van de P4).
- Een kortere pipeline
- Meer execution units: 3 volwaardige integers units, 3 AGU's, beter bruikbare FP-units

Dit resulteert samen in een processor met een lagere kloksnelheid, maar een betere IPC dan de P4.

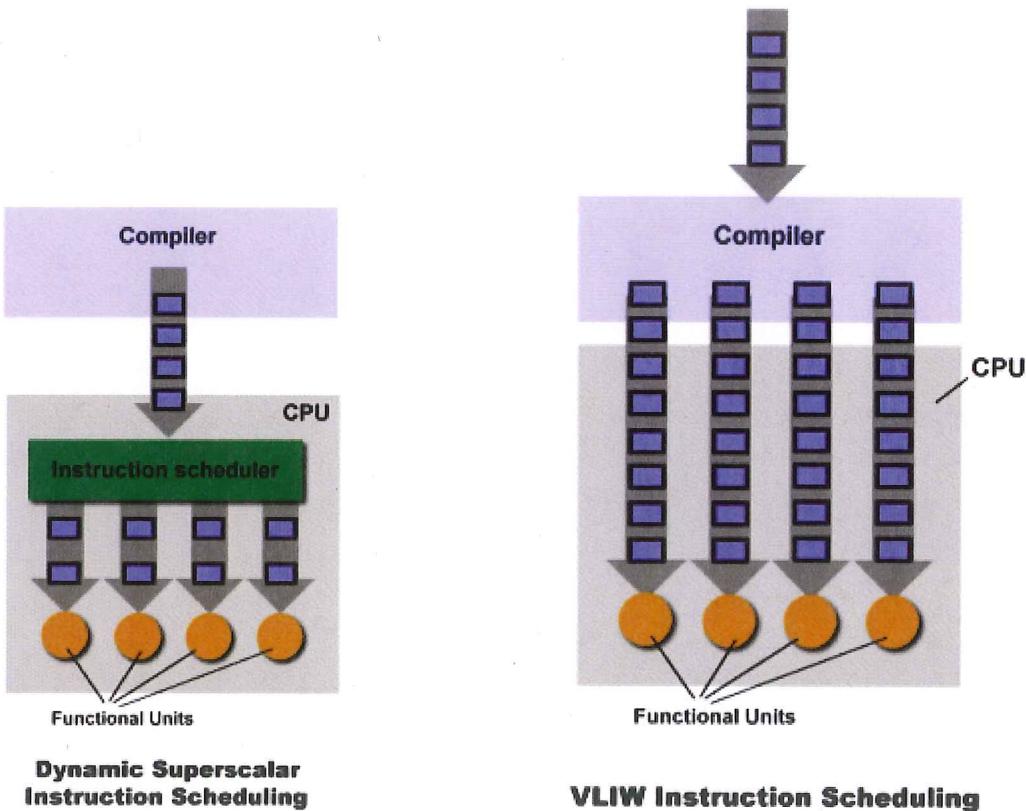
Hoofdstuk 6: 'Very Large Instruction Word' processoren

6.1 VLIW

Naarmate een superscalaire architectuur meer complex wordt, zullen ook de problemen bij de instructie-scheduling complexer worden. De schakelingen in de processor die instaan voor het bepalen van afhankelijkheden en voor het starten van de verwerking van instructies worden in dezelfde mate meer ingewikkeld (en duurder).

Je zou een moderne superscalaire processor kunnen beschouwen als een **dynamische ‘instructie-scheduler’**: de hardware beslist tijdens de uitvoering van een programma welke instructies er parallel uitgevoerd kunnen worden, welke er ‘out of order’ uitgevoerd mogen worden, waar er register renaming kan gebruikt worden, enz...

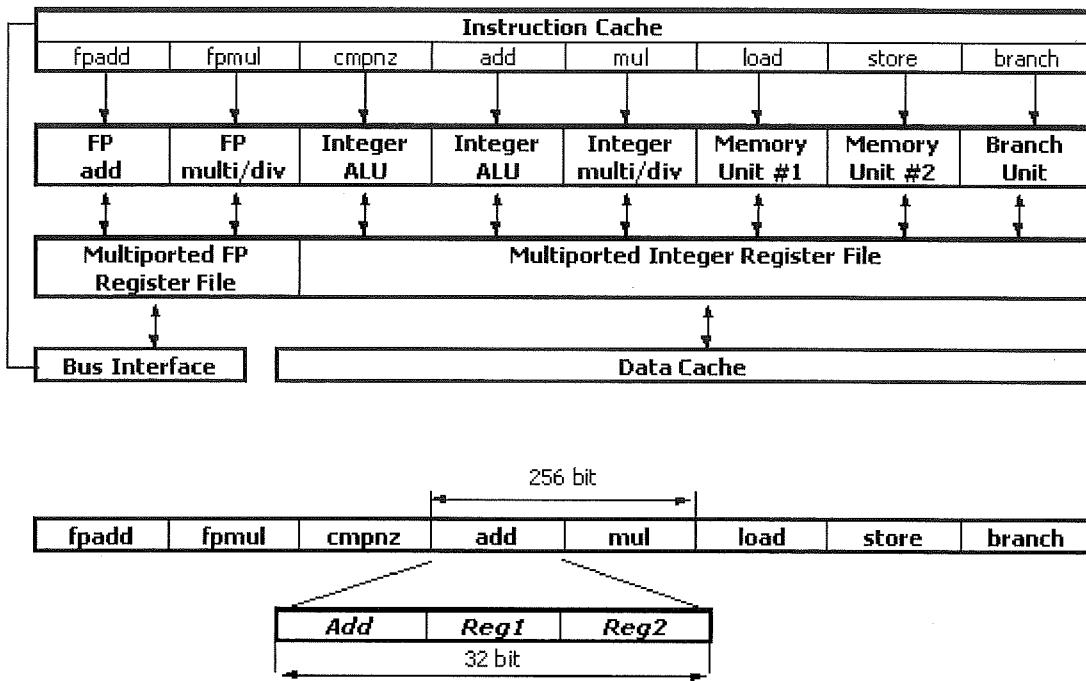
Een alternatieve aanpak zou zijn om deze beslissingen op voorhand door de compiler te laten nemen (**statisch** in plaats van dynamisch). Deze idee vormt de basis van de **VLIW** (**Very Long Instruction Word**) - computer.



bron: Ars Technica

VLIW-machines hebben (nogal voor de hand liggend) instructiecodes die veel langer zijn dan bij een traditionele processor. In één VLIW instructiewoord worden een aantal ‘traditionele’ instructies gecombineerd op een zodanige manier dat ze in de processor tegelijk uitgevoerd kunnen worden.

Een voorbeeld: veronderstel een processor met drie Integer Functional Units, twee Floating Point Functional Units, één Load/Store Unit en één branch unit. Een ‘instructie’ voor een dergelijke processor zou in het ideaal geval kunnen bestaan uit 8 bewerkingen van de verschillende klassen.



Daarbij is het de taak van de compiler om de geschikte bewerkingen op een zodanige manier te combineren dat er zoveel mogelijk bewerkingen in parallel kunnen uitgevoerd worden.

In praktijk zal dat uiteraard niet altijd lukken. Op de plaatsen waar geen geschikte bewerking gevonden wordt, zal dan een NOP-code komen.

Per definitie werkt een VLIW-processor met ‘**fixed-length**’ instructies. Intern zijn er een aantal parallele pipelines (8 in het voorbeeld), elk met een relatief eenvoudige decoder, gekoppeld aan zijn eigen specifieke execution unit.

Deze manier van werken betekent dat in de processor gemakkelijker superscalair gewerkt kan worden: er moet geen tijd of processorhardware meer besteed worden aan het analyseren en manipuleren van de instructiestroom: dat is immers al op voorhand door de compiler gedaan.

Het resultaat is een kleinere goedkopere processor die veel minder vermogen verbruikt om gelijkaardige prestaties te leveren als een klassieke RISC/CISC processor (in theorie tenminste).

Er zijn natuurlijk ook een aantal nadelen aan deze techniek:

- De **compilers** worden veel moeilijker en **complexer** om te schrijven.
- ‘**Code bloating**’: Wanneer de compiler niet in staat is op basis van de gevraagde bewerkingen voldoende geschikte instructies te combineren in één ‘very long instruction word’, moeten er extra NOP’s toegevoegd worden. Hierbij vergroot de hoeveelheid code, terwijl de verwerkingssnelheid en efficiëntie daalt..
- Een **verandering** aan de **processorhardware** (bijvoorbeeld het toevoegen of weglaten van extra functional units) betekent dat ook de compiler moet aangepast worden en dat alle code opnieuw gecompileerd moet worden.

Enkele ‘general purpose’ voorbeelden van VLIW-machines zijn de TRANSMETA- Crusoe processoren en de IA64 familie die gezamenlijk door HP en INTEL ontwikkeld zijn. Enkel Intel heeft op basis van de IA64 architectuur de Itanium processorfamilie commercieel uitgebracht. Hierbij gebruikt Intel de term EPIC (Explicit Parallel Instruction Computer) in plaats van de VLIW.

Daarnaast heeft de VLIW-techniek vooral zijn plaats gevonden in embedded en gespecialiseerde DSP-controllers.

6.2 De Intel Itanium

Zie ook: <http://arstechnica.com/articles/paedia/cpu/ia-64-preview.ars/1>

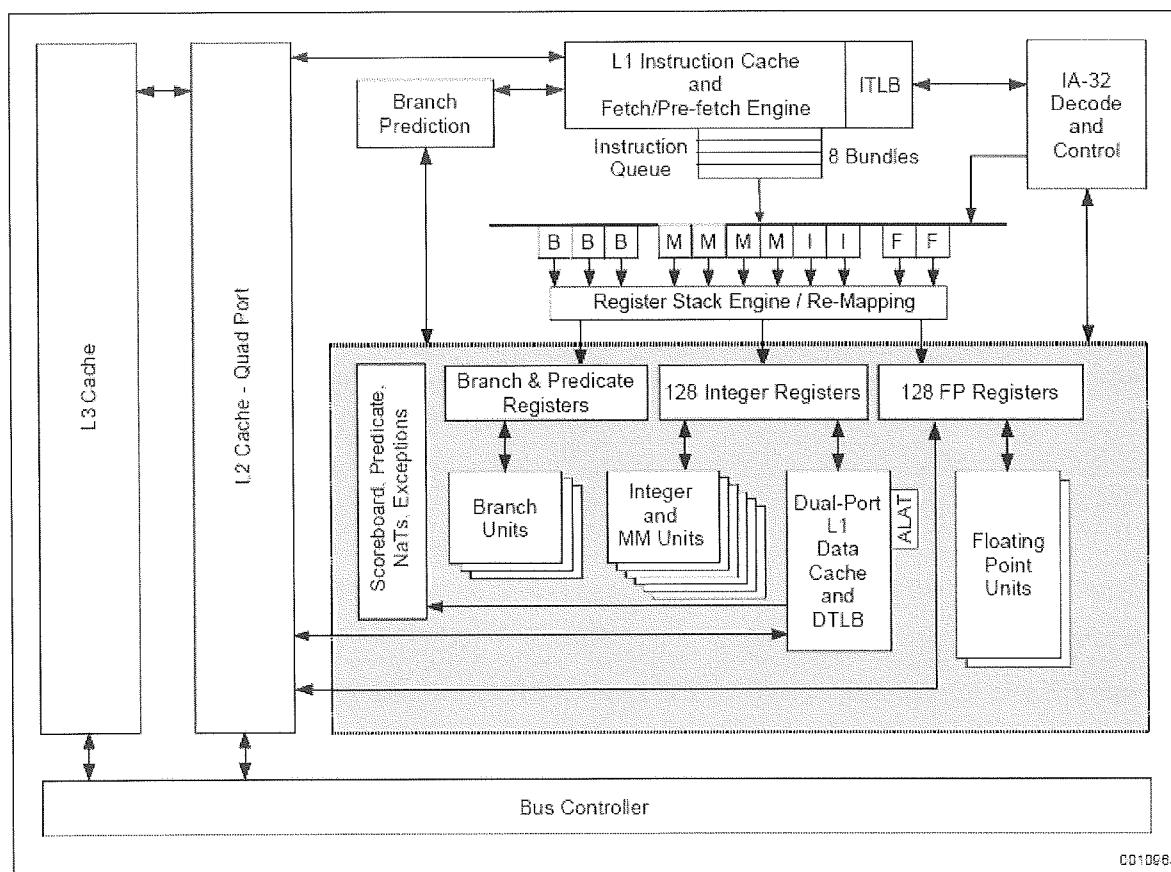
De ‘Explicitly Parallel Instruction Computing (EPIC)’ technology is gebaseerd op de VLIW-filosofie, maar combineert die met een aantal aspecten van klassieke RISC/CISC processoren. Om de architectuur commercieel aantrekkelijker te maken, heeft Intel ook intern de mogelijkheid voorzien om klassieke x86 instructies te verwerken (maar dan wel met een lage performantie).

De Itanium werkt met instructiewoorden van 128 bits, die telkens drie instructies bevatten naast nog wat extra informatie.

De Itanium bevat 6 pipelines van elk 8 stappen diep en werkt met een klok van 1,5 GHz.

De execution units bestaan uit 6 integer units, 6 multimedia units, 2 load en store units, 3 branch units en 4 floating point units.

Per klokcyclus haalt de processor 2 ‘instruction bundles’ op. Elke instruction bundle bevat 3 instructies samen met extra informatie over die instructies (o.a. in welke instruction unit ze uitgevoerd moeten worden).



Bron: Intel

Om de efficiëntie op te drijven en spronginstructies zoveel mogelijk te vermijden, bestaat de mogelijkheid om instructies enkel uit te voeren als bepaalde condities voldaan zijn.

De Itanium-processor is door Intel vooral bedoeld voor de markt van high-end werkstations en servers.

Vanwege de sterk geëvolueerde performantie van de klassieke x86 architecturen en de uitbreiding daarvan naar 64-bit, is de positie van de Itanium heel wat twijfelachtiger geworden.

6.3 De Transmeta-Crusoë

Zie: <http://arstechnica.com/articles/paedia/cpu/crusoe.ars/1>

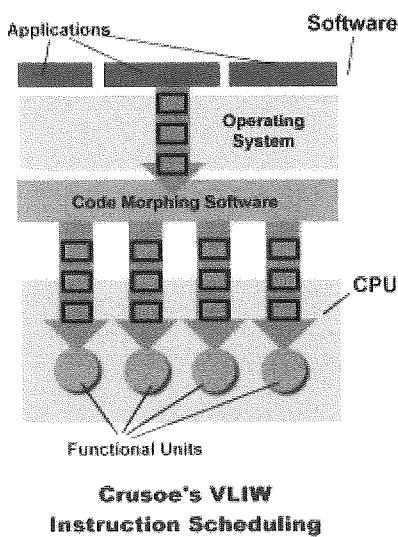
Het bedrijfje Transmeta heeft de VLIW-technologie op een totaal andere manier gebruikt in het ontwerp van zijn processoren. Het was de bedoeling een x86-compatibele processor te ontwerpen die vooral geschikt was in draagbare toepassingen.

Belangrijkste doelstellingen waren dus:

- x86 compatibiliteit
- laag verbruik
- architectuur soepel aan te passen naar betere performantie.

De kern van de Crusoe wordt gevormd door een 128-bit VLIW processor die in staat is om per klokcyclus 4 instructies in te lezen en te verwerken. De originele versie bevatte 2 integer units, één FP-unit, een Load/Store unit en één Branch-unit. De ‘instruction bundles’ van Intel worden hier ‘molecules’ genoemd waarbij elke molecule bestaat uit 4 ‘atoms’ of bewerkingen.

Om de processor x86 compatible te maken bevat de processor een stukje ingebakken software als frontend: de ‘code-morpher’ die voor een dynamische vertaling zorgt van de extern opgeslagen x86 instructies naar de native VLIW-instructies.



Dit alles wordt aangevuld met speciale power-management technologie die de kloksnelheid en core-spanning aanpast aan de benodigde rekenkracht ('Longrun' technologie).

Het resultaat was een processor met een verbruik van 1 Watt aan een klokfrequentie van 700 MHz, tegenover een verbruik van 20 tot 40 watt bij de klassieke processoren aan dezelfde snelheid.

Het gebruik van Code-Morphing maakt het mogelijk dat de kern van de processor gewijzigd wordt, zonder dat de externe software daar last van heeft.

Leuk detail: Linus Torvalds, die aan de basis lag van het Linux-OS, is één van de ontwikkelaars van de Crusoë.

Hoofdstuk 7: Andere vormen van parallelisme

7.1 Indeling architecturen: SISD, SIMD MISD, MIMD

Bij de bespreking van de architectuur van processoren zijn er een heleboel manieren waarop die ingedeeld kunnen worden. Toch is er een klassieke indeling met termen die toch eventjes kort besproken moeten worden:

SISD: Single Instruction, Single Data

Dit is de klassieke processor waarbij tegelijk één instructie uitgevoerd wordt op één eenheid informatie. Ook pipelined en superscalaire processoren worden in deze categorie gerekend omdat deze technieken niets wijzigen aan de logica van de programma's die erdoor verwerkt worden.

SIMD: Single Instruction, Multiple Data

Hierbij wordt eenzelfde bewerking tegelijkertijd uitgevoerd op een aantal afzonderlijke data-objecten. (bv. optelling van 2 matrices).

Dit is bijvoorbeeld het geval bij de MMX-instructies (3DNow, SSE,...) op de Intel IA32-familie, de Altivec op de PowerPC, maar ook bij zogenaamde **vectorprocessoren**.

MISD: Multiple Instruction, Single Data

In praktijk zijn er eigenlijk geen computers van belang ontworpen die op dit principe gebaseerd zijn.

MIMD: Multiple Instruction, Multiple Data

Dit is het klassieke geval van een **multithreaded systeem** waarbij verschillende threads onafhankelijk van elkaar bewerkingen uitvoeren op verschillende stukken informatie. Ook processoren die hyperthreading ondersteunen horen eigenlijk in deze categorie thuis.

7.2 Vectorinstructies (SIMD)

In sommige toepassingen (DSP, wiskundige bewerkingen op matrices, ...) moet dezelfde bewerking toegepast worden op een hele reeks gegevens. Denk daarbij bijvoorbeeld aan het toepassen van een filter in Photoshop, of het renderen van een 3D-beeld. Dit soort bewerkingen komt vooral bij multimedia-toepassingen voor. Het is dan ook bij de opkomst van dit soort toepassingen dat de ontwerpers speciale uitbreidingen in de processorhardware begonnen te voorzien om bepaalde bewerkingen tegelijkertijd op meerdere registers te laten gebeuren.

7.2.1 MMX

De MMX-uitbreiding werd voor het eerst in 1997 als een uitbreiding bij de Pentium-processor gerealiseerd.

In praktijk betekent MMX dat er 8 registers van 64-bit bijkomen (in de eerste versie werd hiervoor dezelfde hardware gebruikt als de FP-registers: MMX-instructies en FP-instructies konden niet door elkaar gebruikt worden).

Deze registers kunnen gebruikt worden voor integer bewerkingen op 1 x 64 bit (1 QWORD), 2 x 32 bit (2 x DWORD), 4 x 16-bit (4xWORD) of 8 x 8 bit (8 x byte).

Deze bewerkingen kunnen ‘saturation arithmetic’ gebruiken, typisch bij DSP-toepassingen.

Deze MMX-uitbreiding was vooral handig voor audio-bewerking en grafische bewerkingen (2D en 3D matrices). Deze laatste bewerkingen worden tegenwoordig veel sneller uitgevoerd in de GPU (zie later).

7.2.2 3DNow!

Een jaar na de invoering van MMX door Intel, kwam AMD met een uitbreiding hierop onder de naam ‘3DNow!’: hierbij waren floating-point bewerkingen mogelijk op 2 x 32-bit. Dit was vooral voor AMD belangrijk omdat door de opkomst van 3D-graphics en gaming het belang van FP-berekeningen enorm toenam en de Athlon op dat gebied niet erg sterk was.

3DNow werd verschillende malen aangepast en uitgebreid. Vanwege het gebrek aan compatibiliteit met de Intel-processoren werd vanaf de Athlon XP de SSE-uitbreiding van Intel geïntegreerd in ‘3DNow! professional’.

7.2.3 SSE (Streaming SIMD Expressions)

Met het toenemende belang van de FP-berekeningen en de migratie van integer graphics berekeningen naar de videokaart werd het voordeel van de MMX-uitbreiding snel uitgehouden. Als antwoord kwam Intel in 1999 (Pentium III) met een tweede uitbreiding die de nadelen van MMX wegwerkte: de SSE-uitbreiding:

Hierbij worden 8 extra 128-bit registers voorzien (deze keer los van de FP/MMX-registers). Hierin kunnen FP-bewerkingen uitgevoerd worden op 4 single precision 32-bit FP getallen.

SSE is ondertussen ook verder geëvolueerd (tot SSE3) met ondersteuning van integer bewerkingen en extra fixed-point en floating-point bewerkingen.

7.2.4 Altivec (Velocity engine, VMX)

Onder deze naam werd door Apple, IBM en Freescale Semiconductor (vroeger Motorola) een SIMD uitbreiding voorzien voor de PowerPC (vanaf de G4). Ze is vergelijkbaar met SSE van Intel, maar met relatief grote verschillen in de beschikbare instructies en mogelijkheden. Ook in de XENON-processor (XBOX360) en de CELL-processor (PS3) worden versies van VMX gebruikt

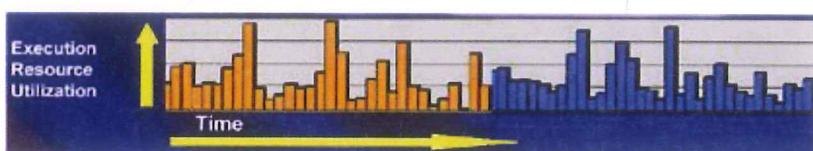
7.3 Thread level Parallelisme (TLP): Hyperthreading

Zelfs met alle technieken uit de voorgaande hoofdstukken is het nog steeds niet te vermijden dat er af en toe delen (functional units) van de processor ongebruikt blijven.

De techniek van ‘Hyperthreading’ of ‘Simultaneous Multithreading’ gaat nog een stap verder in het gebruik van parallelisme.

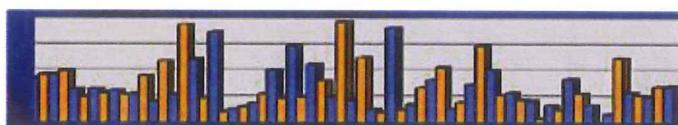
Een klassieke processor is ‘single-threaded’: op een willekeurig ogenblik kan slechts één thread verwerkt worden. Hierbij wordt de toestand van het programma bepaald door de inhoud van de verschillende processorregisters (de ‘Architectural State’).

Meerdere programma’s/tasks kunnen daarbij de een na de ander uitgevoerd worden in een single-threaded besturingssysteem:



Wanneer het besturingssysteem multithreading op één enkele processor toelaat, dan worden eigenlijk de verschillende threads om beurt eventjes verwerkt.

Het overschakelen tussen twee threads gebeurt door het besturingssysteem dat de processor-toestand van de verschillende threads bijhoudt.



Om met dergelijke klassieke processoren tegelijkertijd meerdere threads te verwerken, moet met meerdere processoren gewerkt worden.

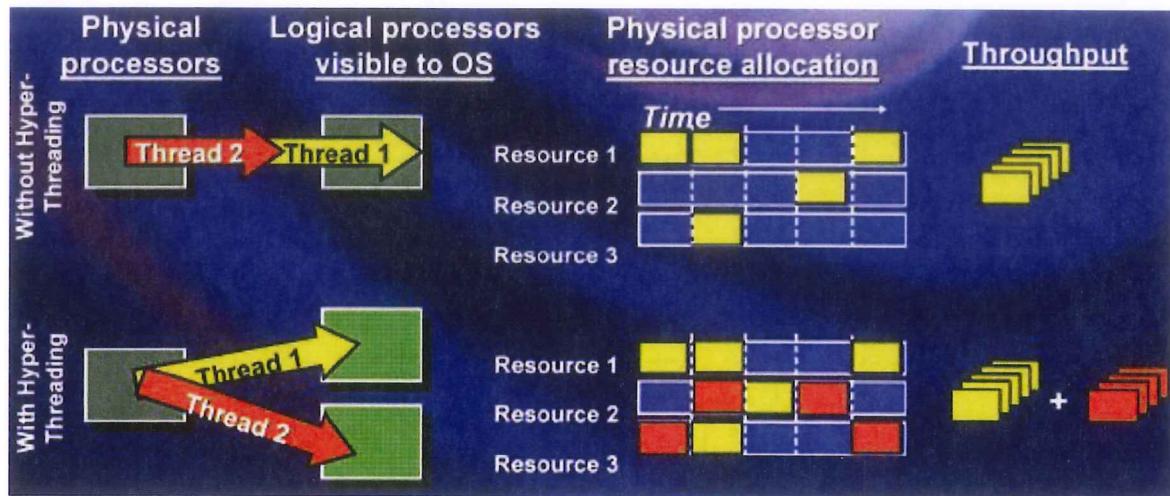
‘Hyperthreading’ is er op gebaseerd dat één **fysieke** processor zich naar het besturingssysteem (en de gebruiker) voordoet als twee **logische** processoren.

Daarvoor zijn in de processor twee sets van registers beschikbaar (met onder andere twee Instruction Pointers). Hiervoor is eigenlijk relatief weinig extra hardware nodig. De processor voert tegelijk twee threads uit die onderling concurreren voor de beschikbare resources. De functional units van de processor zijn niet ontdubbeld!

Op deze manier zal een deel van de resources die door de ene thread niet gebruikt kunnen worden, mogelijk gebruikt kunnen worden binnen de tweede thread.



Hoewel door deze techniek uiteraard niet dezelfde performantie gehaald wordt als bij een dual-processor systeem, zal de processor toch efficiënter gebruikt worden dan zonder multi-threading:

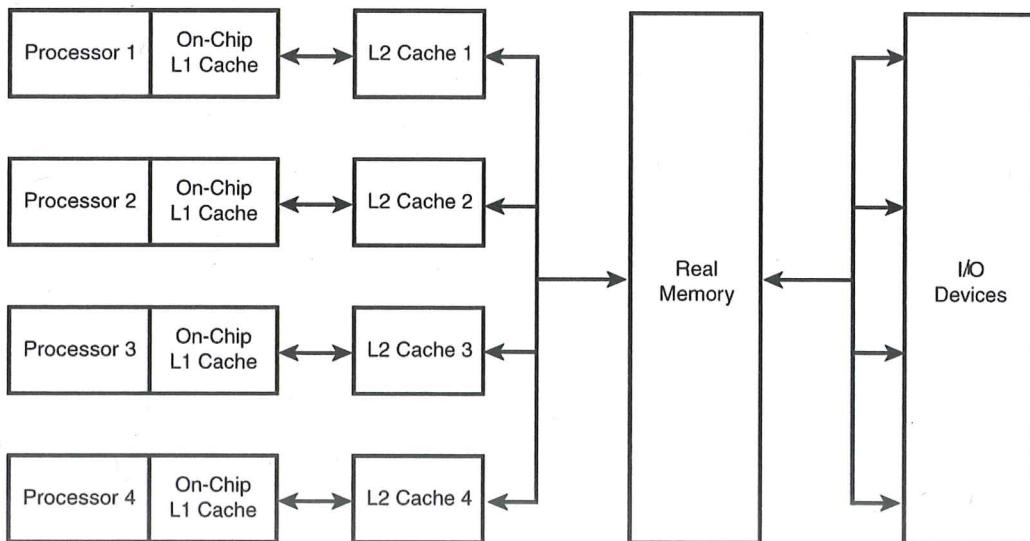


In praktijk zal de snelheidswinst zeer sterk afhangen van het feit of er meerdere threads beschikbaar zijn om tegelijk uitgevoerd te worden evenals van de vraag of die threads elkaar goed kunnen aanvullen bij het gebruik van de processorresources.

7.4 SMP: Symmetric multiprocessing

Wanneer meer performantie nodig is kan overgestapt worden naar het gebruik van meerdere processoren in parallel. In praktijk gebeurt dit meestal onder de vorm van '**Symmetric multiprocessing (SMP)**'. Hierbij maken verschillende processoren gebruik van een gemeenschappelijk geheugensysteem.

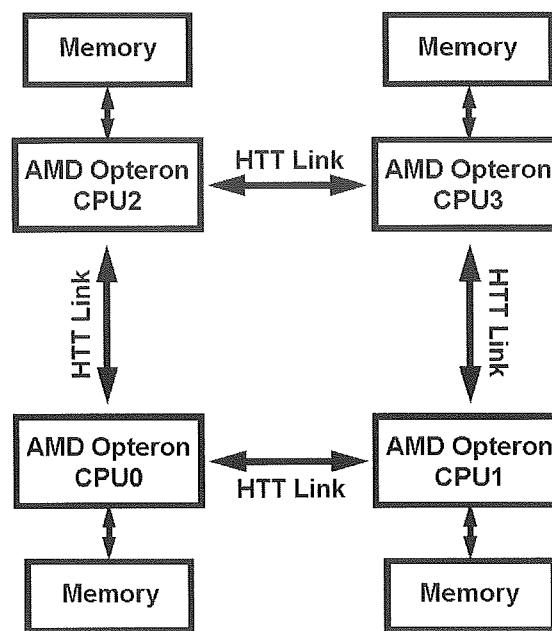
Dit maakt het eenvoudig mogelijk verschillende threads van één proces op verschillende processoren uit te voeren.



Hierbij zijn er twee grote problemen:

- Bandbreedte van het geheugen: traditioneel is het geheugen altijd trager geweest dan de processoren. Dit betekent dat een processor regelmatig moet wachten op het tragere geheugen. Wanneer er meerdere processoren gebruik maken van hetzelfde geheugen zal dit probleem enkel erger worden.
- Cache-consistentie: om de traagheid van het geheugen te compenseren, wordt o.a. gewerkt met cachegeheugens. hierbij wordt er een kopie van een deel van het trage centraal geheugen bewaard in het snellere cachegeheugen van de processor.
Wanneer een processor bij een schrijfoperatie de inhoud zijn cache wijzigt, moet de hardware ervoor zorgen dat alle andere processoren op de hoogte gebracht worden van die wijziging. Als dit niet zou gebeuren, bestaat het risico dat één van de andere processoren bewerkingen uitvoert met verouderde en dus foutieve data.

Als alternatief voor SMP bestaat er ook een multiprocessor-technologie waarbij elke processor zijn eigen geheugen systeem heeft (**Non-Uniform Memory Architecture**, NUMA). Een voorbeeld:



Dit lost beide bovenstaande problemen op, maar creëert weer andere:

- Verschillende threads van één proces kunnen niet tegelijk op verschillende processoren uitgevoerd worden: de data en instructies van één proces worden namelijk in één gemeenschappelijke geheugencontext bijgehouden.
- De communicatie tussen verschillende processen wordt veel ingewikkelder en trager omdat er een mechanisme moet voorzien worden voor uitwisseling van data tussen de verschillende geheugensystemen.

7.5 Multicore-processoren

Vanwege de steeds verder gaande miniaturisering en de problemen bij het zoeken naar steeds beter instruction level parallelisme zijn de grote processorfabrikanten overgegaan tot het implementeren van meerdere cores (2 tot 10-tallen) in één processor.

Hierbij komen overigens dezelfde problemen als vermeld bij SMP-systemen opnieuw de kop opsteken: concurrentie tussen de 2 cores voor toegang tot het geheugen, cache consistentie met het bijkomend probleem van de beperkte bandbreedte van de frontside bus voor koppeling met het geheugen.

Als een algemene trend zal dan ook de bandbreedte tussen de cores en het geheugen verhoogd moeten worden.

Opmerking:

Door de evolutie van de architectuur van de processoren blijkt het effect van nieuwere veranderingen steeds meer afhankelijk van de **kwaliteit** van de **software** die uitgevoerd wordt.

In eerste instantie speelt de mate waarin code **geoptimaliseerd** is voor een specifieke processor een belangrijke rol.

Daarnaast is het door de recente ontwikkelingen ook van zeer groot belang wat de algemene structuur is van de software: concreet is het belangrijk dat het rekenwerk zo goed mogelijk verdeeld is tussen **verschillende threads** die onafhankelijk van elkaar kunnen werken.

In praktijk is dit laatste niet altijd even eenvoudig.

7.6 Asymmetric multiprocessing (AMP)

In een SMP- of NUMA-systeem wordt ‘symmetrisch’ gewerkt: alle processoren zijn gelijk-aardig en een willekeurig proces kan op gelijk welke van de beschikbare processoren uitgevoerd worden.

Daar tegenover staat **Asymmetric Multiprocessing (AMP)**: daarbij hebben afzonderlijke processoren een specifieke taak. Zo kan bv. één processor gebruikt worden om het besturingssysteem uit te voeren, een andere om gebruikersprogramma’s uit te voeren, enz...

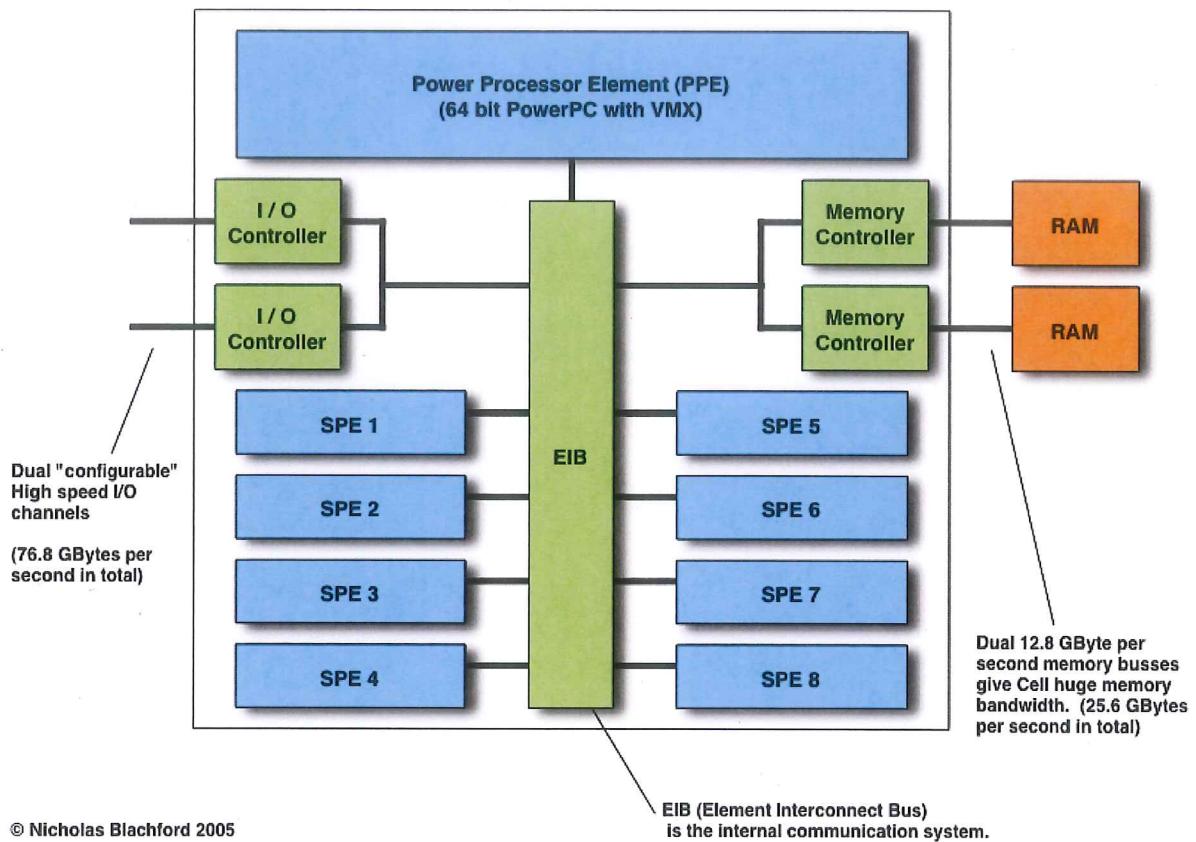
Eventueel kan zelfs het hardware-ontwerp hierop afgestemd zijn (met verschillende types processoren voor verschillende taken).

Een speciaal geval van AMP is het concept waarbij **gespecialiseerde processoren** specifieke taken overnemen van de normale ‘general purpose’ CPU’s. Voorbeelden hiervan zijn: Digitale audioprocessoren, moderne videokaarten (zie Appendix A), ‘TCP Offload Engines (TOE)’, de PhysX kaart van AGEIA (<http://anandtech.com/video/showdoc.aspx?i=2751>) ...

7.7 Case-study: de Cell-processor

7.7.1 Overzicht

Cell Processor Architecture



- PPC General purpose core (met klassieke L1 en L2 cache)
- n x ‘Synergistic Processor Elements’
- Element Interface bus
- dual channel memory controller
- Dual channel I/O-controller
- kloksnelheden .. 4GHz

7.7.2 general purpose core (Power PC)

Een ‘light’ general purpose-processor: bedoeld voor besturingssysteem, algemene taken, en coördinatie van SPE’s.

Compleet nieuw ontwerp, afwijkend van bestaande PPC’s (well binair compatible).

2 Issue, in-order: nipt superscalair, maar niet gesofisticeerd, zeer beperkte branch-prediction (veronderstelt dus intelligentie in compiler ter compensatie).

Dual-threaded (ook ter compensatie, vergelijkbaar met Intel’s hyperthreading)
Korte pipeline, toch bedoeld voor hoge clockrates ter compensatie van lage IPC.

Conclusie:

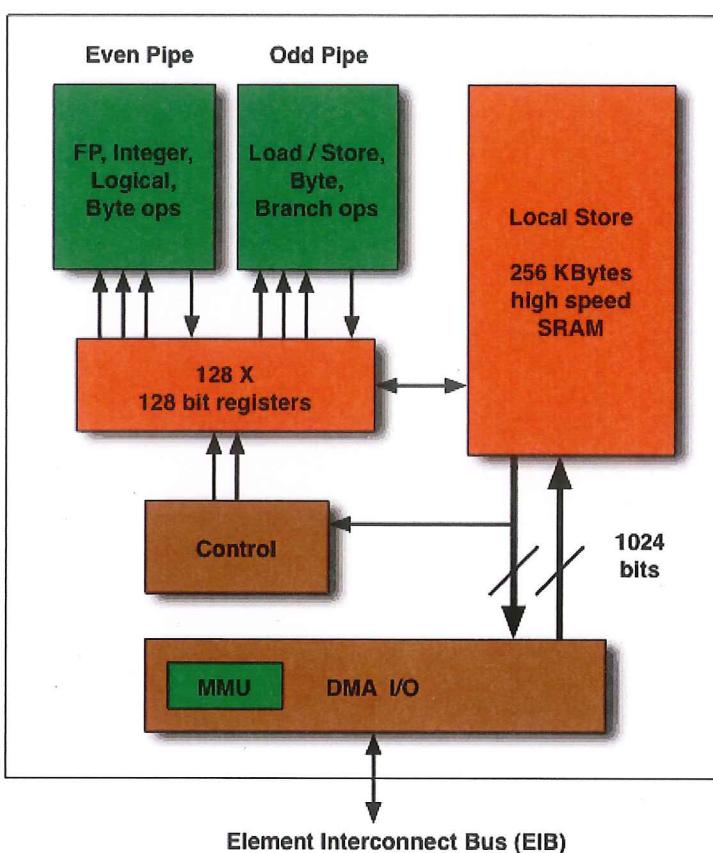
Geen sterk punt, niet noodzakelijk nadeel, maar vereist nieuwe expertise van programmeurs of zeer intelligente compilers (o.a. voor ‘sheduling’ van 2-issue instructies)

7.7.3 Synergistic Processor Elements (SPE)

4 tot 16 SPE's: gespecialiseerde Floating Point DSP-georiënteerde cores (nu 8)

Cell SPE Architecture

Each SPE is an independent vector CPU capable of 32 GFLOPs or 32 GOPs (32 bit @ 4GHz.)



© Nicholas Blachford 2005

SPE = eigenlijk autonoom zeer specifiek systeem.

Compleet vector-georiënteerd: 128-bit = 4 x single FP = (2 x double FP)

2-issue (at best), in-order, geen branch prediction, korte pipeline

local store: eigen 256 KB zeer snelle SRAM: vervangt cache

Memory-controller naar EIB ~zelf te programmeren (theoretisch 64GB/s).

128 x 128-bit registers.

ISA = hardware ! dus alles in handen van de programmeur.

conclusie:

Niet gesofisticeerd, back to basics, maar wel SIMD.

Efficiëntie hangt dus zeer sterk af van kwaliteit van de code: vereist zeer goede assembler programmeur/zeer goede compiler (haalbaar?), maar compleet nieuwe expertise op zeer laag niveau (microcode of assembler).

7.7.4 Element Interface Bus (EIB)

EIB : cross-connect voor intern datatransport tussen I/O, extern geheugen, PPE-cache en SPE local-stores.

4 x 16bit brede ring, gekloktd aan helft van processorklok,
laat gelijktijdig drie transfers toe op de ring

theoretisch 384 GB/s !!

externe geheugenbus @ 25 GB/s

externe I/O-bus @ 76 GB/s

Ter vergelijking: Core2Duo: 8.2 GB/s, Opteron: 1 of 2 x 6,4 GB/s

7.7.5 Modellen voor gebruik van de Cell

Problemen om de rekenkracht van de Cell te kunnen benutten:

- ‘probleem vinden voor de oplossing’
- TLP programmeren op dergelijke schaal
- management van de 8 cores (door de GP-core)
- memory bandwidth (25 GB/s voor 9 cores)
- vereiste competentie van programmeur

Enkele modellen:

Task Level parallelisme:

Elke SPE voert een andere taak uit: decryptie van data, user interface, audio-stream, video-stream...

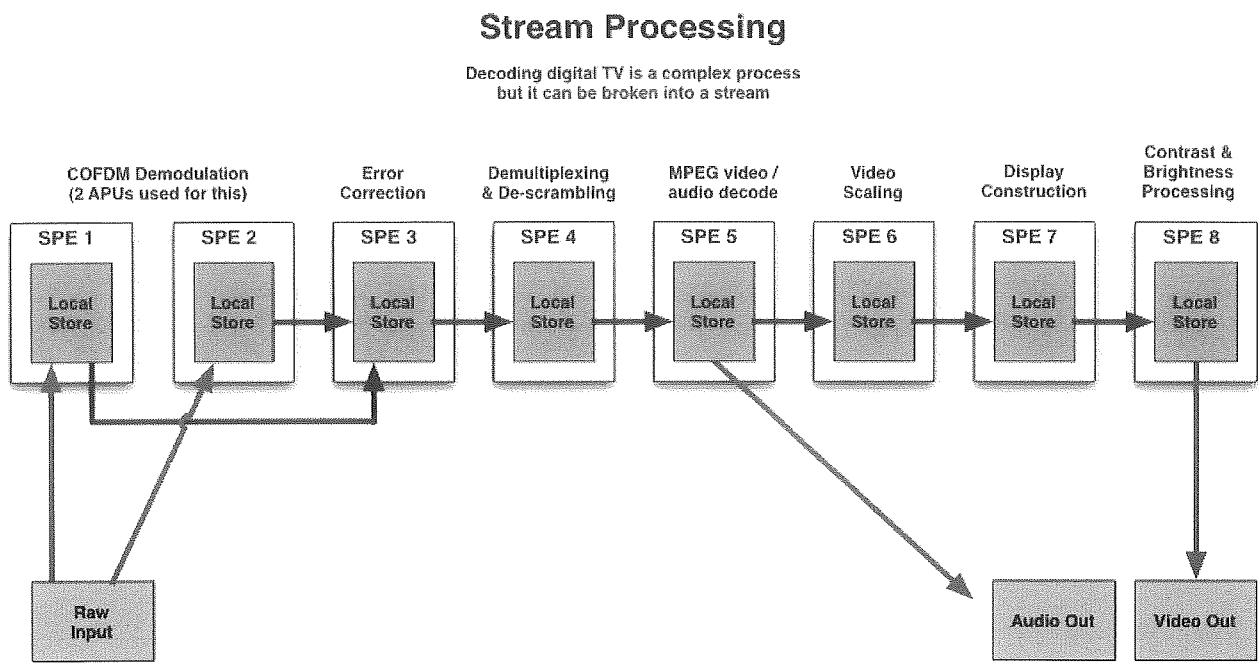
Collaboration:

Elke SPE voert dezelfde taak uit, maar op een ander stukje data: rendering, ray-tracing, MPG-compressie...

Cascade / pipeline:

Stream-processing: output van de ene SPE is input van de volgende...

Voorbeeld: decoding van HD-TV (zie volgende pagina).



© Nicholas Blachford 2005

referenties:

- http://www.gamezero.com/team-0/articles/interviews/dr_h_peter_hofstee/
- http://www.stanford.edu/class/ee380/Abstracts/Cell_060222.pdf
- <http://www.anandtech.com/showdoc.aspx?i=2379&p=1>
- <http://arstechnica.com/articles/paedia/cpu/cell-1.ars>
- <http://arstechnica.com/articles/paedia/cpu/cell-2.ars>
- <http://arstechnica.com/news.ars/post/20050502-4872.html>
- <http://arstechnica.com/news.ars/post/20060225-6265.html>
- http://www.blachford.info/computer/Cell/Cell0_v2.html
- <http://www.hpcwire.com/hpc/679134.html>