Processorarchitectuur

2ICT

Michiel Van den Berghe

2015

Inhoud

[1 Overzicht werking processor 3](#_Toc441586035)

[1.1 Uitvoeren van programmacode 3](#_Toc441586036)

[1.2 Het formaat van een processorinstructie 4](#_Toc441586037)

[1.3 Verwerking van een instructie 4](#_Toc441586038)

[1.4 Instuction Set Architecture 5](#_Toc441586039)

[2 Basisarchitectuur van een processor 8](#_Toc441586040)

[2.1 Interne architectuur van een processor 8](#_Toc441586041)

[2.2 Basisschema processor 8](#_Toc441586042)

[2.3 Controle-Unit met hardware decoder 10](#_Toc441586043)

[2.4 Microprogrammatie 10](#_Toc441586044)

[2.5 Complex Instruction Set Computers (CISC) 11](#_Toc441586045)

[2.6 Reduces Instruction Set Computers (RISC) 11](#_Toc441586046)

[3 Pipelining 12](#_Toc441586047)

[3.1 Model van een pipelined RISC architectuur 13](#_Toc441586048)

[3.2 Pipeline ‘hazards’ 14](#_Toc441586049)

[4 Superscalaire processoren 15](#_Toc441586050)

[4.1 Superscalaire architecturen 15](#_Toc441586051)

[4.2 Een eenvoudig superscalair model 16](#_Toc441586052)

[4.3 Out of Order execution 17](#_Toc441586053)

[4.4 Window of execution 17](#_Toc441586054)

[4.5 Afhankelijkheden 17](#_Toc441586055)

[5 Vergelijking P4 ‘Netburst’ – Athlon 18](#_Toc441586056)

[5.1 Pentium P4 – Blokschema 18](#_Toc441586057)

[5.2 P4 Front Side Bus 18](#_Toc441586058)

[5.3 Execution Trace Cache 19](#_Toc441586059)

[5.4 Hyperpipeline 19](#_Toc441586060)

[5.5 Out of order execution 19](#_Toc441586061)

[5.6 Execution Units: ‘Rapid Execution Engine’ 19](#_Toc441586062)

[5.7 Besluit 20](#_Toc441586063)

[5.8 Athlon – Blockschema 20](#_Toc441586064)

[6 ‘Very Large Instruction Word’ processoren 21](#_Toc441586065)

[6.1 VLIW 21](#_Toc441586066)

[6.2 De Intel Itanium 21](#_Toc441586067)

[6.3 De Transmeta-Crusoë 21](#_Toc441586068)

[7 Andere vormen van parallellisme 22](#_Toc441586069)

[7.1 Indeling architecturen: SISD, SIMD, MISD, MIMD 22](#_Toc441586070)

[7.2 Vectorinstructies (SIMD) 22](#_Toc441586071)

[7.3 Thread Level Parallelisme (TLP): Hyperthreading 23](#_Toc441586072)

[7.4 Symmetric MultiProcessing (SMP) 23](#_Toc441586073)

[7.5 Multicore-processoren 23](#_Toc441586074)

[7.6 Asymmetric MultiProcessing (AMP) 23](#_Toc441586075)

[7.7 Case Study: De Cell-processor 24](#_Toc441586076)

[8 GPU 26](#_Toc441586077)

[8.1 3D graphics rendering 26](#_Toc441586078)

[8.2 Geometrie 26](#_Toc441586079)

[8.3 Rendering 27](#_Toc441586080)

[9 GPGPU 28](#_Toc441586081)

[9.1 Nvidia Geforce 6 28](#_Toc441586082)

[9.2 Unified Shader Model (>= DirectX 10) 28](#_Toc441586083)

[9.3 GP GPU processing 28](#_Toc441586084)

[10 High performance computing 28](#_Toc441586085)

[11 Power management in processoren 29](#_Toc441586086)

[11.1 Redenen voor power management 29](#_Toc441586087)

[11.2 Op niveau van CPU (hardware) 29](#_Toc441586088)

[11.3 Op niveau van OS (Software) 30](#_Toc441586089)

# Overzicht werking processor

## Uitvoeren van programmacode

In de processor worden instructies verwerkt waaruit het besturingssysteem en de toepassingen zijn gebouwd. Dit zijn een beperkt aantal relatief eenvoudige instructies, deze staan als binaire informatie opgeslagen in het geheugen. De geschreven code moet hiernaar omgezet worden.

### Managed code: Java, C#, VB.Net

**Werking**

Code🡪 Compiler🡪 Tussencode🡪 Interpreter (Sandbox-omgeving)🡪 CPU-instructies en services van OS

**Gevolgen**

* Extra bescherming door interpreter, geheugenbeheer gebeurt automatisch (performantiewinst)
* Source code en tussencode kunnen niet rechtsreeks op CPU runnen
* Interpreter bestaat uit native processorinstructies
* Elke instructie in een lus wordt elke keer opnieuw geïnterpreteerd (performantieverlies)
* Oplossing: Just In Time compilatie 🡪 instructies uit tussencode worden opgeslagen (performantiewinst)

**Conclusie**

* Weinig performantieverlies bij goede kwaliteit code
* Productiever en minder kans op fouten dan bij native code
* Betere beveiliging dan bij native code

### Native (unmanaged) code: C, C++, Fortran…

**Werking**

High Level Language 🡪 Compiler 🡪 Native binary code (executable) 🡪 CPU en OS

**Gevolgen**

* Toepassingscode draait rechtstreeks op CPU
* Technische aspecten (memory management, scheduling) zijn verantwoordelijkheid programmeur
* Kans op hogere performantie (High Level Language 🡪 Native binary code)
* Meer kans op fouten

**Conclusie**

* Veel minder controle van OS op fouten of kwaadwillende code
* Slechte code is mogelijke oorzaak van instabiel systeem

### Assembly language

**Werking**

Low level assembler code 🡪 Assembler 🡪 Native binary code 🡪 CPU en OS

**Gevolgen**

* Programmeur bepaalt tot in laatste detail wat de processor doet
* Specifiek per processor(familie)
* Erg technisch en foutgevoelig
* Tijdrovend

**Conclusie**

* Vereist zeer goede en ervaren programmeur voor goede code
* Wanneer maximale performantie nodig is
* Wanneer er zeer beperkte resources zijn (µcontrollers, embedded)

## Het formaat van een processorinstructie

**Native instructies**

Binair opgeslagen in het geheugen, bestaat uit Operation code en Operanden.

Operation code = instructie zelf (wordt eerst gelezen)

Operanden = variabelen (indien nodig gelezen)

*Voorbeeld: mov ah, 0x3f 🡺 kopieer 0x3f naar het register ah*

Niet altijd evenlang, kan ook meer dan 2 bytes zijn.

## Verwerking van een instructie

**Fetch – Decode – Execute**

Eerst wordt de instructie uit het geheugen opgehaald, de **instruction fetch**. Daarna wordt de binaire waarde van de instructie opgeslagen in een het **Instruction Register** (buffergeheugen). Daarna zal de **control unit** van de processor de instructie ontleden in elementaire bewerkingen, de **decodeerfase**. De processor bepaalt dan of er nog operanden nodig zijn (registers, datapaden, ALU). Uiteindelijk zal in de **execution-fase** de control unit ervoor zorgen dat deze elementaire bewerkingen worden uitgevoerd (ophalen operanden, bewerking uitvoeren, stockeren resultaat) door de nodige controlesignalen te genereren. Na de instructie gebeurt deze cyclus opnieuw.

### De instruction fetch

De verbinding van de adres- en databus met het inwendige van de processor gebeurt via de **AdresBusBuffer** en de **DataBusBuffer**. Deze dienen om de externe signalen genoeg vermogen te geven en eventueel de spanningsniveaus aan te passen aan de externe elektronica. Om bij het ophalen van de instructies het juiste adres te kunnen genereren, bevat elke processor een telregister dat het adres bevat van de uit te voeren instructie, dit is de **Program Counter of Instruction Pointer**. Deze wordt bij elke instruction fetch automatisch verhoogd.

Verloop instruction fetch:

* Inhoud PC wordt via de ABB op de adresbus gezet
* CU maakt het externe Read-signaal actief zodat het geheugen actief wordt
* Geheugen plaatst gevraagde instructie op de databus
* CU vergrendelt de instructie-code via de DBB in het IR
* PC wordt verhoogd

Indien de instructie operanden bevat zal de processor deze in de execute-fase inlezen en de PC verder verhogen.

### Von Neuman architectuur ⬄ Harvard architectuur

|  |  |
| --- | --- |
| Von Neuman architectuur | Harvard architectuur |
| I/O system ⬄ CPU ⬄ Code & Data Memory | I/O system ⬄ CPU ⬄ Code Memory  ⬄ Data Memory |
| Bottleneck: geheugenbus | Complexe schakeling (bus, buffers, cache) |
| +/- alle general purpose computers | Somiige microcontrollers (on chip memory) |

## Instuction Set Architecture

### Omschrijving

Belangrijkste aspecten:

* Assemblerinstructies
* Registers
* Geheugenorganisatie
* I/O
* Beveiliging

### De 6502 architectuur

In Apple II, Comdore 64, Acorn Atom/BBC, Atari

#### Programmeermodel

* 8-bit ALU en 16-bit adresbus
* 64 kiB geheugen (ROM, RAM, I/O) 🡪 Memory mapped I/O

#### Registers

* **Accumulator:** Algemeen 8-bit rekenregister (berekeningen en logische bewerkingen)
* **X, Y:** Indexregisters (adresberekeningen)
* **Status Register:** Statusvlaggen Zero – Carry – Overflow – Negative – Decimal – Interrupt
* **Program Counter:** enige 16-bit register (volgende instructieadres)
* **Stack Pointer:** 256 bytes groot, in RAM 0$100 - $01FF

#### Instructies

* Binaire bewerkingen (AND, OR, SHIFT, ROL)
* Rekenkundige bewerkingen (geen vermenigvuldiging en deling)
* Verplaatsen van of naar register
* (Voorwaardelijke) spronginstructies
* Gespecialiseerde instructies voor stack en statusvlaggen

In hardware enkel 8-bit integer bewerkingen.

#### Adresseermethodes

* **Implicit:** De instructie legt impliciet de operand(en) vast. *TAX ; accumulator 🡺 X-reg*
* **Immediate:** De operand wordt rechtstreeks in de instructie gegeven. *LDA #$10 ; 0x10 🡺 A-reg*
* **Absolute:** In de instructie wordt het geheugenadres opgegeven van de operand. *ADC $2100; A-reg = A-reg + content from address 2100*
* **Indexed:** Het opgegeven adres wordt verhoogd met de inhoud van het gebruikte reg. *STA $1000,X; A-reg = content from address 1000 + X-reg*
* **Indirect:** Het te gebruiken geheugenadres wordt meegegeven. Indirecte adressering i.c.m. .Indexed addressing en enkel voor adressen in eerste 256 bytes v/h geheugen. *CMP ($B0,X) ; compare A to byte found at address in ($B0 + X)*

#### Interrupts

* **Non Maskable Interrupt:** Kan niet uitgeschakeld worden. Bij actief: lopende instructie afwerken, PC en SR op stack, springen naar adres op $FFFA. Op $FFFA en $FFFB moet dus het adres van de interrupt handler voor NMI staan
* **IRQ:** Gelijkaardig aan NMI maar kan uitgeschakeld woorden door de I-vlag op 1 te zetten. Vector wordt gelezen op $FFFE en $FFFF
* **BRK:** Software-interrupt. Zelfde effect als actieve IRQ.

### De Intel IA-32 architectuur (x86)

#### Programmeermodel

* Intel 4004 (4-bit), 8008 (8-bit), 8080 (8-bit)
* 1976: Intel 8086 (16-bit, 20-bit adresbus 🡪 1MiB)
* 1985: Intel 80386 (32-bit, 32-bit adresbus 🡪 4GiB)

80386 was compatibel met 8086 (16-bit ‘real mode’), maar had eigen 32-bit ‘protected mode’. Prototype van de moderne processor met beveiligingsmechanismes in hardware. Later onder druk van AMD: 64-bit mode.

Typische kenmerken:

* Bij het inschakelen start de processor op in ‘real mode’. Na initialisatie wordt overgeschakeld naar ‘protected mode’, met volledige capaciteiten.
* 32-bit ALU die ook 8-bit en 16-bit bewerkingen kan uitvoeren. De adresbus en externe databus zijn ook 32-bit (4GiB).
* Randapparatuur met Isolated I/O (64KiB en aparte instructies)
* Geheugenorganisatie in afzonderlijke segmenten met specifieke eigenschappen. Elk proces krijgt minstens een codesegment, datasegment en stacksegment. Deze zitten overal en kunnen overlappen. Er is ook virtueel geheugen voorzien (als theoretisch geheugen > fysiek geheugen).

#### Registers

Algemene rekenregisters: EAX 32-bit, AX 16-bit, AH en AL 8-bit

* **AX:** Accumulator (E/H/L)
* **BX:** Base (E/H/L)
* **CX:** Count (E/H/L)
* **DX:** Data (E/H/L)

Registers voor adresberekeningen: geen bewerkingen, 8-bit

* **BP:** Base Pointer
* **SI:** Source-Index
* **DI:** Destination-Index
* **SP:** Stack-Pointer

Segmentregisters: Informatie over geheugen

* **CS:** Code Segment
* **SS:** Stack Segment
* **DS:** Data Segment
* **EFGS:** Extra Segment

Adres van volgende uit te voeren instructie

* **IP:** Instruction Pointer

Statusvlaggen: Carry, Parity, Zero, Sign, Interrupt enable, Overflow…

* **FLAGS**

#### Instructies

Bewerkingen zowel rechtstreeks in het geheugen als in de rekenregisters.

* Verplaatsingen
* Berekeningen
* Binaire logische bewerkingen
* Voorwaardelijke spronginstructies
* Sinds Intel 486 ook floating point
* Later ook SSE-instructies

#### Adresseermethodes

* Register
* Immediate
* Direct
* Register Indirect
* Register Relative
* Base + index
* Base Relative + Index
* Scaled Indexed

Meer mogelijkheden dan 8-bit. Aantal adresseermethodes zijn speciaal voor arrays en velden in objecten.

#### Interrupts en exceptions

256 Verschillende exceptionhandlers in de Interrupt Vector Table voor zowel externe als interne interrupts.

#### Beveiliging

4 privilegeniveaus (in praktijk 0 en 3)

0: Operating System Kernel 🡪 complete controle

1 en 2: Operating System Services 🡪 Beperkte toegang

3: Application ‘User’ Mode 🡪 Beperkte toegang

### De ARM-architectuur

Acorn Risc Machines, kwam net tegelijkertijd met IBM-PC x86 dus andere afzetmarkt 🡪 embedded control. Zeer geschikt voor taken met relatief hoge rekenkracht en laag energieverbruik. Deze bevat weinig transistoren dus goedkoop en klein. Deze architectuur werd niet geproduceerd als processor door de ontwerpers maar de architectuur werd verkocht met licenties, zo kon een fabrikant de architectuur gebruiken op zijn eigen hardware. Nu zit de ARM in 90% van de mobile devices.

#### Programmeermodel

* 32-bit processor, nadien ook 64-bit
* Op basis van RISC-principes. Load/store architectuur (gegevens laden, bewerking in intern CPU- register, gegevens wegschrijven)
* Minimaal 16 registers van 32/64 bit
* Bijna alle instructies kunnen conditioneel uitgevoerd worden 🡪 compact, efficiënt en goed voor pipeline
* Alle instructies zijn 32-bit lang
* Beperkt aantal instructies, maar worden in 1 klokcyclus uitgevoerd
* Uitgebreide en zeer krachtige adresseermethodes
* Beveiliging: ‘user mode’, ‘supervisor mode’, ‘system mode’
* Bij goed geschreven code 🡪 Zeer efficiënt

# Basisarchitectuur van een processor

## Interne architectuur van een processor

Bij 8-bit processoren 🡺 gelijkaardige basisarchitectuur (nadruk op foutloos verwerken van instructies)

Later 🡺 grote ontwikkelingen:

* Comfort van de programmeur verhogen door het krachtiger maken van de instructies en adresseermethodes. Minder instructies nodig 🡺 productiever en minder fouten
* Performantie van processor verhogen door verhogen klokfrequentie en efficiëntie van interne architectuur verhogen (minder klokperiodes per instructie)
* Extra voorzieningen voor multitasking (elk proces isoleren)

## Basisschema processor

Belangrijkste elementen:

* Registers
* Rekeneenheid (ALU)
* Controle-unit (CU)
* Interne bussen
* Bus-buffers

Interne controlesignalen sturen de werking van de verschillende componenten.

### Processorregister

Een processorregister is een kleine, snelle geheugenplaats (opgebouwd uit een aantal D-flipflops) waarin één geheugenwoord van 8, 16, 32, 64… bits opgeslagen kan worden. Elke processor heeft enkele gespecialiseerde registers:

* **Program Counter/Instruction Pointer:** Geheugenadres van de volgende instructie (wordt automatisch verhoogd naar het adres van de volgende instructie bij fetch)
* **Instruction Register:** Wanneer de operation code van een instructie bij een fetch ingelezen wordt van een extern geheugen, dan wordt deze hierin opgeslagen (van daar kan de CU de bits lezen bij het decoderen van de instructie)
* **Status Register:** Aantal vlaggen waarin de huidige toestand van de processor weergegeven wordt. Een paar standaardvlaggen:
  + **Zero vlag (Z):** 1 als laatst berekende resultaat 0 was
  + **Carry vlag (C):** Bevat eventueel een carry-bit na optelling/aftrekking
  + **Sign vlag (S, N):** Bevat de hoogste bit van het laatst berekende resultaat. Als er gerekend wordt met 2’s complement dan zal de Sign-bit op 1 staan bij een negatief resultaat.
  + **Overflow vlag (O, V):** Indien er gerekend wordt met 2’s complement en het resultaat kan niet in het register, dan zal de overflow vlag op 1 staan.

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

### Rekeneenheid (ALU)

Schakeling waarin de bewerkingen effectief uitgevoerd worden. Het gaat dan minimaal over optellingen, aftrekkingen van int 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. Soms is er ook Multiply and Accumulate (meerdere bewerkingen in één stap).

### Control Unit (CU)

Deze coördineert de werking van de processor. Hij zorgt voor de nodige controlesignalen (intern en extern) voor het ophalen van een instructie, decodeert de instructie en zal op basis daarvan de nodige controlesignalen genereren voor het uitvoeren.

Naast de externe adres-, data- en controlebus zijn er ook intern bussen nodig. Om technische redenen zijn er buffers nodig tussen de interne en externe bussen:

* Intern lage spanningen (0,8V – 1,5V) ⬄ extern hogere spanningen (1,3V – 1,5V)
* Intern weinig vermogen nodig ⬄ extern veel vermogen nodig

## Controle-Unit met hardware decoder

In allereerste processoren gebeurde decoderen na intstruction-fetch door hardware. De verwerking van een assembler-instructie gebeurt intern in aantal stappen. Deze gebeuren door aan- of uitschakelen van controlesignalen. Bij elke instructie dus een opeenvolging van controlesignalen. In de eerste microprocessoren werd dit gegenereerd door een hardwareschakeling:

De bits van de opcode zijn dus een input voor een digitale schakeling. Elke volgorde van bits heeft eigen uitgang 🡺 de decoder is een soort sequencer voor digitale controle signalen. Het effect van de instructies is ingebakken in het ontwerp van de sequencer 🡺 hardware decoder

Complexere instructies 🡺 ontwerp hardware wordt moeilijker

**Voordeel:**

* Snelle werking want hardware schakeling (enkel beperking door schakeltijd technologie)

**Nadeel:**

* Zeer complexe schakeling
* Moeilijk te ontwerpen of aan te passen
* Vroeger geen systemen om complexe ontwerpen te testen (zoals CAD)

## Microprogrammatie

Bij complexere 8-bit processoren en opkomst van 16-bit processoren 🡺 microprogrammatie (vanuit mainframewereld)

**Interne opbouw processor:**

* Datapath: Deel van de processor waar data opgeslagen, verwerkt of verplaatst wordt (interne bussen, registers, ALU, buffers)
* Control Unit: Stuurt in een aantal stappen de blokken van het datapath (via controlesignalen)

Bij hardware decoder 🡺 voor elk controlesignaal een specifieke toestand aanleggen (H of L)

Dit kan ook via intern geheugenblok met opeenvolgende toestanden. Om instructie uit te voeren 🡺 op juiste plaats in intern geheugen starten 🡺 opeenvolgende geheugenplaatsen naar buiten brengen 🡺 controlesignalen aansturen.

Elke locatie in geheugen wordt dan micro-instructie genoemd. Op deze manier kan een instructie van de programmeur opgesplitst worden in een opeenvolging van een aantal micro-instructies.

In plaats van hardware decoder dus een intern geheugen (Control Store) waarin voor elke macro-instructie een opeenvolging van micro-instructies staat. Bij ophalen macro-instructie 🡺 micro-instructies 🡺 controlelijnen aansturen

**Gevolgen:**

* Gemakkelijker om nieuwe instructies toe te voegen (Control Store uitbreiden)
* Interne hardware kan gewijzigd worden zonder de Instruction Set Architecture aan te moeten passen
* Fouten kunnen gemakkelijk verholpen worden door microcode te herschrijven. Bijvoorbeeld de interne microcode van ROM vervangen door een bug-vrije microcode (veel goedkoper)
* Processorontwerp wordt een kwestie van software 🡺 makkelijker testen en ontwikkelen
* Mogelijk om verschillende processoren te ontwerpen die toch compatibel zijn (microcode van alle processoren moet dezelfde macrocode ondersteunen)
* Bij gelijke klokfrequentie: decoder op basis van microcode trager dan hardware decoder

Tegenwoordig gebruiken alle processoren een combinatie van de twee:

* Eenvoudige instructies 🡺 hardware
* Complexe instructies 🡺 microcode

## Complex Instruction Set Computers (CISC)

Aantal instructies van een processor is sterk opgevoerd, instructies zelf warden complexer en krachtiger 🡺 makkelijker voor programmeur en eenvoudiger ontwerp compiler.

🡺 **Complex Instruction Set Computer**

Niet noodzakelijk efficiënter: groot aantal instructies en adresseermethodes 🡺 grote hoeveelheid microcode (sommige amper gebruikt) en complexe instructies zijn vrij traag door groot aantal micro-instructies.

## Reduces Instruction Set Computers (RISC)

CISC heeft zware beperkingen bij het zoeken naar hoge performantie. Er kwamen steeds meer gespecialiseerde instructies en adresseermethodes 🡺 meer microcode 🡺 complexer ontwerp maar niet altijd sneller.

RISC 🡺 processor zonder microcode met hardware decoder. Om de decoder snel te houden: beperkt aantal elementaire instructies en adresseermethodes (snel en efficiënt).

Redenering: veel kleine instructies is sneller dan enkele complexe.

Het is dus belangrijk om zo efficiënt mogelijk instructies uit te voeren 🡺 gemiddeld aantal instructies per klokcyclus (IPC)

Typische kenmerken RISC:

* Beperkte, eenvoudige instructieset
* Vaste instructielengte (32 bits)
* Veel processorregisters (geen bewerkingen in extern geheugen)
* Doel is IPC van 1

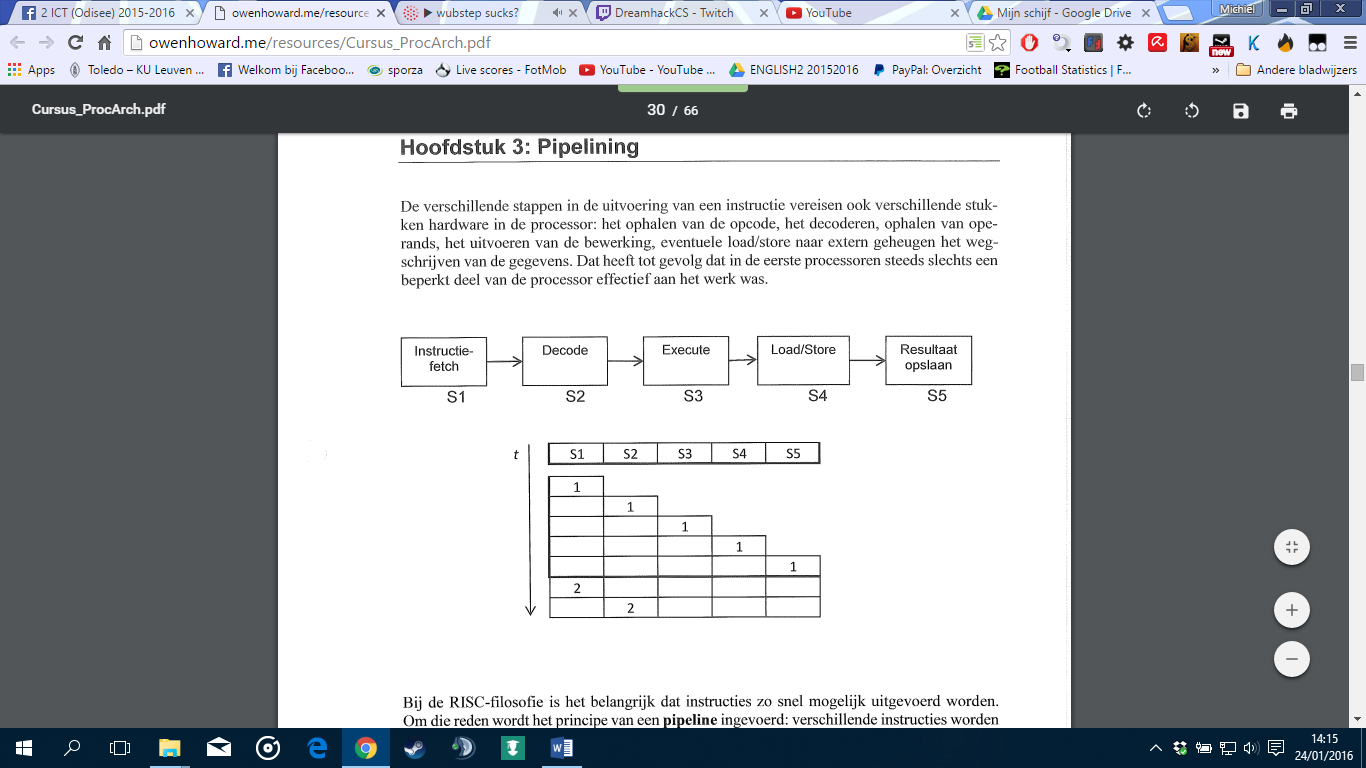
Gevolg:

* Meer code voor hetzelfde werk maar hogere snelheid
* Gemiddelde instructielengte RISC > CISC 🡺 programma op RISC zal groter zijn

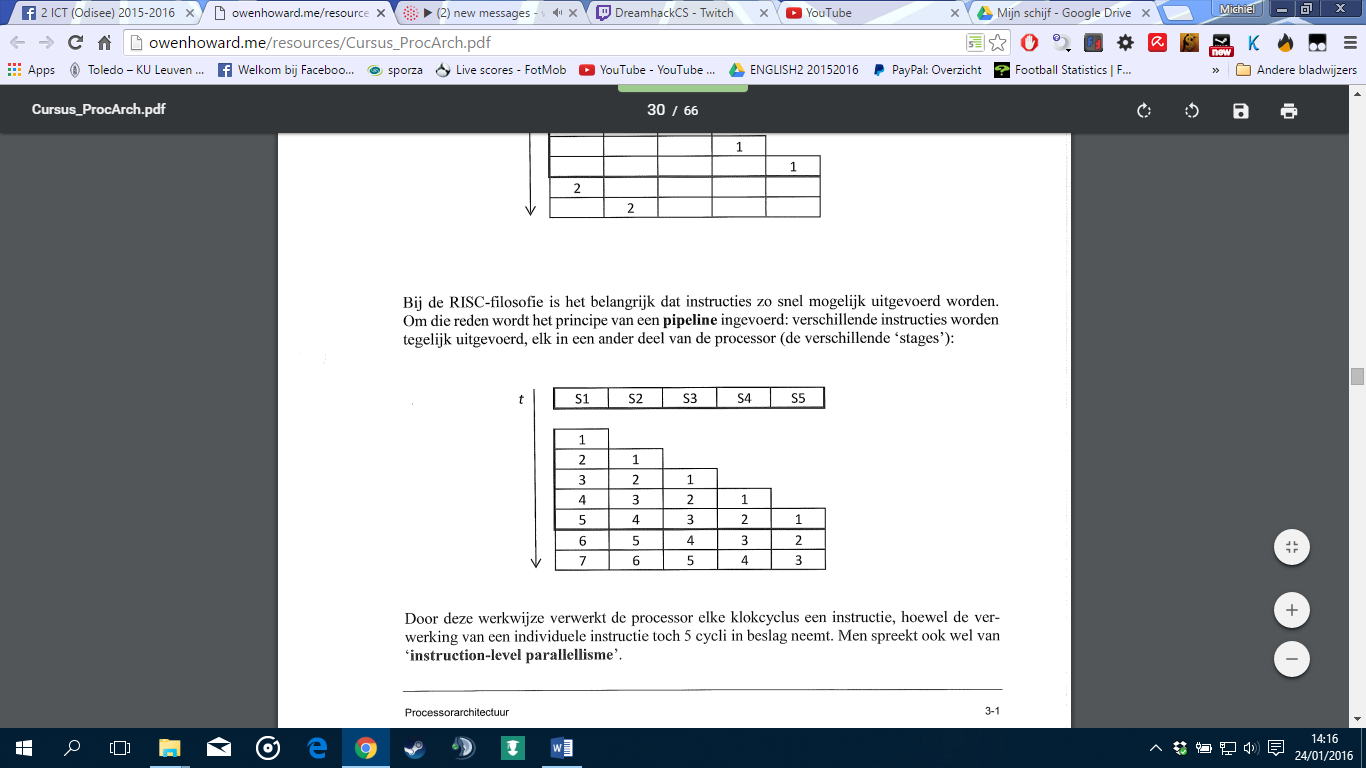
Tegenwoordig aspecten van CISC en RISC in processor.

# Pipelining

Verschillende stappen in uitvoering van een instructie vereisen verschillende stukken hardware in de processor: ophalen opcoden, decoderen, ophalen operands, uitvoeren bewerking, eventuele load/store naar extern geheugen, wegschrijven gegevens 🡺 slechts deel van processor van het werk in eerste processoren



Bij RISC: zo snel mogelijk uitvoeren 🡺 pipeline (verschillende instructies worden tegelijk uitgevoerd maar in ander deel van de processor)



Processor verwerkt elke klokcyclus een instructie: **Instruction-level parallellisme**

In praktijk duurt elke stap niet even lang 🡺 afhankelijk van instructie. Belangrijk is hoe ver de klokfrequentie opgedreven kan worden, dit hangt af van traagste fase. Als bijvoorbeeld S1-4 4ns nodig hebben en S5 8ns dan wordt de klokperiode 8ns en de klokfrequentie 125MHz.

Omgekeerd zal de klokfrequentie opgedreven kunnen worden door een instructie op te splitsen in meerdere stages. Bijvoorbeeld stap van 8ns opsplitsen in 2 stappen van 4ns 🡺 klokfrequentie 250MHz.

Vroeger typisch 5 stages in pipelines, tegenwoordig in meer en kleinere stages.

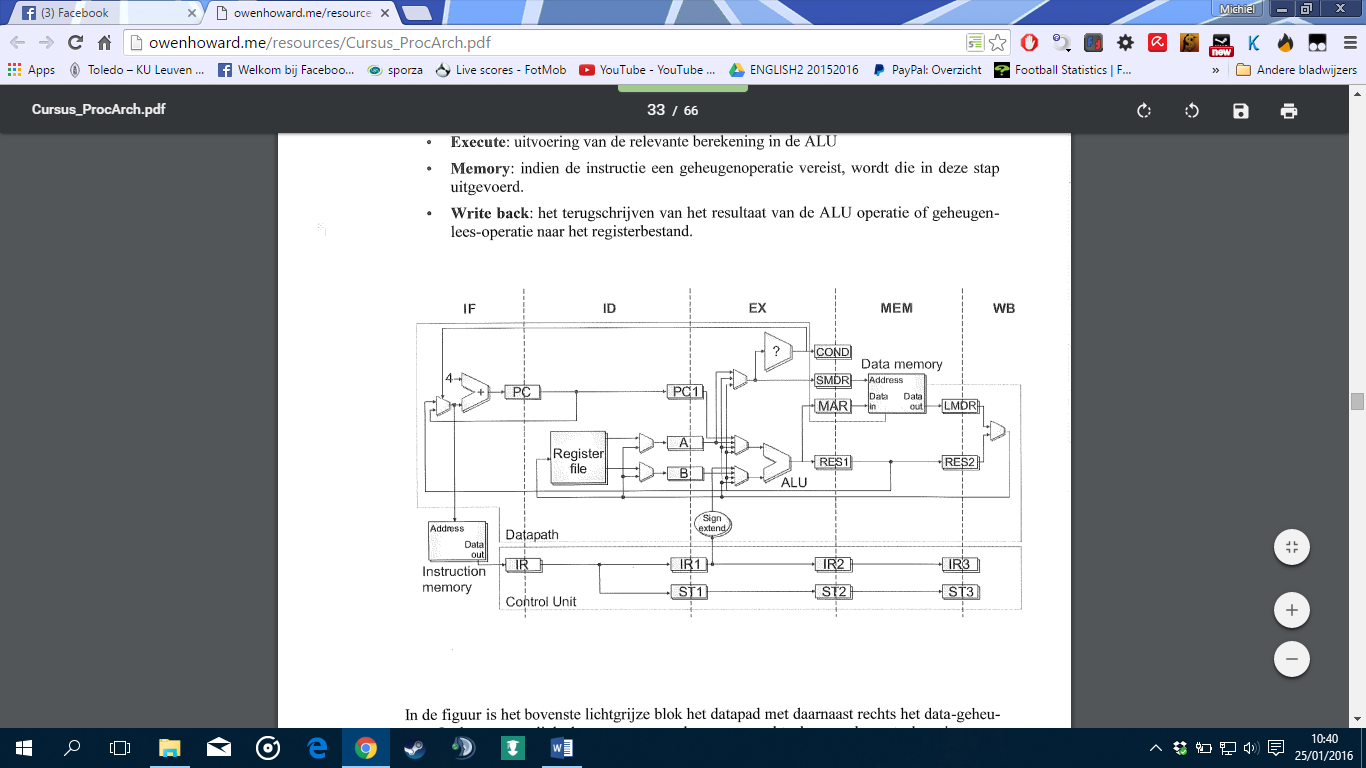
Er is dus een duidelijk verband tussen de lengte van de pipeline en de klokfrequentie. Hoe langer de pipeline, hoe hoger de klokfrequentie. Toch zijn er een aantal beperkingen:

* Gebruikte technologie heeft een schakeltijd voor digitale schakelingen
* Als klokfrequentie stijgt, stijgt het als warmte gedissipeerde vermogen (niet lineair)
* Voortplantingstijd van elektronische signalen in de chip is eindig (lengte van interne verbindingen beperkt klokfrequentie)

## Model van een pipelined RISC architectuur

Theoretisch model met 5 stages

* **Instruction Fetch (IF):** Ophalen instructie uit geheugen en aanpassen PC voor volgende instructie
* **Instruction Decode (ID):** Decoderen van de instructie en uitlezen van de operanden uit het registerbestand voor de ALU bewerking
* **Execute (EX):** Uitvoeren van de relevantie berekening in de ALU
* **Memory (MEM):** Indien de instructie een geheugenoperatie vereist, wordt deze stap uitgevoerd
* **Write Back (WB):** Terugschrijven van het resultaat van de berekening of memory-operatie naar het registerbestand



* **IF:** Instructie uit geheugen en berekent volgende waarde voor PC. In geval van sprong 🡺 volgende PC uit RES1 volgens sprongvoorwaarde COND
* **ID:** IR in ST1 vertalen (beschrijft de aan te sturen controlesignalen). Operanden uit register en klaar voor ALU in A en B
* **EX:** Berekening in ALU. Operanden kunnen ook uit PC of als constante uit instructie komen. Sprongconditie wordt berekend. Resultaat ALU in RES1, resultaat sprongconditie in COND
* **MEM:** Indien geheugenoperatie 🡺 geheugenadres in ALU berekend, te schrijven data uit register via B en SMDR (kan verschillende cycli in beslag nemen)
* **WB:** Terugschrijven van ALU-operatie (in RES2) of MEM-operatie (in LDMR) naar register

## Pipeline ‘hazards’

Meer stages 🡺 hogere klokfrequentie. Dit heeft ook nadelen: alleen efficiënt als instructies zuiver sequentieel worden uitgevoerd.

### Controlehazard

Risico dat instructies (gedeeltelijk) uitgevoerd worden waar dat niet zou mogen. Dit is typisch bij spronginstructies. Bijvoorbeeld spronginstructie naar lijn 7 op lijn 3 🡺 lijn 4 en 5 worden ingelezen terwijl er naar lijn 7 moet gegaan worden maar processor weet dit pas als hij lijn 3 in EX is.

#### Oplossingen voor controlehazards

**Flushen van pipeline**

Overtollige instructies worden genegeerd (NOP), waarna processor verder kan met de juiste instructie 🡺 gedurende enkel klokcycli geen instructies verwerken (STALL).

**Delayed branch**

X aantal instructies na de sprong worden altijd uitgevoerd, programmeur moet dus gepaste instructies vinden net na de sprong. Anders NOP gebruiken.

**Branch prediction**

Proberen voorspellen aan de hand van opgehaalde instructies. In het eenvoudigste geval wordt in een tabel bijgehouden met daarin de voor elke recente spronginstructie, het adres van de volgende instructie. Bij elke spronginstructie wordt dan de volgende instructie van de tabel gebruikt. Er wordt dus verondersteld dat de altijd dezelfde sprong genomen gaat worden (werkt goed bij lussen)

Bijvoorbeeld AMD Athlon (2048 sprongen opslaan) of Intel P4 (4096 sprongen opslaan maar lange pipeline). Nu gesofisticeerder en tot 95% hit rate.

#### Speculative execution

In het geval van een spronginstructie, voorspelt de processor met behulp van branch prediction welke instructies in pipeline komen. Het is niet zeker of dit juist is 🡺 speculative execution.

Indien miss 2 mogelijke technieken:

* **Roll Back:** Alle effecten worden ongedaan gemaakt en instructies worden opnieuw teniet gedaan
* **Kladregisters:** Resultaten van speculative execution worden niet echt gebruikt of opgeslagen, maar bijgehouden in kladregisters

### Datahazard

Ontstaat wanneer een instructie afhankelijk is van het resultaat van een vorige instructie 🡺 Dependency

Bijvoorbeeld: een getal X wordt geset, een getal Y wordt geset, ADD X, Y. De waarde van Y is nog niet juist bij de ADD omdat deze er direct op volgt in de pipeline.

Wanneer 2 opeenvolgende of nabije instructies hetzelfde object lezen of schrijven ontstaat er een afhankelijkheid. 4 soorten:

* **Read After Write (RAW):** Normaal eerst geschreven, dan gelezen. Echte afhankelijkheid. Lezen moet wachten tot na schrijven
* **Write After Read (WAR):** Normaal eerst gelezen, dan geschreven. Anti-afhankelijkheid. Schrijven moet wachten tot na lezen (oplossen door bijvoorbeeld resultaat ergens anders voorlopig wegschrijven)
* **Write After Write (WAW):** 2x geschreven. Anti-afhankelijkheid. 2de operatie mag niet gebeuren voor 1ste (oplossen door bijvoorbeeld resultaat ergens anders voorlopig wegschrijven)
* **Read After Read (RAR):** 2x gelezen. Geen afhankelijkheid, leesoperaties kunnen in verschillende volgorde.

#### Oplossingen voor datahazards

Anti-afhankelijkheden kunnen vermeden worden door een verstandige registerkeuze van de programmeur of compiler (in tijdelijk register opslaan). Ook processor kan in bepaalde omstandigheden zelf beslissen om een ander register te gebruiken 🡺 Register Renaming (sommige processoren hebben ontoegankelijke reserveregisters hiervoor)

In het geval van een echte afhankelijkheid (RAW) 🡺 andere oplossingen

* **Instructies uit mekaar zetten:** Statische methode, afstand hangt af van architectuur (pipeline). Zeer machine-specifiek en niet efficiënt (soms NOP nodig om genoeg afstand te maken)
* **Pipeline blokkeren:** Dynamische methode, bij elke afhankelijkheid uitvoer van volgende instructie uitstellen tot er genoeg afstand is.
* **Forwarding in datapad:** Zodra resultaat bestaat op de processor, beschikbaar voor volgende instructies (niet meer wachten op wegschrijven naar registers). Forwarding kan vanuit EX of vanuit MEM

Voor forwarding zijn extra verbindingen nodig:

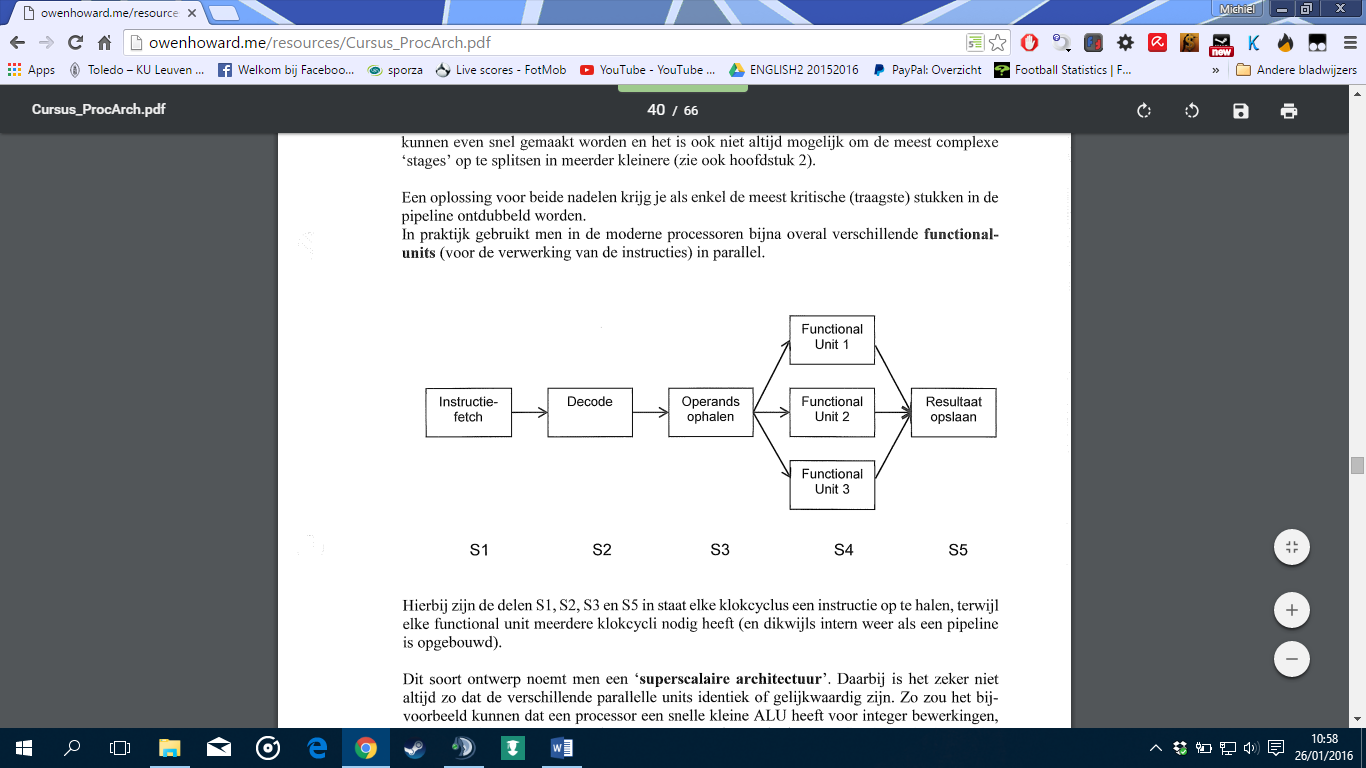
* RES1 – ALU (MEM naar EX)
* RES2/LMDR – ALU (WB naar EX)
* RES2/LMDR – A,B (WB naar ID, is normale werking)

# Superscalaire processoren

## Superscalaire architecturen

Eerste stap naar snellere processoren 🡺 meerdere pipelines (niet echt sneller en complexer)

Ook probleem van niet even lange stages blijft bestaan (soms niet onsplitsbaar omdat te complex)



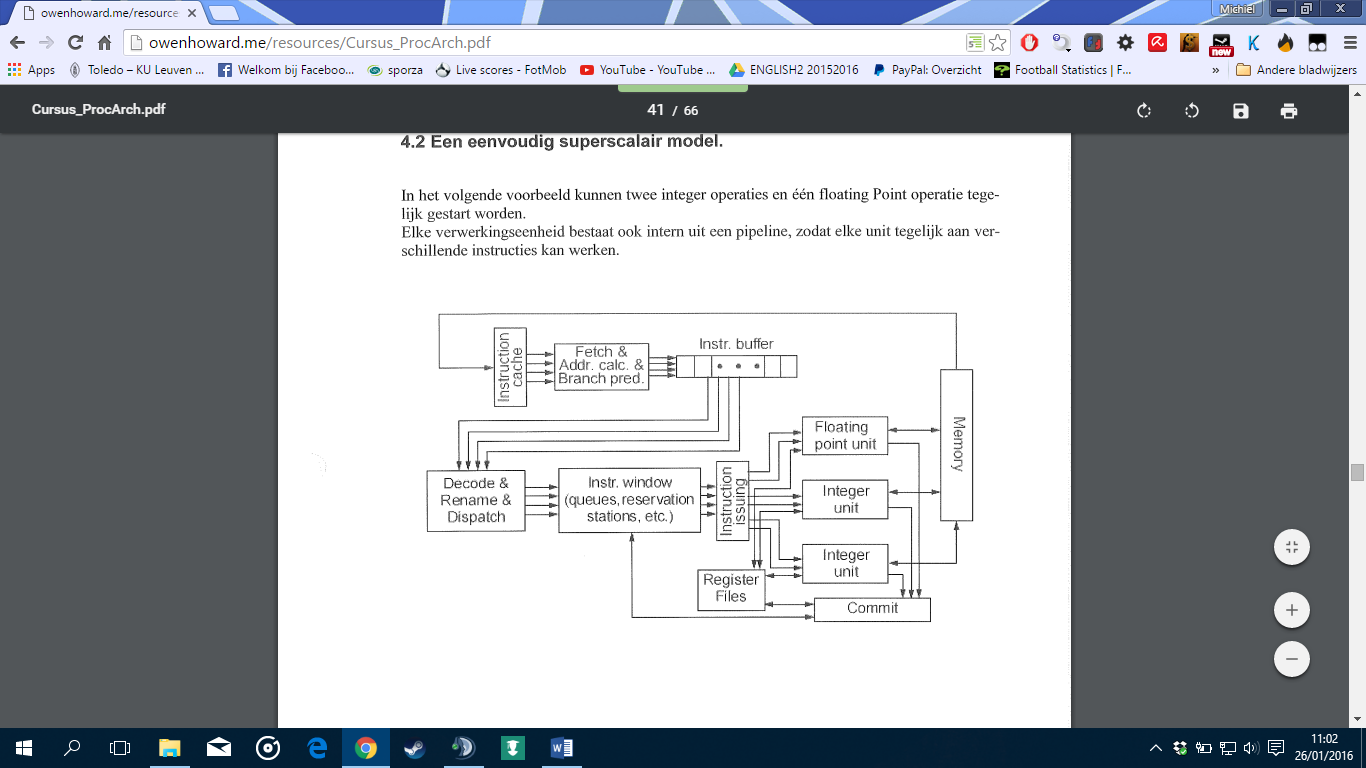
Oplossing: kritische (trage) punten in pipeline ontdubbelen 🡺 functional units gebruiken (bestaan meestal weer zelf uit pipelines) 🡺 Superscalaire architectuur

Functional units zijn niet altijd identiek of gelijkwaardig (bijvoorbeeld snelle kleine ALU voor int en 2 grote trage ALUs voor floating point)

Om processor zo snel mogelijk te maken: meerdere instructies tegelijk ophalen en decoderen (vanuit cache-hiërarchie). Er wordt rekening gehouden met sprongvoorspelling.

## Een eenvoudig superscalair model

2 int ALUs en 1 floating point ALU met interne pipeline



Eerst meerdere instructies fetchen uit instruction cache (Registry Renaming waar nuttig). Wanneer klaar voor verwerking: instructie klaarzetten (Dispatch) in een wachtrij (Reserving station). Bij beschikbaarheid EX unit 🡺 instructieverwerking starten (Issuing)

Neveneffect: extra logica voor READ en WRITE om conflicten te vermijden en koppeling te voorzien 🡺 verschillende registers verzameld in register file

## Out of Order execution

Door superscalaire architectuur 🡺 meerdere instructies tegelijk

Eenvoudigste aanpak: volgorde van verwerking = volgorde van programma (in order issuing) 🡺 kans klein dat instructies effectief tegelijk kunnen uitgevoerd worden. Dit kan enkel als:

* Opeenvolgende instructies in verschillende EX units kunnen uitgevoerd worden
* Geen afhankelijkheden tussen instructies

In praktijk: niet alle EX units tegelijk aan het werk. Om beperkte efficiëntie te verbeteren 🡺 dynamic instructions scheduling (of out of order execution). Hierbij wordt voor de verwerking van instructies enkel rekening gehouden met afhankelijkheden tussen instructies en beschikbare EX units. Niet meer met de volgorde (Uiteindelijke resultaat moet wel gelijk zijn)

Nog een extra stap die ervoor zorgt dat effecten lijken alsof alles in juiste volgorde wordt uitgevoerd 🡺 reorder buffer (of retire unit)

## Window of execution

Belangrijk om afhankelijkheden te checken 🡺 beperkt aantal instructies mogelijk 🡺 om efficiëntie te verhogen: genoeg aantal mogelijke instructies (window of execution)

Dit wordt bepaald door twee factoren:

* **Processor hardware:** Capaciteit om snel instructies op te halen, decoderen en op te slaan
* **Structuur programma:** Spronginstructies

Door branch prediction kan window of execution groter gemaakt worden 🡺 tijdelijk uitvoeren instructies tot sprong berekent is 🡺 indien voorspelling correct 🡺 commit (berekeningen effectief gebruiken)

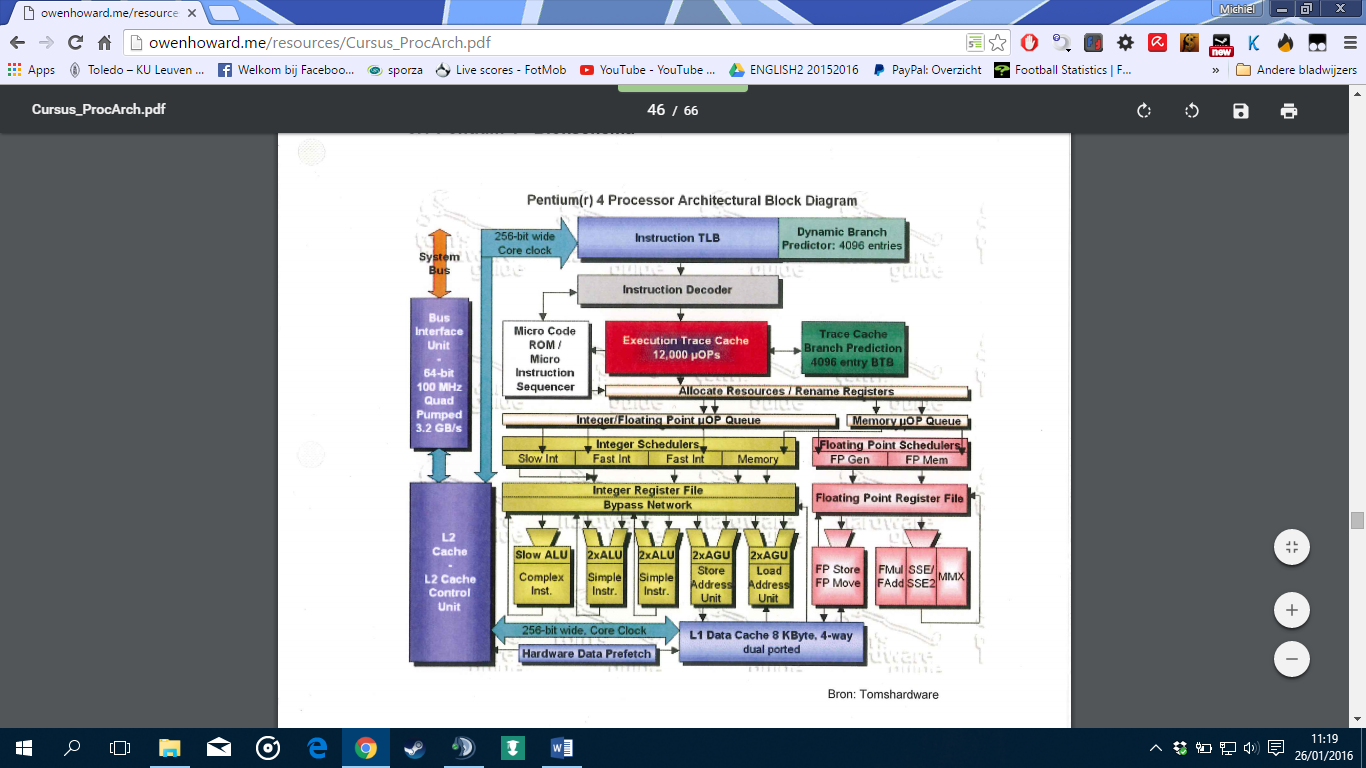
## Afhankelijkheden

Superscalaire processoren 🡺 grotere kans op problemen met afhankelijkheden

* **Anti-afhankelijkheden:** Alleen problemen bij out of order execution 🡺 register renaming
* **Echte afhankelijkheden:** Forwarding en pipeline stilleggen 🡺 out of order execution 🡺 verloren tijd opvullen met andere instructies

# Vergelijking P4 ‘Netburst’ – Athlon

## Pentium P4 – Blokschema



* **Instruction TLB:** Buffer met de instructies zoals ze uit L2 cache geprefetched zijn
* **Branch Target Buffer (BTB):** Tabel met adressen waar naar gesprongen wordt in het programma
* **Micro-operation (µOP):** Gedecodeerde instructies die rechtstreeks door EX units uitgevoerd kunnen worden (decoder bevat elke x86 instructie in een fixed length µOP behalve er wanneer ROM complete µOP wordt gegenereerd)
* **Address Generation Unit (AGU):** Berekent de adressen waar operanden gelezen of opgeslagen moeten worden

## P4 Front Side Bus

Originele P4: 64bit front side bus (100MHz Quad Data Rate) 🡺 8 bytes x 4 (Quad Rate) x 100MHz = 3,2GB/s. Later 133 MHz 🡺 4,2GB/s

Deze FSB is gekoppeld aan een On-chip L2 8 way set associative cache van 2MB (origineel 256kB). L2 cache is gekoppeld aan processorcore (L1 data cache en instruction prefetcher) via 256bit bus met dezelfde clock als de processorcore (L2 cache latency is 7 klokcycli)

Voor data gebruikt P4 een L1 cache van 8kB (4-way SA) met een zeer lage latency (2 klokcycli). Deze cache is dual-ported (1 load en 1 store per klokcyclus)

## Execution Trace Cache

Geen klassieke L1 cache voor de instructies 🡺 eerst instructies decoderen en vertalen naar µOP 🡺 plaatsen in execution trace cache

Bij klassieke L1 cache: per instructie opnieuw decoderen. Hier niet 🡺 grote snelheidswinst bij complexe instructies of lussen

P4 past binnen deze cache 12.000µOPs branch prediction toe 🡺 beperkt stilvallen pipeline door bottleneck in decoders. Per klokcyclus gemiddeld 3µOPs.

## Hyperpipeline

Extreem lange pipeline (20 stappen) 🡺 zo hoog mogelijke klokfrequentie (maar ook grote gevolgen bij probleem want tot 126 instructies in flight)

Door dubbele branch prediction zelden een stall. IP, IF en Trace Cache beslaat eerste 5 stappen.

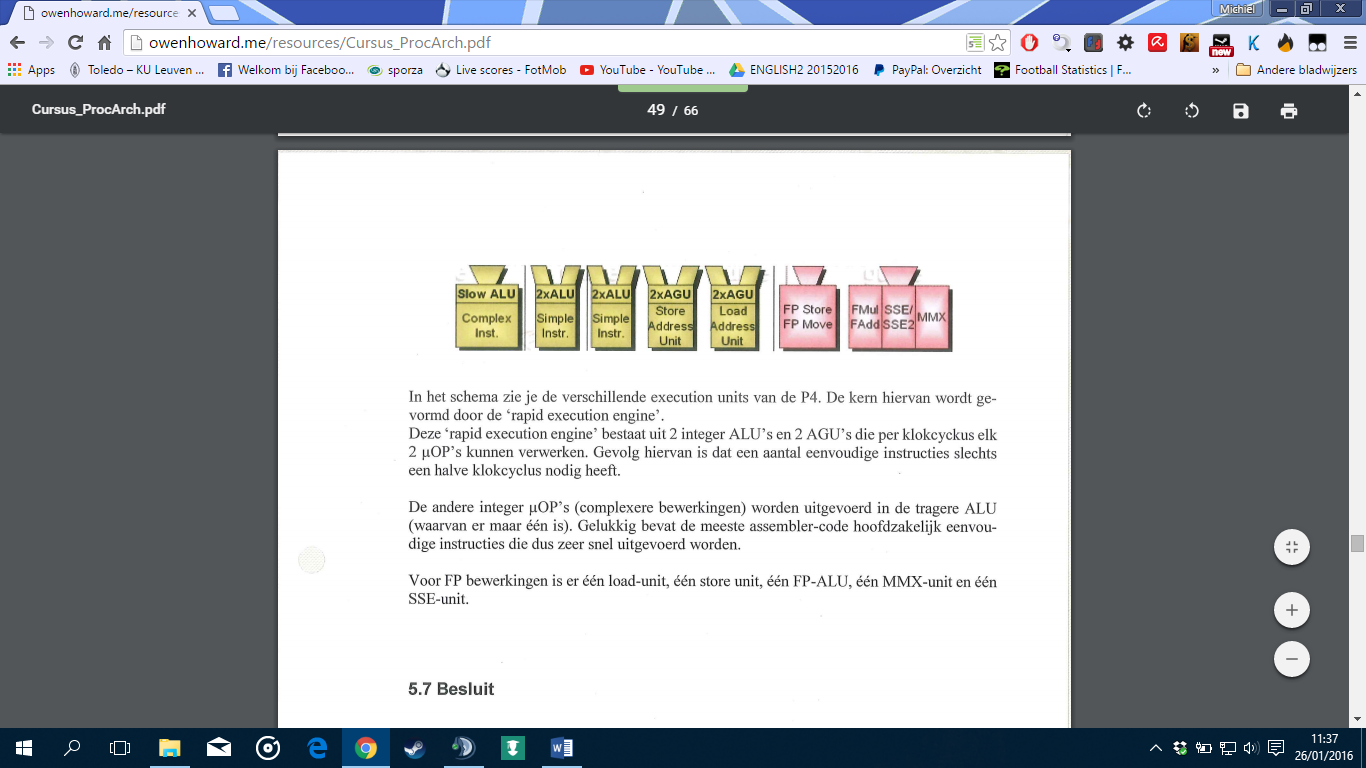
## Out of order execution

Bij het schedulen van de µOPs voor uitvoering gebeuren volgende stappen:

* Toekennen resources (EX units, registers)
* Bepalen afhankelijkheden
* Register renaming (128 register fysiek ipv 8 theoretisch)
* µOPs klaarzetten in µOP queue
* Verwerking van µOP zodra EX unit beschikbaar is

Maximaal 6 µOPs tegelijk per klokcyclus

## Execution Units: ‘Rapid Execution Engine’



Kern is Rapid Execution Engine (2 µOPs per klokcyclus)

* 2 int ALUs
* 2 AGUs

Andere integer µOPs worden uitgevoerd in Slow ALU.

Floating Point:

* 1 load-unit
* 1 store-unit
* 1 FP-ALU
* 1 MMX-unit
* 1 SSE-unit

## Besluit

Voornaamste doelstelling is zo hoog mogelijke kloksnelheid door:

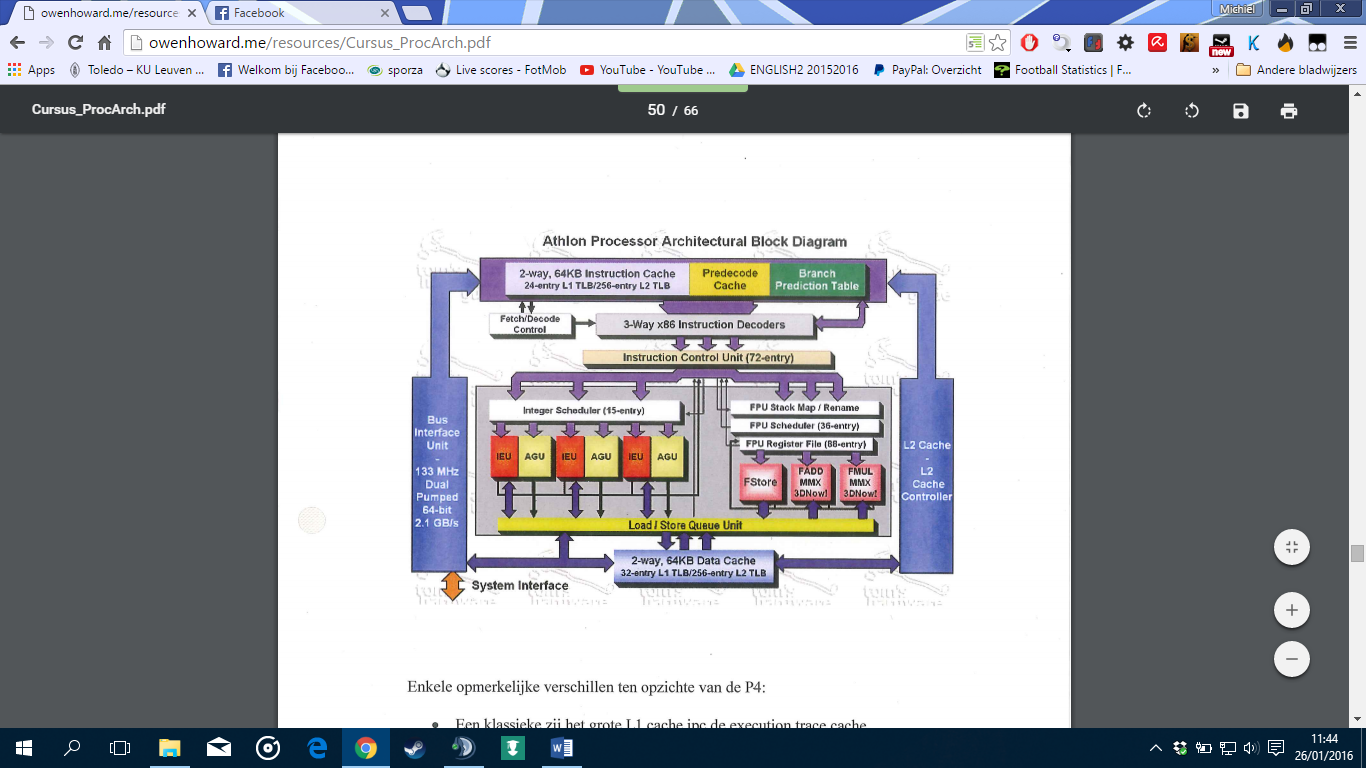
* Lange pipeline
* Snelle EX units
* Nodige logica om problemen te verhelpen

Lagere IPC-waarde dan bij andere architecturen

🡺 Performantie uit hoge kloksnelheid (P4 trager dan P3 bij gelijke kloksnelheid)

Ook vereiste om nadelen van lange pipeline te verhelpen (snel genoeg externe geheugen) 🡺 hoge troughput van FSB en snel genoeg geheugen

## Athlon – Blockschema



Verschillen tegenover P4:

* Klassieke (grote) L1 cache
* 3 parallel decoders (grotere capaciteit voor decoderen)
* Kortere pipeline
* Meer EX Units: 3 volwaardige int units, 3 AGUs, beter bruikbare FP units

🡺 Lagere kloksnelheid maar betere IPC

# ‘Very Large Instruction Word’ processoren

## VLIW

Naarmate een superscalaire architectuur complexer wordt 🡺 complexere problemen bij instructie-scheduling 🡺 schakelingen die instaan voor afhankelijkheden en starten verwerking worden ingewikkeld en duur

Moderne superscalaire processor = dynamische instructie-scheduler

Alternatieve aanpak: deze beslissingen op voorhand laten gebeuren door processor (statisch ipv dynamisch) 🡺 VLIW

VLIW hebben langere instructiecodes (in 1 VLIW instructiewoord zitten een aantal instructies gecombineerd). Taak van compiler is zoveel mogelijk geschikte bewerkingen te vinden die tegelijk kunnen uitgevoerd worden, bij geen geschikt bewerking NOP. Per definitie zijn VLIW instructies fixed-length. Intern zijn er een aantal pipelines met relatief eenvoudige decoder aan een EX unit.

🡺 Gemakkelijker superscalair werken: geen tijd of processorhardware aan analyseren en manipuleren instructiestroom 🡺 kleinere goedkopere processor

**Nadelen:**

* Moeilijkere compilers
* Code Bloating: Wanneer geen combinatie mogelijk is in 1 VLIW komen er NOPs bij 🡺 minder efficiënt
* Verandering aan hardware 🡺 verandering aan compiler

VLIW vooral in embedded en gespecialiseerde DSP-controllers

## De Intel Itanium

Explictly Paralle Instruction Computing (EPIC) is een combinatie van VLIW en RISC/CISC. Ook mogelijkheid om x86-instructies te verwerken (lagere performantie)

* 128-bit instructiewoorden (3 instructies + informatie over EX unit)
* 6 pipelines van elk 8 stappen
* Klok van 1,5GHz
* EX units: 6 int, 6 multimedia, 2 load en store, 3 branch, 4 FP
* 2 instruction bundles per cyclus

Om efficiëntie op te drijven: mogelijkheid om instructies uit te voeren bij bepaalde conditie.

Vooral voor high-end werkstations en servers (twijfelachtige positie door x86)

## De Transmeta-Crusoë

VLIW op een totaal andere manier gebruikt 🡺 x86-processor voor draagbare toepassingen

Doelstellingen:

* x86 compatibel
* Laag verbruik
* Architectuur soepel aan te passen naar betere performantie

Kern is 128-bit VLIW processor (4 IPC) met 2 int units, 1 FP unit, 1 Load/Store unit en 1 branch unit.

De instruction bundles worden molecules genoemd die bestaan uit 4 atoms. Om x86 compatibel te zijn: ingebakken software ‘code morpher’ die zorgt voor vertaling van x86 naar VLIW (zorgt er ook voor dat architectuur gewijzigd kan worden zonder dat externe software er last van heeft). Dit alles wordt aangevuld met speciale power-management technologie die de kloksnelheid en core-spanning aanpast naar de benodigde rekenkracht (Longrun) 🡺 1W verbruik bij 700MHz (⬄ 20W – 40W bij klassieke processor).

# Andere vormen van parallellisme

## Indeling architecturen: SISD, SIMD, MISD, MIMD

### SISD – Single Instruction, Single Data

Klassieke processor waarbij 1 instructie wordt uitgevoerd op 1 unit informatie (ook pipelined en superscalaire processoren behoren hiertoe)

### SIMD – Single Instruction, Multiple Data

Dezelfde bewerking wordt tegelijkertijd uitgevoerd op een aantal afzonderlijk data-objecten (bvb optellen matrices). Bijvoorbeeld MMX op IA-32, Alvitec op PowerPC en vectorprocessoren

### MISD – Multiple Instruction, Single Data

Geen computers van belang op dit principe

### MIMD – Multiple Instruction, Multiple Data

Multithreaded systeem en hyperthreading

## Vectorinstructies (SIMD)

Dezelfde bewerking toepassen op een hele reeks gegevens (bijvoorbeeld filter in PS, renderen 3D) 🡺 hardware die bepaalde bewerkingen op meerdere registers toepast

### MMX

Uitbreiding op Pentium in 1997. 8 64-bit registers extra voor integer bewerkingen (1 x 64 bit – 1 x QWORD, 2 x 32 bit – 2 x DWORD, 4 x 16 bit – 4 x WORD, 8 x 8 bit – 8 x Byte) Deze bewerkingen kunnen saturation arethmic gebruiken (typisch bij DSP). Vooral handig bij audio-bewerking en grafische bewerkingen (2D en 3D matrices)

### 3DNow!

1998 door AMD. FP-bewerkingen op 2 x 32 bit (vooral nuttig voor 3D-graphics en games). Vele uitbreidingen waaronder compatibiliteit voor Intel in 3DNow! Professional.

### Streaming SIMD Expressions

1999 door Intel. Toename FP-berekeningen en migratie van int graphics naar videokaart 🡺 SSE ipv MMX. 8 extra 128 bit registers voor bewerkingen op 4 single precision 32 bit FP. SSE 🡺 SSE3 (integer bewerkingen, extra fixed-point en FP bewerkingen)

### Altivec (Velocity engine, VMX)

Door Apple, IBM en Freescale Semiconductor eeen SIMD uitbreiding voorzien voor de PowerPC (vanaf G4). Vergelijkbaar met SSE, maar grote verschillen in beschikbare instructies en mogelijkheden. Ook in PS3 en XBOX360 worden versies van VMW gebruikt.

## Thread Level Parallelisme (TLP): Hyperthreading

Af en toe nog ongebruikte units 🡺 hyperthreading (verder pararllelisme)

**Klassieke processor is single threaded: 1 thread verwerken op een moment.** Toestand van programma in processorregister (Architectural State) 🡺 meerdere programma’s een voor een na elkaar.

Multithreading op 1 processor 🡺 2 threads om de beurt eventjes op verwerken.

1 fysieke processor doet zich voor als 2 logische processoren voor OS en gebruiker 🡺 2 sets van registers en 2 IPs (weinig hardware uitbreiding nodig)

Niet even efficiënt als dual-processor. Snelheidswinst hangt af van beschikbaarheid threads en aanvulbaarheid threads

## Symmetric MultiProcessing (SMP)

Voor meer performantie 🡺 meerdere processoren in parallel 🡺 SMP (gemeenschappelijk geheugensysteem) 🡺 verschillende threads van 1 proces op verschillende processoren

2 problemen:

* **Bandbreedte van geheugen:** Geheugen trager dan processoren 🡺 processor moet regelmatig wachten (zeker bij meerdere processoren op 1 geheugen)
* **Cache-consistentie:** Om traagheid te compenseren 🡺 cache 🡺 alle processoren moeten op de hoogte gebracht worden als cache van 1 processor naar geheugen wordt geschreven

Alternatief voor SMP: Non-Uniform Memory Architecture (NUMA)

2 andere problemen:

* Verschillende threads van 1 proces kunnen niet op verschillende processoren
* Communicatie tussen processoren wordt ingewikkelder en trager

## Multicore-processoren

Implementeren van meerdere cores in 1 processor, zelfde problemen als bij SMP 🡺 bandbreedte frontside bus verhogen.

Door evolutie van processoren 🡺 afhankelijkheid van kwaliteit software verhoogd

* Optimalisatie voor processor
* Verdeling van rekenwerk tussen verschillende threads die onafhankelijk van mekaar kunnen werken

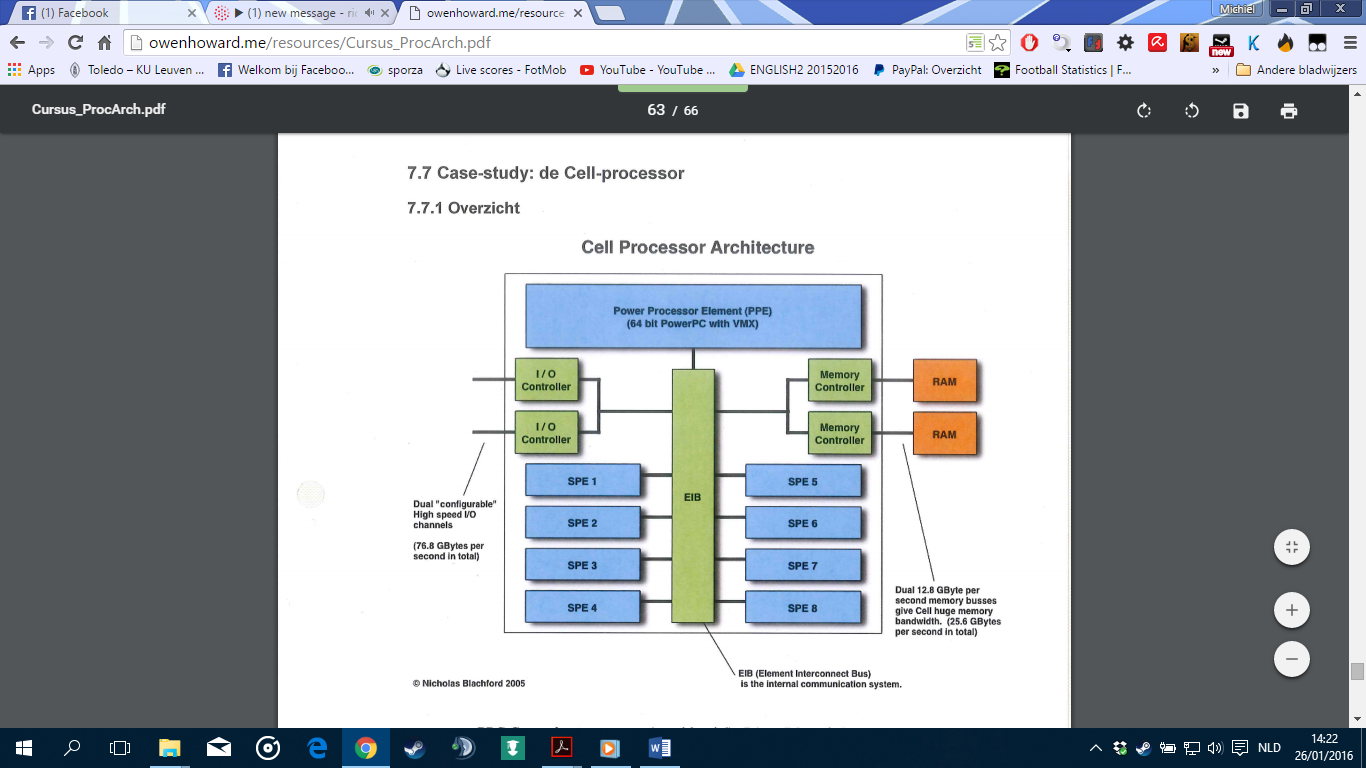
## Asymmetric MultiProcessing (AMP)

Elke processor heeft een andere taak (1 processor voor OS, 1 voor programma’s…)

Hardware-ontwerp kan gebaseerd zijn op de taak voor de processor. Speciaal geval hiervan zijn gespecialiseerde processoren die taken overnemen van gewone CPUs (bijvoorbeeld audioprocessoren, videokaarten…)

## Case Study: De Cell-processor

### Overzicht



* PPC General Purpose Core (L1 en L2 cache)
* X aantal ‘Synergistic Processor Elements’
* Element Interface Bus
* Dual Channel memory controller
* Dual Channel I/O-controller
* Kloksnelheden tot 4GHz

### General Purpose Controller (PowerPC)

Light processor voor OS, algemene taken en coördinatie van SPEs. Compleet nieuw ontwerp (binair compatibel)

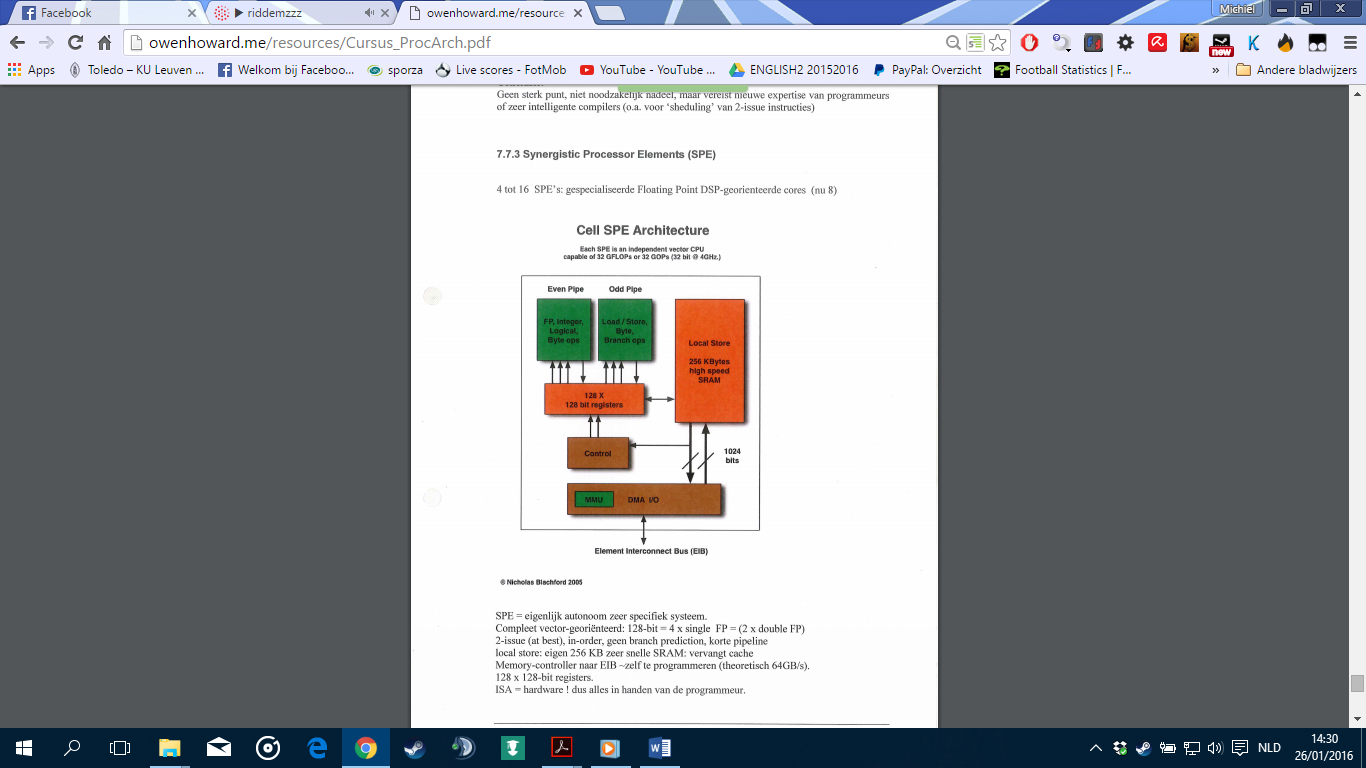
2 Issue, in-orde: nipt superscalair, niet gesofisticeerd, beperkte branch-prediction

Dual-threaded, korte pipeline, toch bedoeld voor hoge clockrate ter compensatie van lage IPC.

🡺Geen sterk punt, niet noodzakelijk nadeel maar vereist nieuwe expertise van programmeurs of intelligente compilers

### Synergistic Processor Elements (SPE)

4 tot 16 SPEs: Gespecialiseerde FP DSP-georiënteerde cores



* Autonoom zeer specifiek systeem, compleet vector-georiënteerd (128 bit = 4 x Single FP)
* 2-issue at best, in-order, geen branch prediction, korte pipeline
* 256kB local storage (SRAM) vervangt cache
* Memory-controller naar EIB (zelf te programmeren, theoretisch 64GB/s)
* 128 x 128 bit registers
* ISA = Hardware

🡺Back to basics, wel SIMD. Efficiëntie afhankelijk van code

### Element Interface Bus (EIB)

* Cross-connect voor intern datatransport tussen I/O, extern geheugen, PPE-cache en SPE-local-stores
* 4 x 16 bit brede ring (halve processorklok)
* 3 transfers gelijktijdig
* Theoretisch 384GB/s
* Externe geheugenbus 25GB/s
* Externe I/O-bus 76GB/s

### Modellen voor gebruik van de Cell

Problemen om de rekenkracht van de Cell te kunnen benutten:

* Probleem vinden voor oplossing
* TLP programmeren op degelijke schaal
* Management van de 8 cores (door GP core)
* Memory bandwidth (25GB/s voor 9 cores)
* Vereiste competentie programmeur

Enkele modellen:

* **Task Level Parallelisme:** Elke SPE voert andere taak uit
* **Collaboration:** Elke SPE voert dezelfde taak uit maar op ander stukje data
* **Cascade / Pipeline:** Stream processing, O SPE 1 🡪 I SPE 2 🡪 O SPE 2 🡪 I SPE 3 …

# GPU

## 3D graphics rendering

GPU’s zijn special onwtikkeld voor het renderen van complexe 3D scenes

### Prehistorie

Geen of zeer beperkte graphics mogelijkheden. CPU berekent juiste waarde voor elk pixel, en stockeert die in framebuffer (=bitmapvoor uiteindelijk beeld op scherm). Videogenerator scant voor elk frame de framebuffer en genereert daaruit het videosignaal.

### 2D accelerator cards

Opkomst GUI 🡺 hardware ondersteuning via gespecialiseerde processor.

Grafische processor krijgt ‘high-level’ commando’s van de CPU en berekent juiste pixelwaarden, verplaatst nodige bits

### 3D graphics

* Veel complexer dan 2D, veel meer rekenwerk
* Objecten: samengesteld uit polygonen, voorgesteld door de coördinaten van de vertexen (knooppunten) in ‘World coördinatespace’ (3-assig assenstelsel)
* Totale beeld wordt opgebouwd door elke polygoon afzonderlijk te verwerken in een aantal stappen:
  + Geometrie: polygonen in de 3D-ruimte(op basis van de vertexen = hoekpunten)
  + Rendering: pixels (fragments) in 2D schermcoördinaten
* Overgang tussen de twee: ‘rasterization’
* LocalCoördinateSystem per object (transformatie van LCS naar WCS nodig 🡪 matrix)

## Geometrie

### Camera

Positie, oriëntatie en openingshoek in WCS. Camera-space (transformatie nodig)

### Zichtbare ruimte (frustrum)

Afgebakend gebied met als grenzen piramide uit de camera, near plane en far plane

### Culling

Weglaten van veelhoeken die toch niet zichtbaar zullen zijn (frustrum culling en backface culling)

### Clipping

Veelhoeken die randen van frustrum snijden omvormen naar veelhoeken in frustrum

## Rendering

### Textuur mapping

Eigenschappen van pixel worden herrekend op basis van textuurmap

Problemen:

* Sterk inzoomen 🡺 pixel worden te groot
* Uitzoomen 🡺 veel informatie die niet zichtbaar is
* Moiré patronen doorinterferentie tussen beide resoluties

### MIP mapping

Verschillende versies van een textuur met verschillende resoluties

### Textuur filtering

Interpolatie pixels

### Textuur bump-mapping

Eigenschappen van pixel worden herrekend op basis van bump-map 🡪 effect op belichting en schaduwen

### Z-buffering, Alfa blending

Z-buffering: combinatie van fragmenten van verschillende veelhoeken op 1 positie (enkel voorste fragment zichtbaar)

Alpha-blending 🡪 transparantie

### Anti-aliasing

Vermijden van een trap-effect vanwege beperkte resolutie scherm

### Motion blur

Kunstmatige onscherpte om indruk van snelheid te geven

### Pipeline

Program (CPU) 🡪 Driver (CPU) 🡪 GPU Front End 🡪Vertex Processing 🡪 Primitive Assembly 🡪 Rasterization & Interpolation 🡪 Fragment Processing 🡪 Raster Operations 🡪 Framebuffer

In praktijk verschillende pipelines in parallel

Sinds DirectX 8 programmeerbare shaders

Verwerking in GPU wordt programmeerbaar om speciale effectne mogelijk te maken 🡪 Vorm van SIMF maar massief parallel

# GPGPU

GPU’s speciaal voor (wetenschappelijke) berekeningen

## Nvidia Geforce 6

* 6 vertex-processoren
* 4 fragment-processoren

### Performance

* Vertex shaders: tot 4 coördinaten tegelijk, matrix-bewerkingen
* Fragment/Pixel shaders: 4 kleurcomponenten tegelijk
* Lussen en sprongen toegelaten. Totaal 128 K instructies (low level) typisch 1IPC

### Beperkingen

* Aantal vertex/shader ligt vast per model
* 2 verschillende programmeermodellen
* Niet alle toepassingen geschikt voor massale SIMD verwerking
* Resultaat komt in Video Ram (traag transport naar CPU-RAM)

## Unified Shader Model (>= DirectX 10)

Slechts 1 soort shader

* Bruikbaar voor zowel Vertex-als Pixel-bewerkingen
* Beter aan te passen aan noodzaak
* Slechts één programmeermodel

## GP GPU processing

Rekenkracht inzetten voor niet –grafische bewerkingen

* Moeilijke programmatie (zeker in assembler)
* Geheugentoegang en –management blijft pijnpunt

🡺 Nu geprogrammeerd in ‘hogere’ talen: CUDA, OPENCL,HLSL (DirectCompute)

Bij tests: GPU 30 tot 40 x sneller dan CPU

# High performance computing

* Nood aan performantie blijft stijden
* Gemeten in Flops (Single & double precision)
* Hoge kloksnelheden & IPC bereikt limieten
* Fysiek: 12nm
* Maar vanaf 5nm quantumeffecten
* Doel 2019: Exascla computing (EFlops system = 1000 PentaFlops)

Evolutie: 1985 0.5MFlops 1,5W 🡪 2015 8,7TFlops (SP) 300W

Klassieke multicore computing: langzame evolutie hard- & software

GPGPU Processing kenmerken:

* Hoge densiteit, goede power efficiëncy
* Vooral efficiënt voor specifieke problemen (streaming)
* Moeilijk programmeerbaar
* Complexe geheugenarchitectuur
* High Latency (geen probleem bij streaming)

Manycore processoren: Relatief groot aantal klassieke cores (Intel Larrabee Project)

Problemen van exascale computing:

* Voeding
* Vermogen datatransport
* Programmeerbaarheid, algoritmen (parallelisern over 100.000+ cores/threads)
* Betrouwbaarheid (1 fout per 10 min) 🡪 Redundante systemen of code immuun voor rekenfouten

# Power management in processoren

## Redenen voor power management

* Thermisch management (bescherming CPU tegen oververhitting, short & long term)
* Verbruik: batterijduur bij mobiele toestellen (notebook, tablet, smartphone, low power embeddeddevices, IoT)
* Processor-efficiëntie (operationele kost servers, HPC)
* ‘Green’ ICT (ecologische aspecten)

## Op niveau van CPU (hardware)

### Dynamic Voltage & Frequency Scaling (DVFS)

Power = C x V2 x F (met C capaciteit van FET’s)

🡺 Als V daalt dan wordt er trager geschakeld en daalt F

Aanvullend circuit opdelen in verschillende zones, elk met eigen spanning en klok:

* CPU
* Cache
* Klokgeneratoren
* Bus-drivers

### Power Gating, Low Power, Idle States

Idle states: CPU/core voert geen instructies uit

Power gating: Delen van de CPU uitschakelen (klok en/of spanning)

Combinaties van de twee:

### C-state in Intel Core i7

* **C0:** Normale staat
* **C1/C1E:** Core stopt, cache snifft naar veranderingen
* **C3:** Core flusht L1 en L2 naar het gesharede L3 cache. Architecturale staat wordt behouden. Core klokken stoppen. Geen sniffing.
* **C6:** Architecturale staat wordt opgeslagen op SRAM, core gaat naar 0V

P-states voor alle cores

### Big little architectuur

* ‘Big’ core: snel, krachtig, veel vermogen (voor zwaar rekenwerk)
* ‘Little’ core: minder performant, zeer efficiënt (voor licht rekenwerk, Big core uitgeschakeld)

Software compatible, typisch in smartphones

## Op niveau van OS (Software)

### Core scheduling

OS bepaalt:

* Op welke core een thread komt
* Welke cores uitgeschakeld worden (C-states)
* Welke P-states gebruikt worden

### Advanced Configuration & Power Interface (ACPI)

Industriestandaard voor configuratie van power states = interface tussen OS en Hardware

**ACPI states:**

* **S0:** Working
* **S0 Low Power:** Modern Standby
* **S1 – S2 – S3:** Sleep
* **S4:** Hibernate
* **S5:** Soft off
* **S6:** Mechanical off