Universiteit van Amsterdam

Samenvatting

Computer Organization and Design

6 september 2018

Studenten:
R. van den Berge
11889330

C.S. Bras
11917318

M. Höfelmann
11844299

R.A.J. Wacanno
11741163

Tutor:
W.R.J. Vermeulen

Practicumgroep:
A1 - ALGOL

Docent:
A. van Inge

Cursus:
Architectuur en Computerorganisatie

Vakcode:
5062ARCO6Y

Inhoudsopgave

1 Computerabstracties en -technologie

R.A.J. Wacanno

1.1 Principes Binnen de Computerarchitectuur

Bij het ontwerpen van computers is men in de loopt de jaren met bepaalde ideeën en principes gekomen welke van groot belang zijn geweest binnen het veld van de computerarchitectuur. Dit zijn veelal principes die bij het bestuderen van de theorie geregeld — het zij direct, het zij indirect — langs zullen komen. Hiervan zijn de volgende acht op dit moment het meest relevant.

De Wet van Moore

Deze 'wet', ooit geformuleerd door Gordon Moore, stelt dat iedere 18–24 maanden het aantal transistoren per IC (Integrated Circuit) verdubbelt.

Het Gebruik van Abstractie

Wanneer men praat over bepaalde onderdelen van een computer zijn niet alle details even relevant. Hierom wordt in deze context vaak abstractie toegepast.

De Versnelling van het Veel Voorkomende

Hoewel dit misschien voor de hand ligt is het ook bij computerontwerp van belang dat, wanneer een proces uit verschillende delen bestaat, het meeste rendement behaald kan worden op het gebied van prestaties wanneer de delen die met grotere frequentie voorkomen efficiënter worden gemaakt.

Parallelle Processen

In sommige gevallen kunnen processen die parallel draaien gebruikt worden om prestaties te verbeteren

Pijplijnen

Bij pijplijnen gaat het om een bijzondere vorm van parallelle processen. Wordt een proces zo geoptimaliseerd dat diens individuele onderdelen samen betere prestaties behalen dan wanneer ieder onderdeel op eigen houtje bezig is.

Voorspellen

Voor sommige processen is het, met het oog op prestaties, voordeliger bepaalde aannames te doen wanneer men weet dat deze dermate accuraat is en mits een mogelijke fout niet te zwaar zal wegen.

De Geheugenhiërarchie

Voor iedere computer is het van belang dat deze beschikt over mogelijkheid het een en ander op te slaan. Verschillende soorten opslag kunnen echter enorm uiteen lopen op het gebied van snelheid, capaciteit, maar ook kosten. Het is daarom dat binnen hardware een bepaalde hiërarchie bestaat met aan de bovenkant het snelste, maar tevens ook kleinste en duurste geheugen en onderaan de opslag met de grootste capaciteit en laagste kosten, maar ook de laagste snelheid.

Betrouwbaarheid d.m.v. Redundantie

Wanneer het van belang is dat een bepaald systeem betrouwbaar is en dus onder vrijwel iedere omstandigheid blijft functioneren kunnen, an bepaalde onderdelen binnen het systeem, meerdere exemplaren worden geïnstalleerd om zo te garanderen dat het systeem bij een mogelijk defect naar behoren blijft werken.

R.A.J. Wacanno

1.2 Onderliggende Software

Om een programma succesvol op hardware te kunnen laten draaien is bepaalde software essentieel. Deze software vormt een tussenlaag tussen het programma en de hardware en maakt het op deze manier mogelijk dat deze twee correct met elkaar kunnen communiceren. Belangrijke voorbeelden van deze software zijn het besturingssysteem, de compiler en de assembler.

Het Besturingssysteem

Het besturingssysteem is onder andere verantwoordelijk voor: het omgaan met in- en uitvoer; het verdelen van opslag en geheugen onder de verschillende processen en mogelijk maken dat meerderen programma's tegelijkertijd en op gecontroleerde op een systeem draaien.

De Compiler

De compiler is onderdeel van een keten die high-level programmatuur omzet om uiteindelijk door de hardware gebruikt te kunnen worden. Om precies te zijn zet de compiler talen (als bijvoorbeeld C en Java) om in zogenaamde assembleertalen.

De Assembler

De assembler maakt de door de compiler gestarte keten af door de reeds genoemde assembleertalen om te zetten in een reeks bits (nullen en enen). Deze bits kunnen vervolgens voor de hardware worden ingelezen en verwerkt.

R.A.J. Wacanno

1.3 Onderliggende Hardware

Om het mogelijk te maken de uitvoer hiervoor genoemde software mogelijk te maken zijn bepaalde hardwarecomponenten binnen een systeem vereist.

In- en Uitvoer

Onder invoer verstaat men hetgeen gegevens in het geheugen schrijft en onder uitvoer hetgeen gegevens uit het geheugen leest.

De Processor

De processor wordt vaak gezien als het hart van een computer; het stelt de computer in staat gegevens van buiten te verwerken om zo nieuwe gegevens aan de gebruiker te presenteren. Deze gegevens haalt de processor uit het geheugen met behulp van **datapaden**. Deze datapaden stellen de processor tevens in staat gegevens
Binnen de processor zit ook een **aansturingseenheid**. Deze stuurt de het datapad, het geheugen en de in- en uitvoer aan op basis van programma-instructies.

Geheugen

Het geheugen onder het snellere deel van de geheugenhiërarchie en staat dichter bij de processor om deze zo van de voor het programma vereiste data te voorzien. Het geheugen kan nog weer worden opgedeeld in het **cache-** en het **werkgeheugen**. Het werkgeheugen bevat de data van het draaiende programma en het cachegeheugen fungeert als een buffer tussen de processor en het werkgeheugen.

Opslag

Opslag beslaat het onderste en dus langzaamste maar ook goedkoopste deel van de geheugenhiërarchie. Bij opslag kan men denken aan harde schijven maar ook het zogenaamde flash geheugen dat in usb-sticks en SSD's zit. Opslag is ideaal voor langetermijnsopslag en voor het bewaren van data die sporadisch door een programma wordt opgevraagd.

Instructieset Architectuur

De instructieset architectuur bepaald welke instructies een stuk assembleertaal kan bevatten wanneer het op een bepaalde processor draait.

R.A.J. Wacanno

1.4 Streven naar Prestaties

Bij het bouwen van computersystemen wordt altijd getracht om zo goed mogelijke prestaties te realiseren. Om dit te kunnen doen moet men in staat zijn om in te zien wat nou eigenlijk goede prestaties zijn; hoe prestaties gemeten kunnen worden en hoe prestaties daadwerkelijk kunnen worden verbeterd.

Prestaties Vaststellen

Bij een computer zijn er twee parameters die als graadmeter voor diens prestaties kunnen dienen met elk zijn eigen toepassing.
Uitvoeringstijd is vooral voor de individuele computergebruiker van belang en kan gedefinieerd worden als de hoeveelheid tijd die het kost on een programma in zijn volledigheid te laten draaien. Twee systemen kunnen op deze manier vergeleken worden als men stelt dat:

$$Prestaties_X = \frac{1}{Uitvoeringstijd_X}$$

Dan volgt het volgende:

$$\begin{aligned} Prestaties_X &> Prestaties_Y \\ \frac{1}{Uitvoeringstijd_X} &> \frac{1}{Uitvoeringstijd_Y} \\ Uitvoeringstijd_Y &> Uitvoeringstijd_X \end{aligned}$$

Bandbreedte is voor datacentrum administratoren van belang en heeft als betekenis de hoeveelheid die per tijdseenheid verwerkt kan worden.

Prestaties Meten

Voor het meten van prestaties kan de uitvoeringstijd van de processor worden gemeten; ofwel de tijd die de processor er over doet een bepaalde taak uit te voeren. Deze kan zowel in seconden als cycli van de processor worden uitgedrukt.

Prestaties van de Processor

De prestaties van een processor kunnen door een tal van factoren worden beïnvloed. Een simpele formule die dit weergeeft is de volgende:

$$Uitvoeringstijd = Processorcycli \cdot Tijd/cyclus$$

Omdat de tijd per cyclus de inverse van de klokfrequentie is kan men ook stellen dat:

$$Uitvoeringstijd = \frac{Processorcycli}{Klokfrequentie}$$

Prestaties m.b.t. Instructies

De prestaties hangen ook nauw samen met de aard van de instructies. Zo is het volgende waar:

$$Klokcycli = Instructies \cdot cycli/instructie$$

De term CPI (Cycles Per Instruction) wordt gedefinieerd als de gemiddelde hoeveelheid cycli die nodig is voor het uitvoeren van een instructie. Met de CPI kan ook de volgende vergelijking worden opgesteld:

$$Uitvoeringstijd = \frac{Instructies \cdot CPI}{Klokfrequentie}$$

Uiteindelijk kan met dit alles een algehele vergelijking worden opgesteld die er als volgt uit ziet:

$$Tijd = Seconden/Programma = \frac{Instructies}{Programma} \cdot \frac{Klokcycli}{Instructie} \cdot \frac{Seconden}{Klokcyclus}$$

De Onontkoombare Stroombarrière

Met het najagen van steeds betere prestaties is men gestuit op onontkoombare eigenschappen van elektriciteit en energie in het algemeen. Wanneer men de klokfrequentie van een processor tracht te verhogen zal deze ook steeds meet stroom verbruiken. Dit heeft tot gevolg dat de processor tijdens gebruik steeds warmer zal worden waardoor beter koeling vereist is. Teven dient, vooral in het geval van bijvoorbeeld draagbare apparaten de toevoer van stroom afdoende te zijn om dergelijke processoren te laten functioneren op hun gewenste snelheid.

De Vermeerdering van Processorkernen

In de meer recente geschiedenis heeft men voor het verbeteren van de prestaties van een systeem zijn heil gezocht in het gebruiken van meerdere processorkernen binnen een processor. Deze ontwikkeling stamt uit het paralleliseren van processen, daar meerdere processorkernen het mogelijk maken meer gegevens tegelijkertijd te verwerken.

R.A.J. Wacanno

1.5 Problematiek op het Gebied van Prestatieverbetering

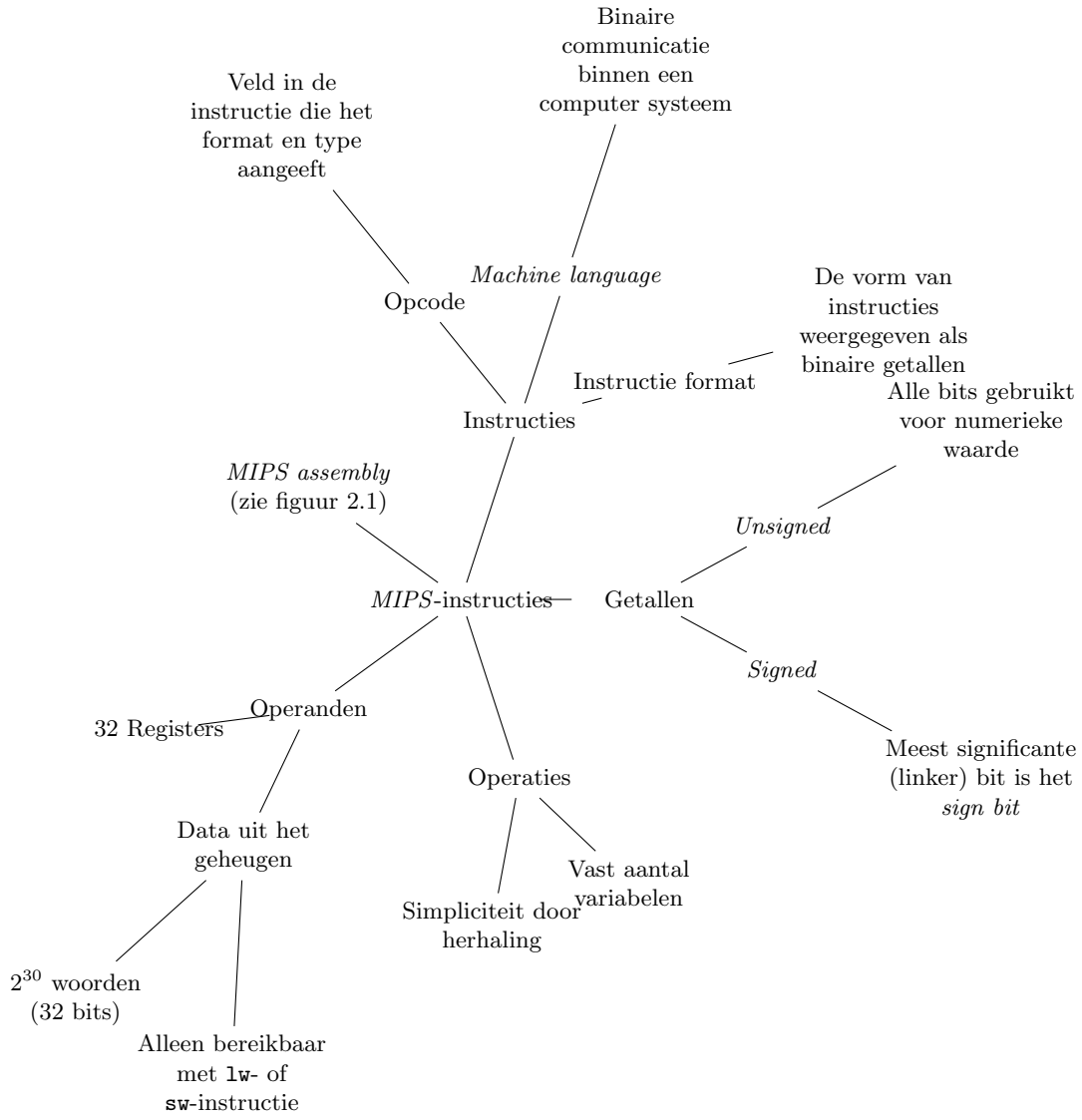
De Wet van Amdahl

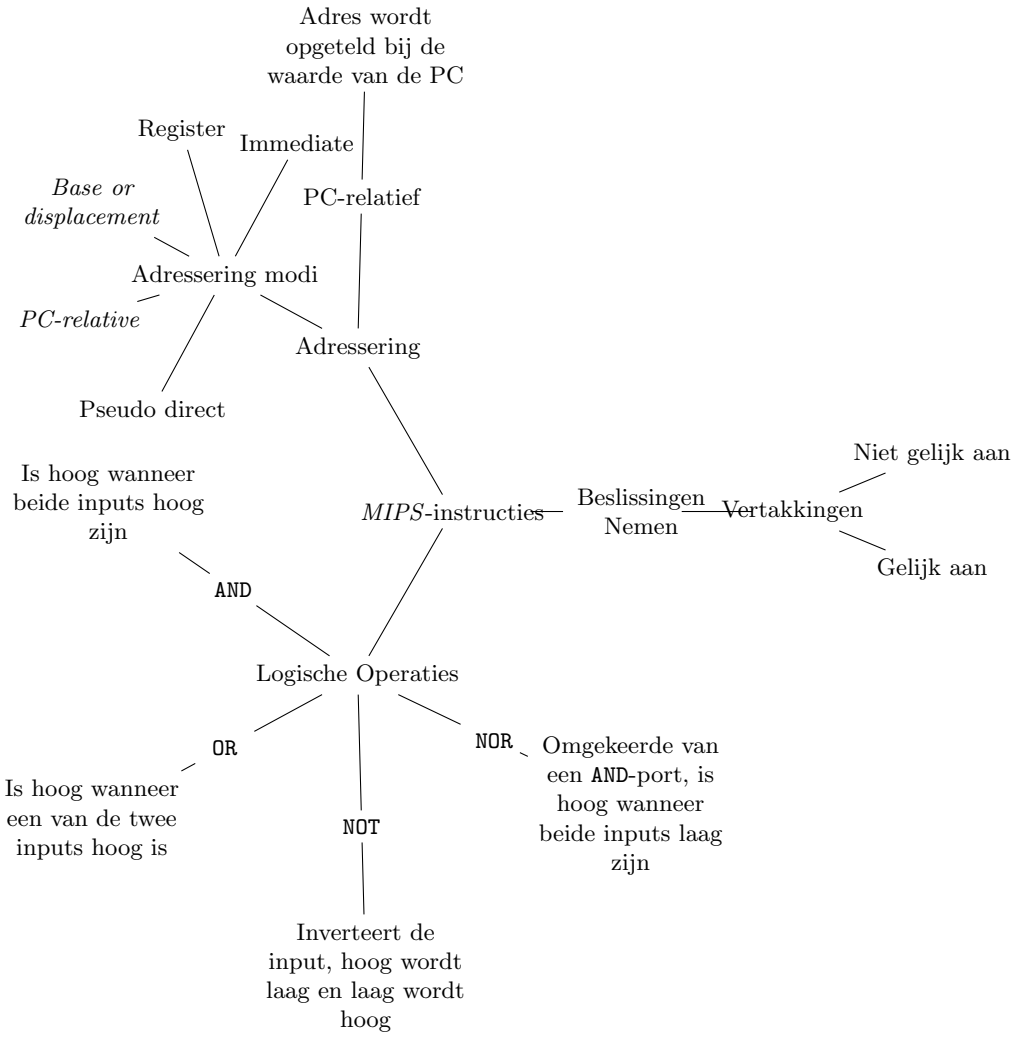
Zie **De Versnelling van het Veel Voorkomende** in combinatie met de onderstaande vergelijking.

$$Uitvoeringstijd_{na} = \frac{Uitvoeringstijd_{beïnvloed}}{Verbeteringshoeveelheid} + Uitvoeringstijd_{onbeïnvloed}$$

2 Instructies: Spreken met een Computer

Hieronder is over twee *mind maps* hoofdstuk 2 verdeeld:




Adres wordt opgeteld bij de waarde van de PC
Register
Immediate
PC-relatief
Base or displacement
Adressering modi
PC-relative
Adressering
Pseudo direct
Niet gelijk aan
Is hoog wanneer beide inputs hoog zijn
MIPS-instructies
Beslissingen Nemen
Vertakkingen
Gelijk aan
AND
Logische Operaties
OR
NOR
Omgekeerde van een AND-port, is hoog wanneer beide inputs laag zijn
Is hoog wanneer een van de twee inputs hoog is
NOT
Inverteert de input, hoog wordt laag en laag wordt hoog

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

Figuur 2.1

3 De Rekenkunde van de Computer

C.S. Bras

3.1 Optellen en Aftrekken

Optellen wordt door de computer op dezelfde manier gedaan als dat wij dat gewend zijn te doen. Van rechts naar links wordt er per bit individueel opgeteld. Als het resultaat hieruit groter is dan 1, wordt het overschot bij de volgende significantie opgeteld.

Aftrekken kan zowel door per bit het aftrekken te verrichten zoals wij dit gewend zijn in het decimale stelsel, evenals door de twee-complement variant van het getal er bij op te tellen wat in essentie dus het optellen van de negatieve tegenpool is.

Overflow bij optellen kan alleen gebeuren wanneer twee getallen dezelfde *sign* hebben, wanneer dit niet het geval is zal het resultaat altijd binnen de twee begin waardes liggen en dus niet *overflowen*. Bij aftrekken is dit precies andersom, overflow kan alleen voorkomen bij getallen met een ander *sign*.

Om te kijken bij een *signed* getal of *overflow* dan toch gebeurd is, wordt er gekeken naar het *sign* bit. Wanneer overflow voorkomt zal deze namelijk niet meer de waarde aannemen van het correcte *sign* maar zal de waarde aannemen van het resultaat.

Bewerking	Waarde A	Waarde B	Uitkomst bij overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A − B	≥ 0	< 0	< 0
A − B	< 0	≥ 0	≥ 0

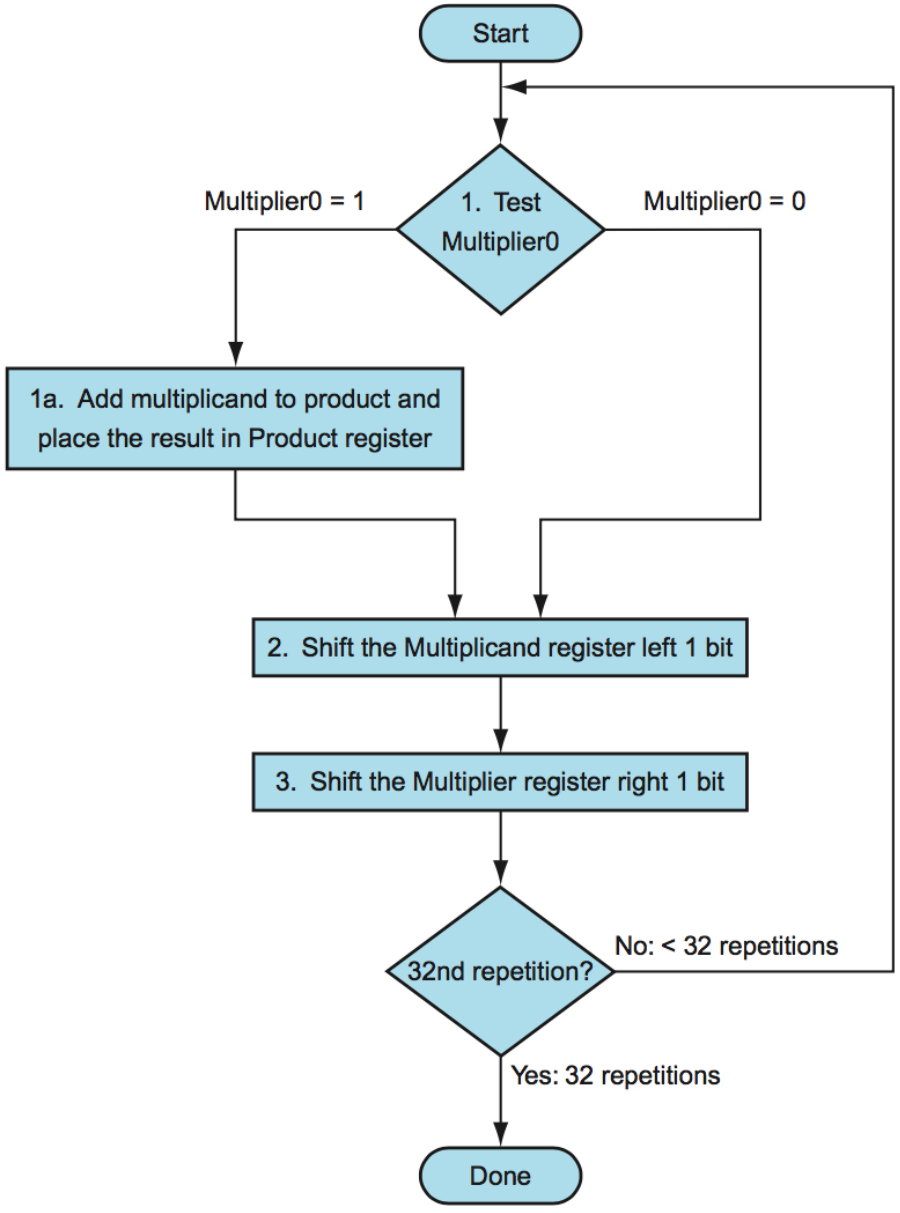
Tabel 1: Tabel met verwachte waardes om overflow te constateren.

Aangezien *unsigned* getallen ook vaak gebruikt worden als geheugen adressen, waarbij *overflow* genegeerd wordt, heeft MIPS verschillende instructies om op te tellen en af te trekken zonder rekening te houden met *overflow* (ADDU, ADDIU, SUBU) evenals instructies die gebruikt worden als er wel rekening moet worden gehouden met *overflow* (ADD, ADDI, SUB).

C.S. Bras

3.2 Vermenigvuldigen

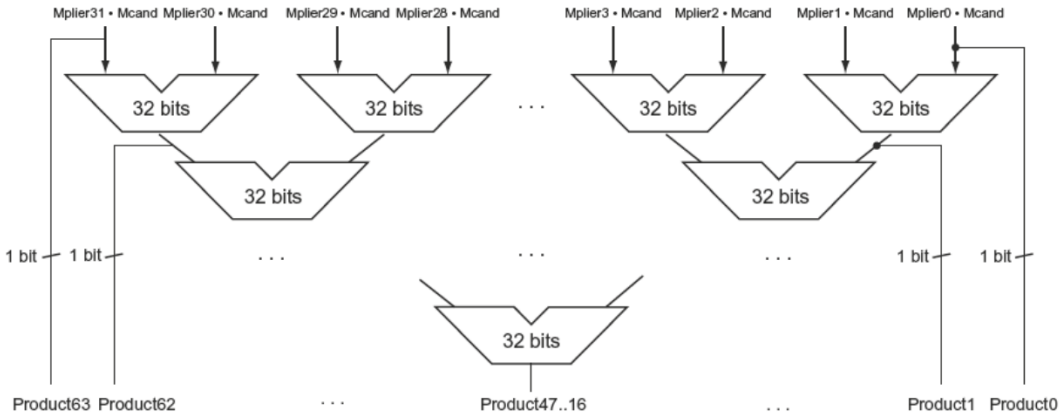
Computers voeren vermenigvuldigingen door een reeks optellingen en shift operaties toe te passen op de waardes waarmee gerekend wordt. Per iteratie wordt er gekeken naar de minst significante bit van de vermenigvuldiger, als deze 1 is wordt het vermenigvuldigtal bij het product opgeteld, met 0 gebeurt er niets. Hierna wordt het vermenigvuldigtal een positie naar links *geshift* terwijl de vermenigvuldiger een positie opschuift naar rechts en wordt alles nogmaals uitgevoerd, net zolang totdat elke bit van de vermenigvuldiger gedaan is.



Figuur 1: Handelingen voor vermenigvuldigen in een schema.

Snelheid wordt behaald door stappen van dit schema parallel uit te voeren, zo worden de registers met de waardes *geshift* terwijl de optelling verricht wordt voor het product.
Een vermenigvuldiging heeft als product een getal ter grootte van de som van het aantal bits van de beginwaardes, dit betekent dus dat een 32-bit getal een product van 64-bits oplevert. In de MIPS architectuur worden hiervan de meest significante bits opgeslagen in het **HI** register en de minst significante bits in het **LO** register. Ondanks dat het product 64 bits lang is, is er maar een 32-bit ALU nodig omdat er alleen maar gerekend wordt met 32-bit getallen.

Verdere snelheidswinst kan behaald worden door meerdere optel units te gebruiken om zo in plaats nog meer handelingen parallel te kunnen verrichten.



Figuur 2: Meerdere optel units als grote vertakking om snelheid van een vermenigvuldiging te verhogen.

3.2.1 Negatief Vermenigvuldigen

Om negatieve getallen te vermenigvuldigen worden eerst de waardes omgezet naar de positieve tegenhangers, daarna worden de benodigde handelingen om te vermenigvuldigen slechts 31 keer gedaan en tijdens de shifts worden de signs behouden, zo blijft de sign-bit buiten de berekening. Aan de hand van de initiële waardes van de sign-bits wordt bepaald welke waarde de sign-bit moet hebben van de uitkomst.

C.S. Bras

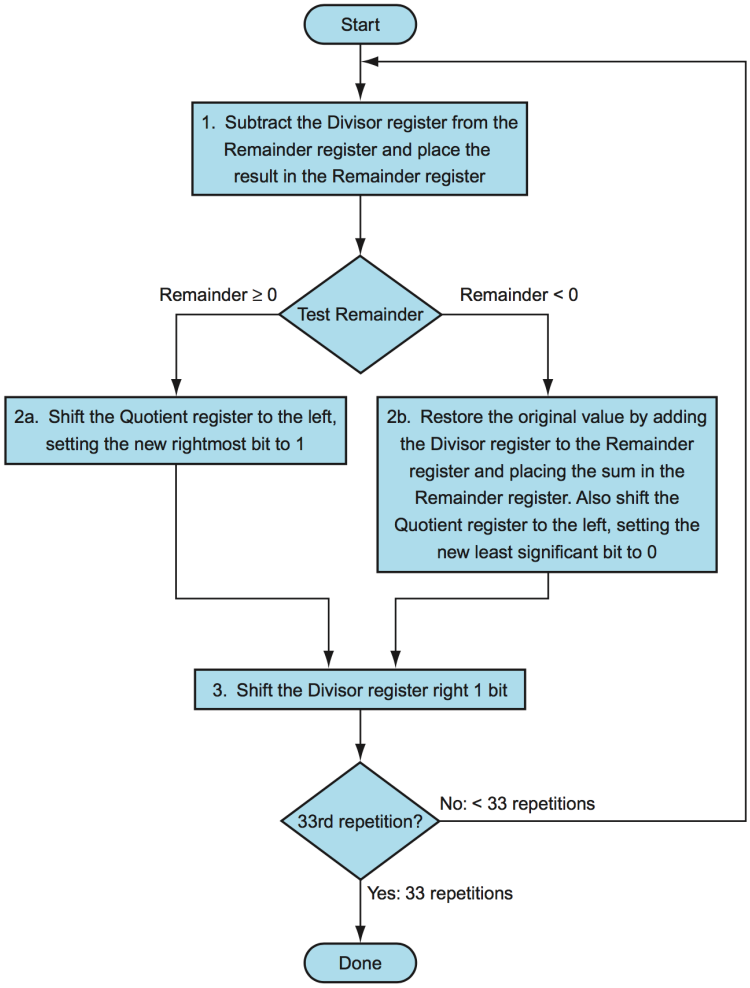
3.3 Delen

Delen bestaat evenals vermenigvuldigen voor een computer uit een langere reeks aan simpelere handelingen. In plaats van optellen, maakt de computer gebruik van aftrekken. Dit wordt gedaan door telkens te kijken naar welke significantie de deler geshift kan worden terwijl deze nog past in het deeltal, dit wordt er dan vanaf getrokken en in het resultaat wordt de bit waarvan de significantie overeen komt met het aantal shifts naar links op 1 gezet. Dit herhaalt zichzelf totdat de deler niet meer past in het getal dat is overgebleven van het deeltal en er dus een rest over blijft. Dit is dus eigenlijk een staartdeling.

$$
\begin{array}{rll}
 & 1001_{ten} & \text{Quotient} \\
\text{Divisor } 1000_{ten} & \overline{)\,1001010_{ten}} & \text{Dividend} \\
 & -1000 & \\
 & \quad 10 & \\
 & \quad 101 & \\
 & \quad 1010 & \\
 & \quad -1000 & \\
 & \quad\quad 10_{ten} & \text{Remainder}
\end{array}
$$

Figuur 3: Een uitwerking van een staartdeling.

De hardware die dit moet uitvoeren, moet over een aantal functies beschikken. De deler begint in de meest significante positie en wordt elke iteratie een positie naar rechts geshift terwijl het quotiënt begint in de minst significante positie en wordt telkens naar links geshift. Tijdens elke iteratie wordt gekeken of de resulterende waarde na het aftrekken kleiner is dan 0 door naar de sign bit te kijken, als dit het geval is betekent dit dat de deler in deze significante positie niet past en dus moet deze er weer bij opgeteld worden om terug te komen bij het getal waarmee deze iteratie begon. Dit staat allemaal beschreven in het figuur hieronder.



Figuur 4: Een schema van het algoritme dat gebruikt wordt om te delen

Net als vermenigvuldigen geeft delen ook een 64-bit resultaat wanneer er gedeeld wordt met

twee 32-bit getallen, echter is het ditmaal verdeeld onder het quotiënt en de rest die overblijft na het delen. Net zoals bij het vermenigvuldigen is er ook geen 64-bit ALU maar een 32-bit variant omdat de shifts tijdens het aftrekken al worden uitgevoerd en er dus geen 64 bits aan ruimte nodig is bij het aftrekken.

3.3.1 Negatief Delen

Negatieve getallen kunnen gedeeld worden door de signs van het deeltal en de deler te onthouden en vervolgens de positieve varianten van deze waardes tegen elkaar te delen zoals hiervoor beschreven is. Dit brengt echter een paar dingen met zich mee waar op gelet moet worden, het meest belangrijk zijnde het juiste sign plaatsen bij het restgetal zodat de formule $Deeltal = Deler \cdot Quotiënt + Rest$ geldig blijft maar er niet voor zorgt dat de absolute waarde van het quotiënt verandert doordat de sign van de deler of het deeltal verandert.

3.3.2 Snelheidsverbeteringen

In tegenstelling tot vermenigvuldigen kunnen er niet een hele hoop optel units achter elkaar gezet worden die meerdere deelstappen tegelijk uitvoeren omdat de sign bit van het resultaat bekend moet zijn om de volgende stap te bepalen. Een manier om toch een hogere snelheid te behalen is door te werken met voorspellende hardware die door middel van een look-up tabel probeert te voorspellen wat de waarde gaat zijn, dit heet de *SRT divison technique*. De snelheidswinst die dit oplevert wordt vooral bepaald door de kwaliteit van de tabel waarin gezocht wordt.

3.3.3 Delen in MIPS

In MIPS kun je de instructies DIV en DIVU gebruiken om respectievelijk een signed en een unsigned integer te delen. Het resultaat wordt net zoals bij een vermenigvuldiging opgeslagen in het HI en LO register en dus zullen deze opgehaald moeten worden met de MFHI en MFLO instructies.

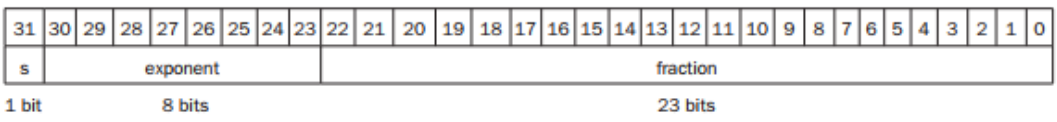
R. van den Berge

3.4 Floating Point

Programmeertalen ondersteunen kommagetallen, welke reële getallen worden genoemd in de wiskunde. Een paar voorbeelden van reële getallen zijn π, e, 0.000000001 $1.0 \cdot 10^{-9}$ en $3,155,760,000$ ofwel $3.15576 \cdot 10^{-9}$. Het laatste getal is groter dan dat een integer kan bevatten. Het getal is geschreven in de **wetenschappelijke notatie**, wat inhoudt dat er één getal voor de komma staat en er een maal 10x wordt toegepast afhankelijk van het aantal cijfers voor of achter de komma van het originele getal. Een getal dat voor de komma geen 0 heeft staan, heet een **normalized** (gestandaardiseerd) getal. Dus $1.0 \cdot 10^{x}$ is een gestandaardiseerd getal, maar $10.0 \cdot 10^{x}$ en $0.1 \cdot 10^{x}$ bijvoorbeeld niet.

Net als decimale getallen kun je ook binaire getallen in de wetenschappelijke notatie schrijven. Om een binair getal in zijn gestandaardiseerde vorm te houden, moet je als basisgetal een getal nemen om bits te kunnen verschuiven, 2 dus. Je neemt dus 2 als basisgetal in plaats van 10. Je krijgt dus $1.xxxxxxx \cdot 2^{yyyy}$. De gestandaardiseerde vorm en wetenschappelijke notatie gecombineerd bieden drie voordelen: Het maakt het data uitwisselen makkelijker met floating point getallen, het versimpelt de wiskundige algoritmes die van tevoren al weten dat ze te maken krijgen met deze notatie en het verhoogd de nauwkeurigheid waarmee de getallen kunnen worden opgeslagen.

Omdat je maar een beperkt aantal bits tot je beschikking hebt moet je een compromis maken tussen de grootte van de exponent en de getallen achter de komma. Ook moet worden aangegeven of het getal positief of negatief is.



Figuur 5: De floating point representatie.

Van links naar rechts gelezen wordt de eerste bit dus gebruikt om aan te geven of het getal positief of negatief is. De volgende acht om de exponent aan te geven, de exponent wordt berekend door de getallen van de acht bits bij elkaar op te tellen -127 (min $0 - 127$ en max $256 - 127$ dus). Dan worden de laatste 23 bits gebruikt om het getal achter de komma uit te rekenen. De vorm is dus $(-1)^s \cdot fraction \cdot 2^{exponent}$.

Op het moment dat er een te groot positief exponent wordt ingevoerd om met deze methode weer te geven vindt er een **overflow** plaats. Dit betekent dus dat het exponent te groot is om het getal met het aantal bits weer te geven. Een **underflow** is hetzelfde als een overflow maar dan bij te grote negatieve exponenten. Om de kans op overflow en underflow te verkleinen heb je ook een **double precision**. Het eerder besproken model heet de **single precision**. Bij de double precision wordt de exponent verhoogd naar 11 bits en de fraction wordt eerst verkleind naar (20) bits en daar worden vervolgens 32 bits aan toegevoegd, een totaal van 52 bits dus voor de fraction en 64 bits voor het gehele getal.

Deze notatie gaat volgens de IEEE (754) floating-point standard uitgevonden in 1980 en sindsdien zit het in elke computer. De term significante cijfers (significand) wordt gebruikt om de fraction van 23 of 52 bits plus de s van 1 bit aan te duiden, dus $23 + 1$ en $52 + 1$ significante cijfers zijn er beschikbaar bij floating point representation. IEEE (754) heeft ook een notatie voor een getal als 0/0, NaN (not a number). De bedenkers van IEEE 754 wilden dat het lezen van het getal voor bijvoorbeeld simpele vergelijkingen zoals kleiner of groter dan 0 de bit voor een positief of negatief getal aan het begin hebben. Hierdoor hoeft bij zo'n vergelijking maar 1 bit gelezen te worden.

Floating-Point Representation

EXAMPLE

Show the IEEE 754 binary representation of the number -0.75_{ten} in single and double precision.

ANSWER

The number -0.75_{ten} is also

$$-3/4_{ten} \text{ or } -3/2^2_{ten}$$

It is also represented by the binary fraction

$$-11_{two}/2^2_{ten} \text{ or } -0.11_{two}$$

In scientific notation, the value is

$$-0.11_{two} \times 2^0$$

and in normalized scientific notation, it is

$$-1.1_{two} \times 2^{-1}$$

The general representation for a single precision number is

$$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent}-127)}$$

Subtracting the bias 127 from the exponent of $-1.1_{two} \times 2^{-1}$ yields

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000_{two}) \times 2^{(126-127)}$$

The single precision binary representation of -0.75_{ten} is then

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

The double precision representation is

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(1022-1023)}$$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

32 bits

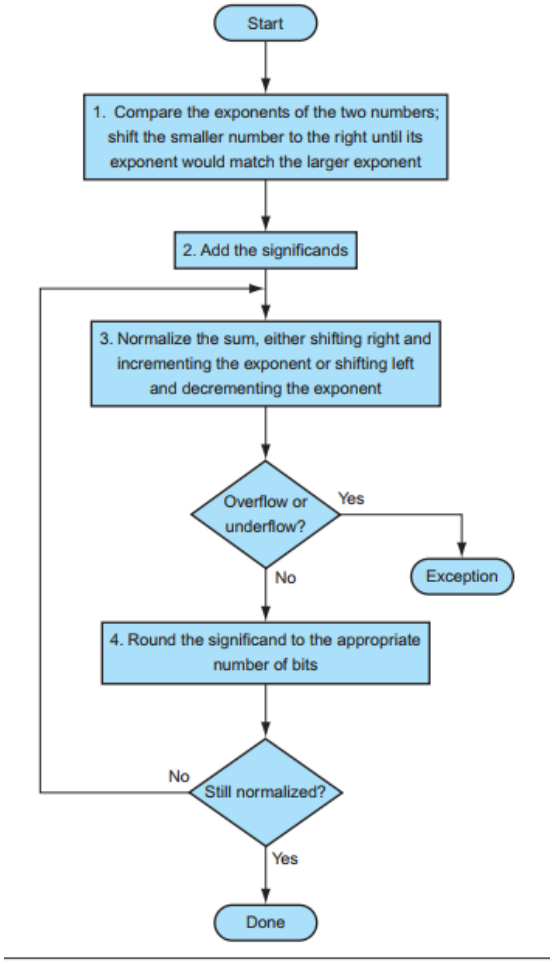
Figuur 6: Hier een voorbeeld van het omzetten van een decimaal getal naar de IEEE 754 representatie.

3.4.1 Floating-Point additie

Om de problemen te beschrijven met het optellen van getallen in wetenschappelijke notatie met floating-point additie is hier eerst een voorbeeld met normale getallen. Het gaat hier om $9.999 \cdot 10^1 + 1.610 \cdot 10^{-1}$. Het uiteindelijke getal heeft maar 4 significante cijfers en 2 getallen voor het exponent.

1. De exponenten moeten gelijk woeden. 10^{-1} wordt 10^1. Het getal wordt dus $0.01610 \cdot 10^1$.
2. Nu ga je de getallen bij elkaar optellen; $0.016 + 9.999 = 10.015$. De som is dus $10.015 \cdot 10^1$.

3. Dit getal moet opnieuw in wetenschappelijke notatie gezet worden. Het wordt dus $1.0015 \cdot 10^2$. Wanneer er zo'n geval is van verschuiven en exponenten vergroten moet er altijd worden gecheckt op een underflow of overflow.
4. Omdat er maar 4 significante cijfers mogen zijn, moet het worden afgerond. $1.0015 \cdot 10^2$ wordt $1.002 \cdot 10^2$.



Figuur 7: Het schema van floating-point additie.

3.4.2 Floating-Point vermenigvuldiging

Bij getallen vermenigvuldigen met getallen in de wetenschappelijke notatie gaat het iets anders dan bij optellen. Hier het voorbeeld bij $1.110 \cdot 1010 \cdot 9.200 \cdot 10^{-5}$. Hier ook weer 4 cijfers voor de significand en 2 voor de exponent.

Stap 1: Je kan bij vermenigvuldigen de exponenten bij elkaar optellen. In dit geval $10 + -5 = 5$. Als je dit doet met een binair exponent dan moet je dus twee binaire exponenten bij elkaar optellen. In dit geval dus $10 + 127 + -5 + 127 = 259$. Dit is te groot om in 8 bits op te slaan. Vandaar dat hier van tevoren al -127 wordt gedaan. De uiteindelijke exponent is dus 259–127 = 132.

Stap 2: Dan ga je de significante cijfers vermenigvuldigen, $9.200 \cdot 1.110 = 10.212$. Som is dus $10.212 \cdot 10^5$.

Stap 3: Dan gaan we het cijfer standaardiseren: $1.0212 \cdot 10^6$.

Stap 4: we mogen maar 4 significante cijfers hebben dus het wordt $1.021 \cdot 10^6$.

Stap 5: Afhankelijk of de bit die aangeeft of de getallen positief of negatief zijn van de getallen gelijk aan elkaar is, is het product positief. Als deze verschilt is het getal negatief.

R. van den Berge

3.5 Fallacies and Pitfalls

Een fallacy in de computerwereld is: Net zoals je een getal n bits naar links opschuift en het getal met 2n wordt vermenigvuldigd, wordt het getal met n bits opschuiving naar rechts gedeeld door 2n. Dit is inderdaad het geval bij unsigned integers, maar niet bij signed integers. $-5/4$ moet met een integer -1 opleveren. -5 in binair is 1111 1111 1111 1111 1111 1111 1111 1011. Als je dit 2 naar rechts opschuift dan krijg je 0011 1111 1111 1111 1111 1111 1111 1110. Dit zou dus -1 moeten opleveren, maar in werkelijkheid is dit $1,073,741,822$. Floating point getallen zijn slechts benaderingen van echte getallen en omdat computers slechts een gelimiteerde nauwkeurigheid hebben werkt de bovenstaande formule niet.

Een andere fallacy is dat parallelle operaties die voor integers werken ook voor floating-point datatypen werken. Hierom moeten programmeurs die parallelle code schrijven met floating-point instructies bewijzen dat deze operaties correcte en betrouwbare resultate weergeven.

Een pitfall is dat ADDIU instructies (add immediate unsigned) zijn 16-bit getal uitbreidt. Een ADDIU instructie wordt gebruikt om waarden toe te voegen aan signed integers als overflow niet uitmaakt. MIPS heeft geen subtract immediate instructie dus de MIPS architecten hebben dus besloten om het immediate veld te sign-extenden.

In 1994 maakte Intel fouten met hun Pentium. Ze gebruikten een algoritme dat bij delingen de volgende 2 bits raadden aan de hand van de belangrijkste bits van de divisor en de dividend. Deze gerade waarden komen uit een tabel die -2, -1, 0, $+1$ of $+2$ bevat. Door fouten in dit algoritme waren de eerste 11 bits altijd juist, maar er zaten regelmatig fouten in bits 12 tot en met 52. Dit kostte intel uiteindelijk 500 miljoen dollar.

4 De Processor

Max Höfelmann

4.1 Introductie

Dit hoofdstuk bevat een uitleg van de principes en technieken die gebruikt worden in het implementeren van een processor. Het grootste gedeelte zal gaan over een meer realistische implementatie van pijplijnen in MIPS, met daarbij nog een deel over de meer complexe benodigdheden voor het implementeren van moeilijkere instructie sets zoals x86.

We gaan onderzoek over een subset van de MIPS kern instructies:

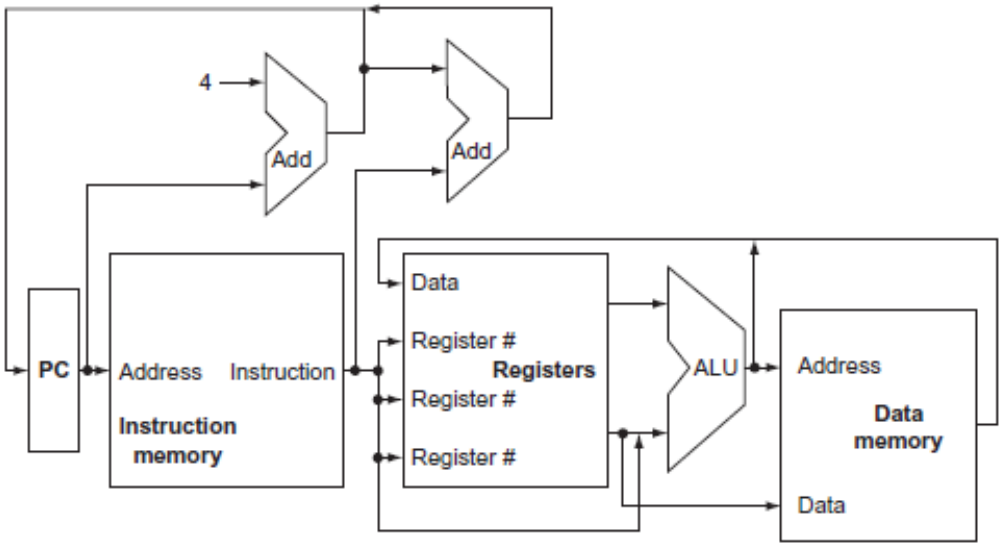
- De geheugen instructies **load word** (lw) en **store word** (sw).
- De arithmetisch-logische instructies add, sub, AND, OR en slt.
- De instructies **branch equal** (beq) en **jump** (j), deze zullen als laatst worden toegevoegd.

Tijdens het onderzoeken van deze instructies zullen we zien hoe verschillende aanpakken de **clock rate** en **CPI** zullen aanpassen. We zullen zien dat eenvoud houd van regelmaat.

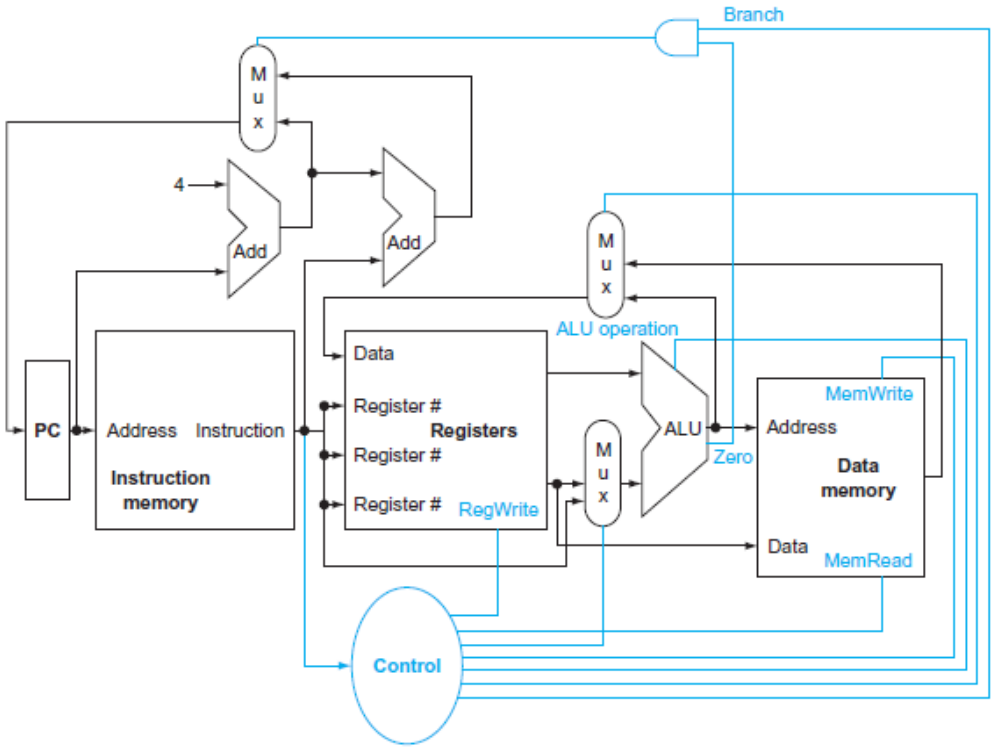
Voor elke instructie zijn de eerste twee stappen indentiek:

1. Stuur de programma teller naar het geheugen dat de code bevat en haal de instructie van dat geheugen.
2. Lees een of twee registers afhankelijk van welke registers er worden vermeld in de instructie. Voor de lw instructie hoeven we maar een register te lezen maar de meeste andere intructies vragen voor twee registers.

Na deze twee stappen ligt het aan de instructie klasse welke actie er nodig is om de instructie te volbrengen. Gelukkig zijn voor elk van de drie instructie klasse (**memory-reference, arithmetic-logical en branches**) de acties grootendeels hetzelfde. In figuur 4.1.1 zien we een hoger-niveau van de MIPS implementatie, gefocussed op de verschillende functionele gedeeltes en hun connecties.



Op verschillende plekken is te zien dat een enkele input kan komen van meerdere sources zoals bij de PC er meerdere ADD's leiden naar de input lijn van de PC. In de praktijk kan dit niet zo simpel en zullen er logic elementen geinstalleerd moeten worden om de juiste source te kiezen. Zo'n logic element wordt vaak een **multiplexor** genoemd of nog beter een **data selector**. In figuur 4.1.2 zijn de 3 benodigde multiplexoren en de daarbijhorende datapaden toegevoegd aan figuur 4.1.1. De toegevoegde control unit, die instructies als input heeft, wordt gebruikt om te bepalen hoe de control lijnen voor de eerste twee multiplexoren lopen. De derde multiplexor, die bepaald of de PC + 4 of het bestemming adres wordt opgeslagen in PC, werkt op basis van de ALU output

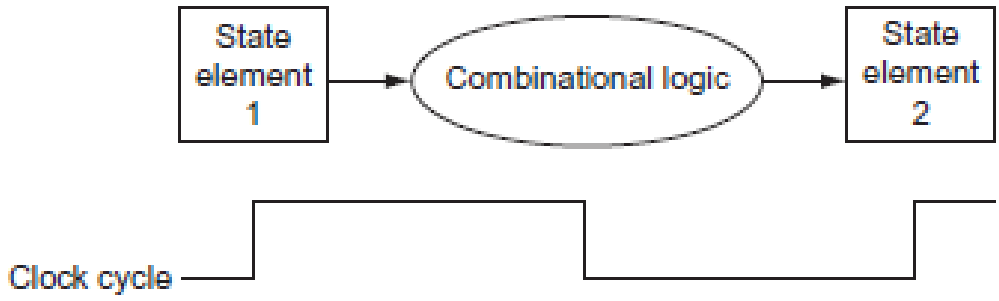


Max Höfelmann

4.2 Logica-ontwerp Verbeteringen

De datapad elementen in MIPS bestaan uit twee verschillende types: elementen die werken via data waardes en elementen die een staat bevatten. Alle elementen die op waardes werken heten **combinational**, dit betekent dat de output alleen afhankelijk is van de inputs. Elementen die staat bevatten noemen we **state elements**, simpelweg een register of een stuk geheugen. Een state element heeft tenminste twee inputs en een output. De inputs zijn de data waardes die moeten worden geschreven in het element en de klok, de klok bepaald wanneer de waardes worden geschreven. De output bevat de waarde die eerder de input in kwam.

Clocking methodoly zorgt ervoor dat hardware voorspelbaar is door te bepalen wanneer data valide en stabiel is relatief tot de klok. **Edge-triggered clocking methodology** zorgt juist dat alle data verandering worden opgeslagen wanneer er een klok ëdge"plaats vind, een snelle transitie van laag naar hoog of andersom. Aangezien de state elements alleen een waarde kunnen opslaan moet een **combinational logic** zowel een state element als input en output hebben. De input is geschreven in een vroegere clock cycle terwijl de output gebruikt kan worden een een volgende clock cycle. Zie figuur 4.2.1



Een **control signal** is een signaal dat gebruikt wordt voor multiplexor selectie of het sturen van de operatie van een functionele eenheid, in tegenstelling tot een **data signal** wat informatie bevat dat wordt geopereerd door een functionele eenheid. Om een state element te veranderen met een control signal moet er sprake zijn van een **asserted** signaal. Dit houd in dat er een hooge waarde

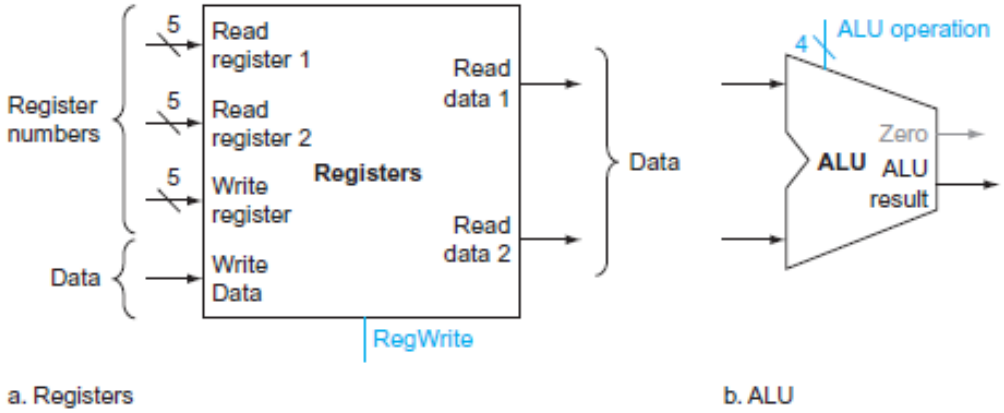
moet zijn of een waarde die waar is. **Deasserted** is het tegenovergestelde en betekent dus laag of false.

Max Höfelmann

4.3 Een Datapad Bouwen

We beginnen aan het bouwen van een datapad door alle groote objecten die nodig zijn om elke klasse uit te voeren in MIPS instructies. **Datapath elements** zijn elementen die gebruikt worden om op geopereerd te worden of data op op te slaan binnen een processor. In MIPS implementaties bevatten de datapad elementen de instructies, data herinneringen, het register bestand, de ALU en de adders. De PC, program counter, is ook van belang. Dit is een register dat het adres van de huidige instructie. Door de een adder kunnen we de PC verhogen zodat we naar het adres van de volgende instructie gaan. Er zijn ook **R-format** instructies. Deze lezen allemaal uit twee registers en voeren allemaal een ALU operatie uit op de inhoud van de registers. Onder R-format instructies vallen add, sub, AND, OR en slt.

De 32 general-purpose registers van de processor zijn opgeslagen in een **register file**. Dit is een collectie van registers waarin elke kan worden geschreven of gelezen, hiervoor is de ALU nodig.



Zoals hierboven in figuur 4.3.1 gezien kan worden zijn er ook meerdere outputs mogelijk.

De MIPS instructies lw en sw maken een 16-bit signed veld in de instructie. Om dit 32-bit te maken is er een **sign extender** nodig. Deze sign extender vergroot simpelweg de grootte van het gesignde veld door de hoge order bits dupliceren. de bew instructie heeft 3 operanten. De eerste twee zijn twee verschillende registers die worden vergeleken voor gelijkheid. De derde operand is de offset die wordt uitgevoerd als de twee registers gelijk zijn. Er zijn 2 details in de definitie van **branch** instructies die we in gedachten moeten houden:

- De instructie set architectuur zegt: de basis van van de berekening van een branch adres, het adres is van de instructie na de branch.
- De architectuur zet tevens de offset field zo dat hij 2 bits naar links shift, een word offset. Dit zorgt voor een offset field van factor 4.

Als de beq true is is er sprake van een **branch taken**, het branch doel adres wordt de nieuwe CP. Als de beq false is is de branch not taken en vervangt de geincrementeerde PC de oude PC zoals elke andere instructie dit ook doet.

4.3.1 Een Enkel Datapad Maken

Het simpelste datapad zal alle instructies willen uitvoeren in een clock cycle. Dit betekent dat geen enkel element twee keer gebruikt mag worden. Elk element dat meerdere keren gebruikt moet worden zal dus gekopieerd moeten worden. Hierdoor zal er een geheugen voor instructies en data moeten zijn. Alhoewel vele elementen gekopieerd zullen moeten worden kunnen er ook vele gedeeld worden verschillende instructie kanalen. We kunnen een multiplexor and control signaal gebruiken om te selecteren welke input gebruikt moet worden bij welke instructie.

Max Höfelmann

4.4 Een Simpel Implementatieschema

We gaan hier kijken naar de meest simpel mogelijke implementatie van de MIPS subset. Deze implementatie wordt gemaakt met het datapad van de vorige sectie en we voegen daar een paar simpele controle functies aan toe zoals lw, sw, beq, add, sub, AND, OR, set on less than en tenslotte **jump to instruction (j)**. De 4-bit ALU control input kunnen we genereren door een kleine controle eenheid met als input het functie veld van de intructie en een 2-bit controle veld wat we de **ALUOp** noemen. De ALUOp bepaald de operatie add (00) voor laden en opslaan, trek af (01) voor beq. De output van de ALU controle eenheid is een 4-bit signaal dat direct de ALU bestuurd door het generen van een van de 4-bit combinaties. Zie figuur 4.4.1

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Voor de inputs die niet interessant zijn bestaan **don't-care termen**. Voor inputs die niet van belang zijn voor de output kan dan een X worden neergezet in de **truth table**. Zie figuur 4.4.2 hieronder voor een voorbeeld van een truth table.

ALUOp		Funct field						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Operation
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

Max Höfelmann

4.5 De Main Control Unit Designen

Om te snappen how je de connectie legt tussen velden van instructies naar een datapad is het handig om te kijken naar de formats van de drie intructie klassen: R-type, branch en load-store instructies. Er zijn meerdere belangrijke observaties over deze instructie formats:

- Het op veld heet de **opcode** altijd in bits 31:26 en wordt ook wel Op[5:0] genoemd.
- De twee registers die gelezen moeten worden zijn altijd gespecificeerd door de rs en rt velden op positie 25:21 en 20:16. Dit is zo voor R-type, branch equal en store instructies.
- Het basis register voor load en store instructies is altijd in bit positie 25:21 (rs).
- De 16-bit offset voor branch equal en load en store zitten altijd op positie 15:0.
- Het bestemming register zit altijd in een van twee plekken. Voor load in 20:16 (rt), R-type 15:11 (rd). Dus, we moeten een multiplexor toevoegen om te selecteren in welk vel we de instructie het nieuwe register nummer willen laten schrijven.

Met deze informatie kunnen we een instructie label en een extra multiplexor toevoegen aan het simpele datapad.

Waarom wordt Single-cycle implementatie vandaag de dag niet gebruikt? Aangezien de cpi wel 1 is zijn de cycles erg lang en is het gehele proces dus niet efficiënt. Dit zorgt ook voor een conflict met een van de regels uit hoofdstuk 1, **Make the common case fast**.

De rest van 4.4 is een lang voorbeeld van hoe je dit echt uitvoert, er is zo goed als geen nieuwe informatie.

R.A.J. Wacanno

4.6 Het Pijplijn-principe in een notendop

Het **pijplijnen** is een veelgebruikte manier om je architectuur instructies te laten uitvoeren.

Om te illustreren waarop dit pijplijnen berust kan men een viertal handelingen nemen: A, B, C en D welke na elkander en tezamen herhaaldelijk uitgevoerd moeten worden. Zonder er niet al teveel bij na te denken zou men de vier handelingen steeds na elkaar uit kunnen voeren (als ABCDABCD enz.). Dit is echter geen efficiënte strategie als de combinatie van handelingen vier of vijf keer uitgevoerd moet worden aangezien de tijd die dit zal kosten gelijk zal zijn aan $n \cdot (A + B + C + D)$; ofwel, de som van de tijd die iedere individuele handeling kost maal het aantal keer dat het herhaald moet worden.

Om deze herhaling van stappen in kortere tijd te laten verlopen kun je het principe van pijplijnen toepassen. Als de als voorbeeld genomen proces ABCD op deze manier wordt uitgevoerd zal dit er — horizontaal afgezet tegen de tijd — zo uit zien:

```
A  B  C  D
   A  B  C  D
      A  B  C  D
```

En zo zou deze serie handelingen er uitzien als het niet gepijplijnd was:

```
A  B  C  D  A  B  C  D  A  B  C  D
```

Als men deze twee diagrammen vergelijkt zijn er twee dingen die opvallen: wanneer de gepijplijnde aanpak benut wordt, is de totale uitvoeringsduur drastisch verminderd en bij het gepijplijnd uitvoeren lopen de individuele ABCD-processen parallel aan elkaar. In dit laatste schuilt dan ook de kracht van het pijplijnen. Door nadat een stap, zoals bijvoorbeeld A, voor de eerst set handelingen is uitgevoerd zal deze onmiddellijk herhaald worden voor de volgende en zo gebeurd dat ook met de andere stappen. Op deze manier wordt op ieder moment toch maar een van de stappen tegelijkertijd uitgevoerd, maar kan toch een significante versnelling gerealiseerd worden door het parallelliseren van de processen.

Onder perfecte omstandigheden zou de versnelling van de uitvoeringstijd van een collectie herhaalde processen als gevolg van pijplijnen gelijk zijn aan het aantal losse stappen waaruit ieder proces bestaat. In dat geval zou de volgende vergelijking kunnen worden opgesteld:

$$Tijd\ tussen\ instructies_{gepijplijnd} = \frac{Tijd\ tussen\ instructies_{niet\ gepijplijnd}}{Hoeveelheid\ pijplijn\text{-}stappen}$$

Met instructies worden hierboven processen als die van ABCD bedoeld

Er is echter vrijwel nooit sprake van perfecte omstandigheden. Het niet evenveel tijd kosten van individuele stappen en het feit dat er aan het begin en einde van iedere uitvoer minder instructies parallel kunnen lopen maakt dat de theoretisch beloofde versnelling niet haalbaar is. Een illustratie hiervan volgt alsmede de toepassing van pijplijnen binnen de MIPS-architectuur.

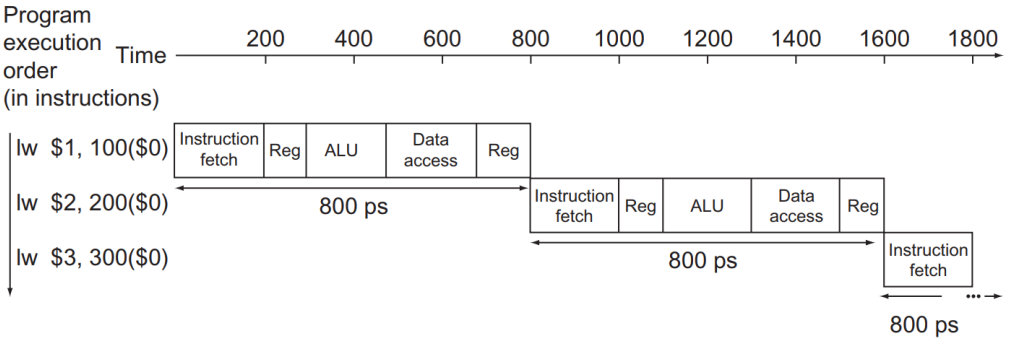
Zoals stappen A, B, C en D zijn er ook binnen een processor bepaalde handelingen die herhaaldelijk worden uitgevoerd. Dit zijn over het algemeen de volgende vijf stappen:

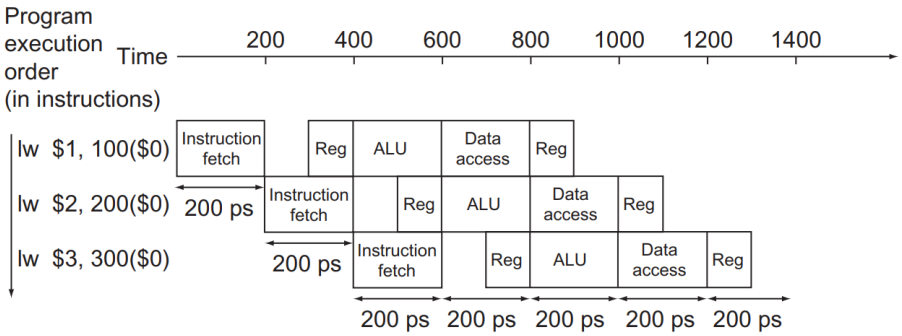
1. Het uit het geheugen halen van een instructie. *(Instruction fetch)*
2. Het lezen van registers terwijl de instructie gedecodeerd wordt; hetgeen bij MIPS-instructie van met de reguliere indeling gelijktijdig kan plaatsvinden. *(Register read)*
3. Het uitvoeren van de operatie of berekenen van een geheugenadres. *(ALU operation)*
4. Het benaderen van een operand in het data-geheugen. *(Data access)*
5. Het schrijven van de resultaten in een register. *(Register write)*

Als we nu ter illustratie de `lw`-instructie nemen heeft deze per stap de volgende hoeveelheid tijd nodig:

Instruction	*Instruction fetch*	*Register read*	*ALU operation*	*Data access*	*Register write*	*Total time*
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps

In de volgende twee figuren zijn zowel de gepijplijnde als de niet gepijplijnde herhaalde uitvoer van `lw` uitgezet tegen de tijd:





Door middel van pijplijnen kan de totale uitvoeringstijd van het bovenstaande voorbeeld van 2400 ps naar 1400 ps worden teruggebracht. Hoewel dit natuurlijk een significante verbetering is, is het niet in overeenstemming met de eerdergenoemde aanname dat de reductie in tijd gelijk zou staan aan het aantal individuele stappen.

De discrepantie tussen de theoretische en daadwerkelijke tijd zijn toe te schrijven aan de eerdergenoemde problemen op het gebied van pijplijnen. Allereerst zijn niet alle stappen even lang in duur. Wegens de aard van het pijplijnen is het cruciaal dat er bij het uitvoeren van stappen een bepaalde ritmiek zodat zodat deze steeds op gelijke interval worden uitgevoerd. Indien er dus een stap is die eerder voltooid is dan anderen moet er een korte rust ingevoegd worden om zo de ritmiek te handhaven. Als tweede valt de tijdswinst tegen als gevolg van het geringe aantal herhalingen. Een groter aantal herhalingen zou de verspilling van tijd die plaatsvindt aan het begin en einde van de uitvoer namelijk teniet doen.

Hier is een vergelijking van versnelling tussen — links — de 3 reeds beken herhalingen en — rechts — een hypothetisch aantal herhalingen van 1.000.000 inclusief de eerdere 3:

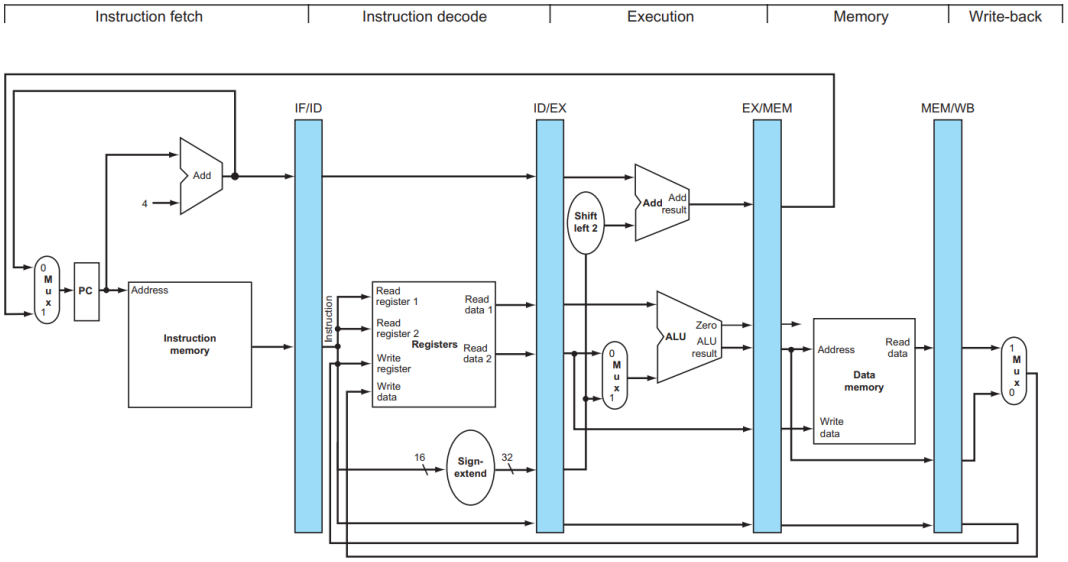
$$\frac{2.400\ ps}{1.400\ ps} = 1.71 \qquad \frac{800.002.400\ ps}{200.001.400\ ps} = 4.00$$

4.6.1 Instructiesets voor Pijplijnen

In het geval van MIPS, een instructieset expliciet bedoeld voor pijplijnen, zijn op vier fronten keuzes gemaakt die het gebruik voor pijplijnen enorm bevorderen. Allereerst zijn instructies gelijk in lengte. Ten tweede is er een gelimiteerd aantal instructie-indelingen. Ten derde komen geheugen-operanden enkel voor in *load*- en *store*-instructies. Ten vierde speelt ook de uitlijning van operanden in het geheugen hierbij een rol.

4.6.2 Pijplijnen m.b.t. het Datapad

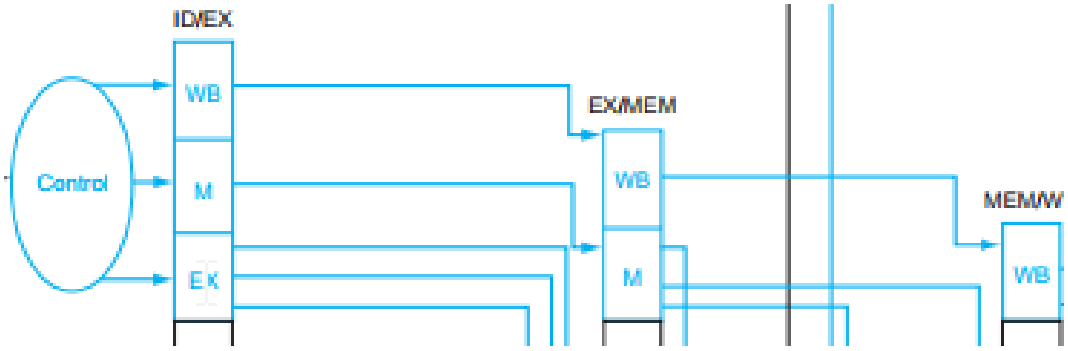
Nu het principe van pijplijnen is behandeld is het uiteraard van belang om te bekijken hoe men dit in hardware implementeert. Hieronder is de reeds bekende *single cycle*-architectuur weergegeven met daarin aanpassingen om pijplijnen mogelijk te maken:



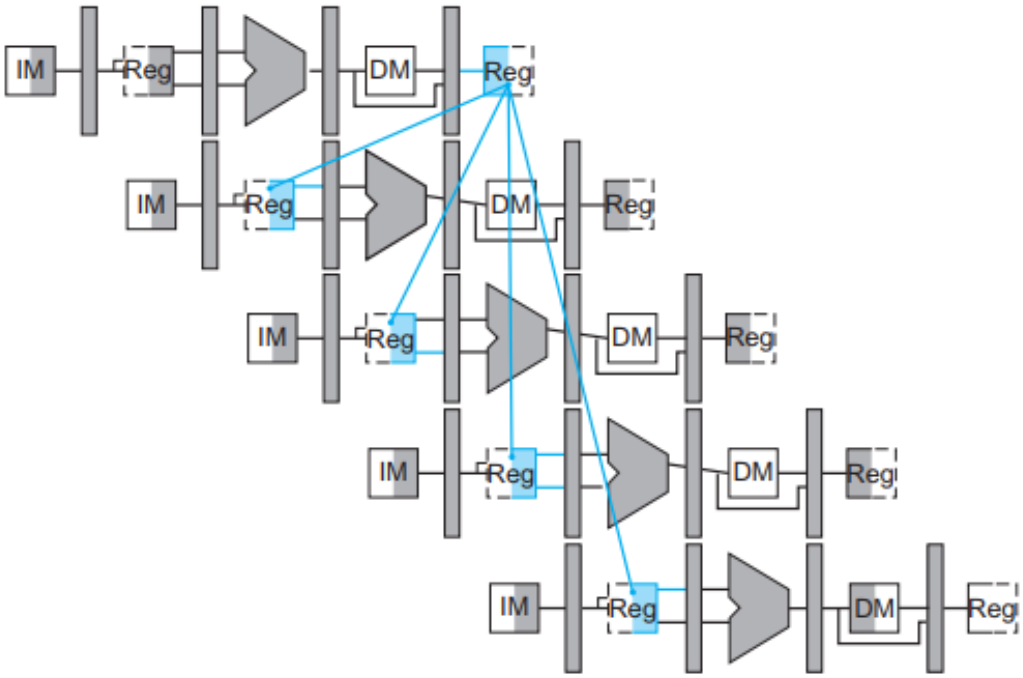
Hierin vallen uiteraard de blauwe balken op. Deze representeren kleine stukjes geheugen welke als buffers tussen de verschillende stadia van het pijplijn proces zitten. Waren deze er niet dan zou de uitvoer van ieder onderdeel, zodra deze was verkregen, als invoer naar het volgende component gestuurd worden; met als gevolg dat dit volgende component, daar het nog bezig is met een berekening, plots invoer krijgt welke het nog niet moet behandelen. De buffers zijn aangesloten op de klok en dit zorgt ervoor dat enkel de juiste invoer naar de verschillende componenten wordt gestuurd aan het begin van ieder stadium.

Het is ook goed om op te merken dat de stroom van data binnen de pijplijn altijd van links naar rechts is; met als enige uitzondering op deze regel het *write back*-stadium. Hier is in het bovenstaande diagram ook een stromingslijn zichtbaar vanaf de multiplexer terug naar het register. Omdat de data die tijdens dit stadium wordt teruggeschreven ook een registeradres met zich mee moet krijgen is de lusvormige stromingslijn zichtbaar boven de lijn van terugschrijving. Deze lus loop aan een kan ook door ieder van de buffers en komt zo op het zelfde moment weer terug bij het register als de data die in het register moet worden teruggeschreven.

Een ander belangrijk gegeven is dat de signalen vanuit de aansturingseenheid ook door de buffers gaan, dit is teven gedaan om de stroom door het datapad en de aansturingssignalen welke nodig zijn voor diens verwerking op het zelfde moment bij de betreffende componenten te laten arriveren. Dit alles is in het volgende figuur zichtbaar:



Zoals misschien al was opgemerkt, is het mogelijk dat binnen sommige instructies mogelijk dat het register op twee punten actief is. Normaliter zou dit een probleem vormen aangezien twee verschillende instructie het zelfde component nodig hebben, de snelheid van het register biedt hier uitkomst. Aangezien het aanroepen van het register qua tijdsduur tweemaal sneller is dan het bij een van de andere componenten zou zijn, kan dit component bunnen een cyclus tweemaal worden aangeroepen. In het onderstaande figuur is dit grafisch gedemonstreerd:



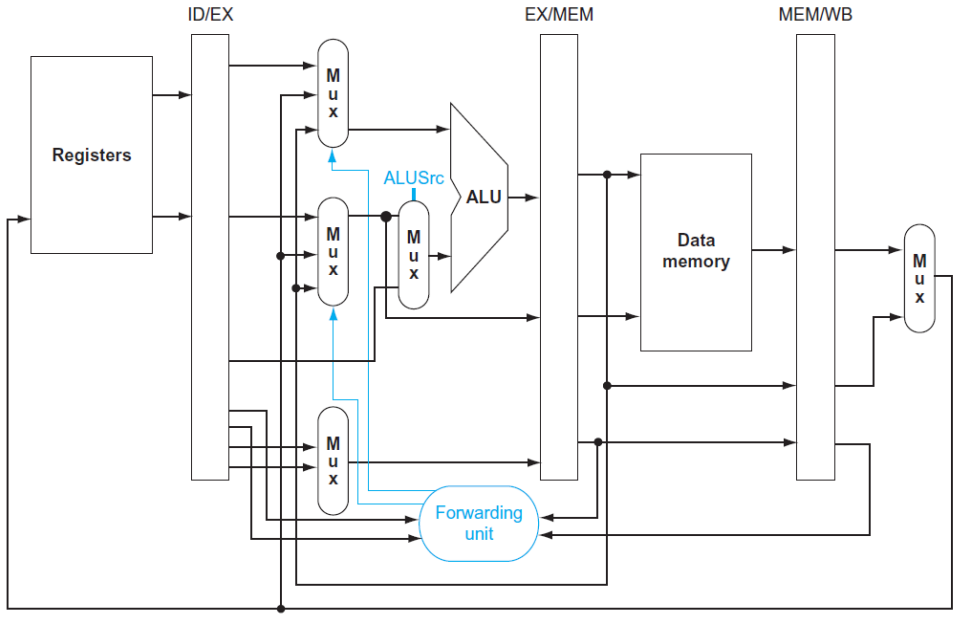
Max Höfelmann

4.7 Data Hazards: doorgeven vs ophouden

Hiervoor hebben we voorbeelden gezien die de kracht van pijplijnen laten zien. Nu gaan we kijken naar de problemen, de ***hazards***. Zie bijvoorbeeld hier:

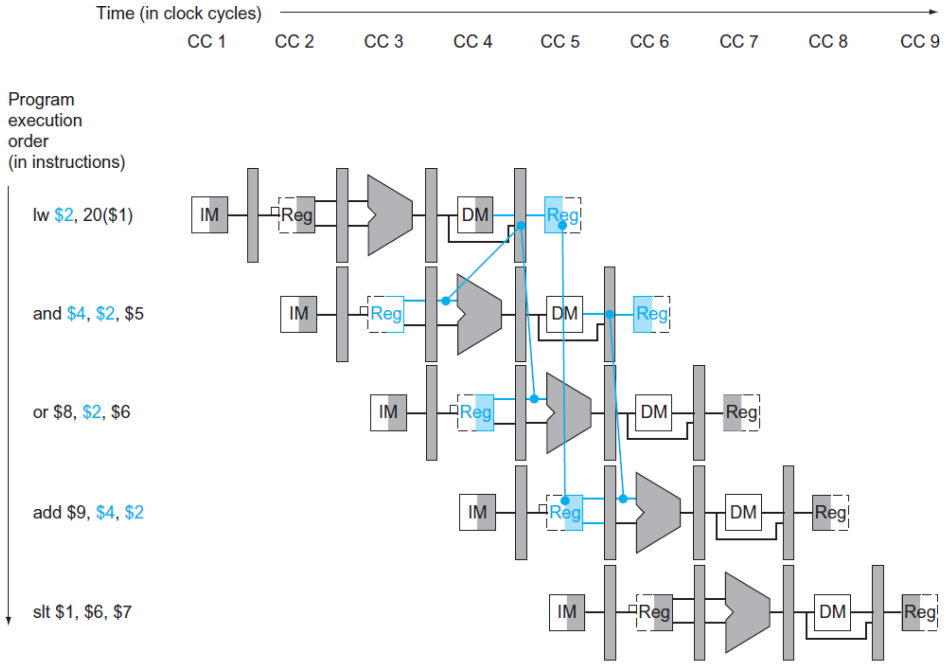
```
sub   $2, $1,$3      # Register $2 written by sub
and   $12,$2,$5      # 1st operand($2) depends on sub
or    $13,$6,$2      # 2nd operand($2) depends on sub
add   $14,$2,$2      # 1st($2) & 2nd($2) depend on sub
sw    $15,100($2)    # Base ($2) depends on sub
```

Alle 4 de instructies zijn afhankelijk van register 2. Als er door meerdere operaties in hetzelfde register wordt geschreven kan de volgende operatie de verkeerde waarde mee krijgen. Dit is op te lossen door bijvoorbeeld ***Forwarden***, oftewel doorgeven. Door een *forwarding* unit in het datapad te plaatsen kan de juiste waarde op het juiste moment worden doorgegeven aan elke instructie. We kunnen ook de SUB en AND instructies vervangen met LW en SW instructies om een ophouding te voorkomen. De data bestaat dan namelijk in het MEM/WB register door de LW instructie zodat de SW instructie het kan gebruiken.



Zoals hierboven te zien is zal bij de *forwarding unit* ook een *multiplexor* toegevoegd moeten worden, in dit geval een 2:1, deze *multiplexor* kiest dan de juiste register input voor de ALU.

Doorgeven werkt niet bij elke hazard die we kunnen ondervinden bij een pijplijn architectuur. Zie de afbeelding hieronder bijvoorbeeld, het register wordt nog gelezen bij *clock cycle* vier maar de ALU moet er wel al operaties mee uitvoeren. Hiervoor zal een ophouding, een ***stall***, toegevoegd moeten worden.



Voor zo'n ophouding toe te voegen kunnen we een instructie gebruiken die geen effect heeft: ***nop***. Door twee *nops* tussen elke instructie toe te voegen garandeer je dat er geen *hazard* meer is m.b.t. de registers.

Max Höfelmann

4.8 Control Hazards

Elke keer een ophouding plaatsen tot een *branch* compleet is is erg traag. Hierom kunnen we ook voorspellen of een *branch* beschikbaar is. Als de helft van de tijd een *branch* niet ingenomen is, dan kost het erg weinig om een instructie te verwijderen. Deze voorspel optimizatie zorgt er dan voor dat de kostte van de *control hazard* gehalveerd wordt. Om deze instructies te verwijderen worden alle waardes simpel weg naar 0 veranderd, het wordt eigenlijk een *stall*. Het verwijderen van instructies in een pijplijn door een onverwacht geval noemen we een ***flush***. We kunnen de snelheid van *branches* ook verbeteren door vooral in het begin van de pijplijn te *flushen* zodat er minder instructies *geflushed* hoeven te worden. Het verwachten dat een *branch* taken is noemen we ***static branch prediction***. ***Dynamic branch prediction*** daarentegen leert van zijn fouten. Als bij de vorige keer de instructie ook al een taken branch had dan onthoud de architectuur dit zodat de volgende keer nieuwe instructies opgenomen kunnen worden om zo de efficiëntie zo hoog mogelijk te krijgen. Om te onthouden of bij de vorige instructie een *branch* taken was of niet bestaat er de ***branch prediction buffer***, of *branch history table*. Hierin staat of een *branch taken* was in de recente instructies. ***branch delay slot*** is de plek die direct na een vertraagde *branch* komt. In de MIPS architectuur wordt deze gevuld door een instructie die geen invloed heeft op de *branch*. *Delayed branch* is gelimiteerd in twee opzichten:

- De restricties van de instructies die in de *delay slots* zitten.
- Onze capaciteit om te voorspellen tijdens het compileren of een *branch* beschikbaar is of niet.

Door deze limitaties is *delayed branch* in populariteit gekelderd. *Dynamic branch* is duurder maar flexibeler en door *Moore's law* snel verdubbelende transistoren wordt het wel steeds goedkoper.

Max Höfelmann

4.9 Exceptions

Een ***exception***, ook wel ***interrupt***, is een onverwachte actie dat de executie van het programma onderbreekt. Een *interrupt* kan ook van buiten de processor komen. Deze *interrupts* waren orignieel ontworpen om problemen in de processor aan te duiden zoals *overflow*. Er wordt in het boek een speciale terminologie gebruikt voor *exceptions* en *interrupts*, zie tabel.

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Voor het operatief systeem om een exceptie te herkennen moet de reden evenals de instructie die het verzorgde duidelijk gemaakt worden. Er zijn twee gebruikelijke methodes om dit te doen. In MIPS is dit door middel van een ***status register***. Anders kunnen ***vectored interrupts*** gebruikt worden. Het adres waarnaar toe de controle wordt gestuurd wordt bepaald door de exceptie bij een *vectored interrupt*. Het operatief systeem kan aan het adres (in hex) zien wat de error is, de adressen zijn gescheiden door 32 bytes of acht instructies.

Voor pijplijn excepties verloopt het iets anders. Als we de executie niet direct stoppen tijdens de instructie waar de *exception* plaats vind, gaat de oude waarde van het register verloren dat zorgde voor, bijvoorbeeld de *overflow*. Om de *exception* goed te kunnen werpen laten we de instructie door gaan alsof er niets aan de hand is en stoppen we de volgende *cycle* net voordat de instructie begint zodat we waardes uit het betreffende register verkregen kunnen worden. ***Imprecise interrupt / exception*** vind vaak plaats in pijplijn architecturen door deze niet precieze manier van excepties vinden. De oorzaak kan niet altijd accuraat gevonden worden namelijk. MIPS ondersteund samen met het grootste gedeelte van de huidige computers ***precise interrupts/ exceptions***, hier is de precieze oorzaak wel duidelijk.

Max Höfelmann

4.10 Parallelism via Instructions

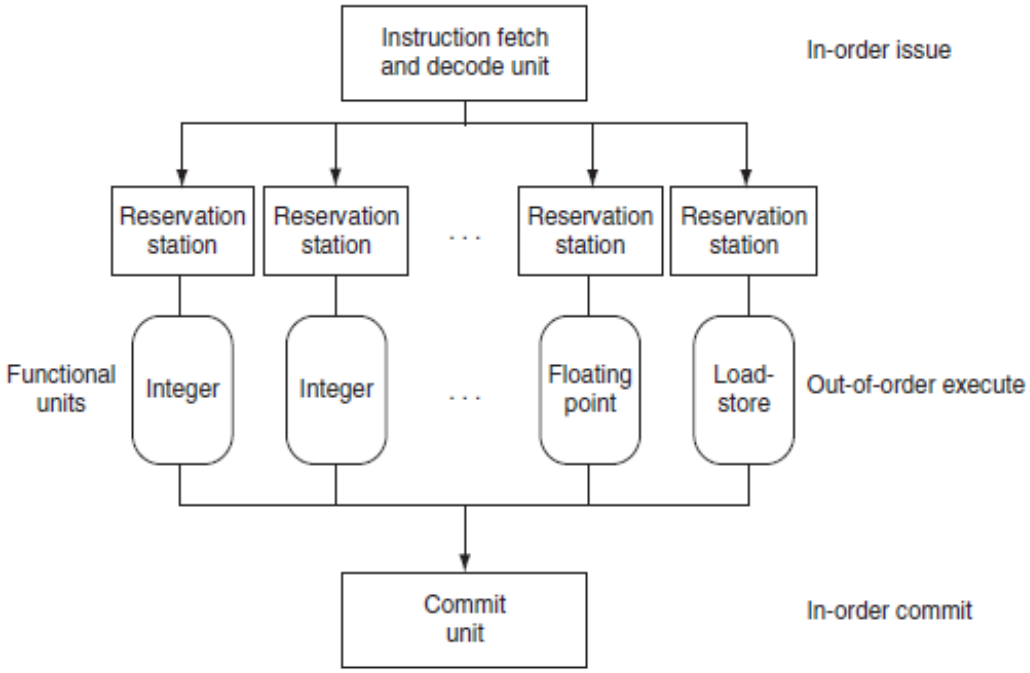
Pijplijnen maak gebruikt van het parallel verlopen van instructies. Dit wordt ook wel **parallelisme op instructie niveau (ILP)** genoemd. Er zijn hierbij twee primaire manieren om de hoeveelheid parallel verlopende instructies te verhogen.

- We kunnen de diepte van de pijplijn vergroten zodat meer instructies overlappen.
- ***Multiple issue*** is ook een optie, de interne componenten worden gekopieerd zodat meerdere instructies in elke pijplijn kunnen plaats vinden. Dus i.p.v. met een wasmachine en droger de was doen, doe je het nu met drie tegelijk! De enige keerzijde hier is dat er ook meer werk nodig is waardes te transporteren tussen pijplijnen omdat er veel meer waardes zijn.

Multiple issue heeft twee voorname implementaties, *Static* en *Dynamic*. Bij *static* worden de besluiten door de compiler gemaakt voor de executie. Bij *dynamic* vind dit plaats tijdens de executie door de processor.

Speculation is gebaseerd op het geweldige idee van voorspellen. Speculatie zorgt ervoor dat de processor kan gokken over verschillende eigenschappen van een instructie, hierdoor kunnen andere toepasselijke instructies meteen beginnen. Speculatie kan gedaan worden door middel van de *compiler* of de *hardware*. Helaas voegt ook dit weer een probleem toe aan de lijst, er kunnen excepties plaats gaan vinden op plekken waar er eigenlijk geen sprake van hoort te zijn. Er wordt dan fout gespeculeerd. Deze fouten kunnen vermeden worden door speciale speculatie support.

Dynamic Multiple-Issue processoren zijn ook wel bekend als ***superscalar*** processoren. Bij deze processoren wordt besloten of er een, nul, or meerder instructies tegelijk uitgevoerd moeten worden. Om de efficiëntie zo hoog mogelijk te houden maken deze processoren gebruik van ***dynamic pipeline scheduling***. Dit is *hardware* support voor het herordenen van de instructie executie om zo doende *stalls* te kunnen voorkomen. Hieronder zie je de werking van *Dynamic pipeline scheduling*.



De ***commit unit*** is waar er besloten wordt of het veilig is om het resultaat door te laten gaan. De ***reservation station***s zijn buffers waarin een functionele unit zit die de operanten en operatie vast houd. Deze unit wordt vaak de ***reorder buffer*** genoemd. Aangezien de instructies bij een *dynamic pipeline schedule* op een andere volgorde uitgevoerd kunnen worden wordt het ook wel een ***out-of-order execution*** genoemd. Als ze wel op volgorde worden weergegeven aan de programmeur is er spraken van *in-order commit*. Hedendaags gebruikt elke *dynamic pipeline schedule* de *in-order commit*.

Je vraagt je misschien af waarom *superscalar* processors gebruikt maken van *dynamic scheduling*. Er zijn die redenen:

- Niet elke stall is voorspelbaar.
- Als hij speculeerd is de exacte volgorde van instructies niet duidelijk tijdens het compileren.
- Doordat de pijplijn de hele tijd van *latency* en grootte veranderd is de meest efficiënte manier van compileren steeds anders, hierdoor is het uiteindelijk minder efficiënt.

Dynamic scheduling zorgt ervoor dat de *hardware* de meeste van deze problemen kan verbergen.

Max Höfelmann

4.11 Real Stuff: The ARM Cortex-A8 and Intel Core i7 Pipelines

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	14	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	2-level	2-level
1st level caches / core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level cache / core	128 - 1024 KiB	256 KiB
3rd level cache (shared)	–	2 - 8 MiB

Zie hierboven de specificaties van beide processoren.

Deze subsectie gaat voornamelijk over hoe beide processoren zich gedragen. Advies is om dit door te lezen in het boek zodat de stof duidelijker wordt. Hieronder een aantal begrippen die gebruikt worden in deze subsectie:

- ***Microarchitecture***: de organizatie van de processor, waaronder de grootte functionele *units*, hen connecties en controle.
- ***Architectural registers***: De instructieset van de zichtbare registers van de processor. Bijv. de 32 int en 16 *float point* registers in MIPS.

5 De Geheugenhiërarchie

R. van den Berge

5.1 Introductie

Programmeurs willen het liefst onbeperkt snel geheugen tot hun beschikking hebben. De onderwerpen van dit hoofdstuk beschrijven de illusie dat computers dat daadwerkelijk hebben. Hiervoor geldt het principe van geheugennabijheid (*principle of locality*). Voor dit principe is het belangrijk dat bij het uitvoeren van een programma niet alle data of code tegelijk opgehaald moet worden. Anders zou het onmogelijk zijn om geheugen snel te maken en alsnog groot te houden. Het is dus de bedoeling dat er bij uitvoer van een programma elke keer maar een relatief klein deel van het geheugen op te vragen.

Er zijn twee types geheugennabijheid:

- **Tijdelijke geheugennabijheid**: Dit principe stelt dat als een datalocatie is aangeroepen, dat het waarschijnlijk is dat deze locatie redelijk snel weer aangeroepen moet worden.
- **Ruimtelijke geheugennabijheid**: Dit principe stelt dat als een datalocatie is aangeroepen, dat datalocaties in de buurt van die locatie waarschijnlijk snel aangeroepen moeten worden.

Veel programma's gebruiken loops, wat resulteert in een hoog aantal tijdelijk plaatsgeheugen. Omdat instructies normaal gesproken achtereenvolgend opgevraagd worden, resulteert dit ook weer in een hoog aantal ruimtelijk plaatsgeheugen. Bijvoorbeeld bij elementen uit een array wordt er gebruik gemaakt van ruimtelijke geheugennabijheid

We profiteren van het principe van plaatsgeheugen door een **geheugenhiërarchie**. Deze bestaat uit lagen van verschillende groottes en snelheden van geheugen. Het snellere geheugen is duurder dan het langzamere en daardoor kleiner. Het doel is om met zo veel mogelijk goedkoop geheugen zo snel als het snelste geheugen te werken. Data heeft een soortgelijke hiërarchie, hoe verder van de processor verwijderd, hoe langzamer. Data is het verst van de processor opgeslagen en wordt dus dichterbij de processor gehouden op het moment dat het nodig is.

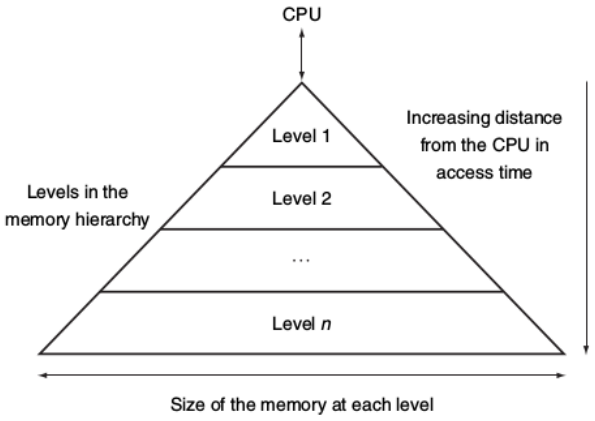


FIGURE 5.3 This diagram shows the structure of a memory hierarchy: as the distance from the processor increases, so does the size. This structure, with the appropriate operating mechanisms, allows the processor to have an access time that is determined primarily by level 1 of the hierarchy and yet have a memory as large as level *n*. Maintaining this illusion is the subject of this chapter. Although the local disk is normally the bottom of the hierarchy, some systems use tape or a file server over a local area network as the next levels of the hierarchy.

Een data-hiërarchie werkt met twee lagen tegelijkertijd, dus we focussen ons op twee lagen. De bovenste laag is sneller dan de onderste en ligt dichter bij de processor. Het minimum aan informatie dat aanwezig of niet aanwezig kan zijn in de tweelaagse hiërarchie heet een **blok (*block*)** of een **regel (*line*)**. Als opgevraagde data zich bevindt in de bovenste laag heet dit een ***hit***, als dit niet zo is heet het een ***miss***. Bij een miss wordt de onderste laag ingeschakeld om de opgevraagde data op te halen.

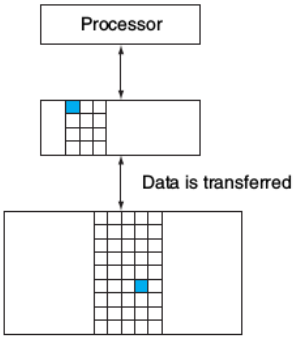


FIGURE 5.2 Every pair of levels in the memory hierarchy can be thought of as having an upper and lower level. Within each level, the unit of information that is present or not is called a *block* or a *line*. Usually we transfer an entire block when we copy something between levels.

De ***hit rate*** is de kans op een hit in de bovenste laag. De ***miss rate*** $(1 - hitrate)$ is de kans op een miss. ***Hit time*** is de tijd die nodig is om de bovenste laag te checken voor een hit. De ***miss penalty*** is de tijd die nodig is om bij een miss een blok uit de bovenste laag te vervangen door het desbetreffende blok uit de onderste laag. Voor een goede performance moeten de *hit rate* en *hit time* goed genoeg zijn om te compenseren voor de *miss penalties* die plaatsvinden.

R. van den Berge

5.2 Geheugen Technologieën

Er zijn vier primaire technologieën in de geheugenhiërarchieën:

- **SRAM technologie**:
  SRAMs zijn geïntegreerde circuits die geheugenreeksen zijn met een enkele toegangspoort die een *read* of *write* kan uitvoeren. SRAMs hoeven niet ververst te worden dus is de tijd om het te gebruiken bijna gelijk aan de *cycle* tijd. SRAM heeft maar een minimaal aantal energie nodig om zijn data te bewaren in stand by mode. Zolang er stroom op zit wordt de data bewaard. SRAM gebruikt 6 tot 8 transistors per bit om te voorkomen dat de informatie wordt verstoord.
- **DRAM technologie**:
  De waarden van data op DRAM worden bewaard in een condensator. Een enkele transistor wordt vervolgens gebruikt om deze data te lezen of te overschrijven. Omdat er hier maar één transistor wordt gebruikt is de dichtheid van DRAM veel hoger dan SRAM en daardoor dus goedkoper. Omdat DRAMs de data opslaan op een condensator moet de data om de zoveel tijd ververst worden. Om dit te doen wordt de data gelezen en opnieuw opgeslagen. Dit gebeurt om de 8 milliseconden. Hiervoor wordt een tweelaagse codeer structuur gebruikt, waardoor een hele rij aan data ververst kan worden.
- **Flashgeheugen**:
  Flash type is een EEPROM (*electrically erasable programmable read-only memory*) memory type. De naam *flash* komt doordat het geheugen in een keer deels of volledig verwijderd kan worden om er iets nieuws in te op te slaan. Dit wordt gebruikt voor o.a. USB-sticks.
- **Schijfgeheugen**:
  Een magnetische harde schijf bestaat uit allemaal hele dunne metalen platen, die draaien met van 5400 tot 15000 RPM. Deze metalen platen zijn bedekt met magnetisch materiaal om dingen op te slaan. Het oppervlak van zo'n cirkel is onderverdeeld in ***tracks***, dit zijn concentrische cirkels die met tienduizenden het oppervlak vormen. **Sectoren** is de minimale data die opgeslagen of gelezen kan worden (normaal gesproken 512 - 4096 bytes).
  De performance hangt af van 3 stappen:
  - ***Seek***:
    Dit is het plaatsen van de leesnaald van de schijven boven de juiste datalocatie.
  - ***Rotational delay***:
    Er moet worden gewacht tot de sector van de data op de schijf onder de naald gedraaid is. Gemiddelde rotational delay $= \frac{0.5 rotation}{rotationspersecond}$
  - ***Transfer time***:
    De *transfer time* is de tijd voor het verplaatsen van een blok bits.

R. van den Berge

5.3 De basis van caches

De cache is een plek om veilig dingen op te slaan. Er zijn bij het kijken of het opgevraagde item al aanwezig is in de cache twee vragen:

- Hoe weten we of het item al aanwezig is in de cache?
- En zo ja, hoe vinden we die?

De antwoorden op de vragen hangen samen. Als een *word* maar op één plek in de cache opgeslagen kan worden, dan is het heel makkelijk deze ook weer terug te vinden. De locatie waar de *word* wordt opgeslagen is gebaseerd op het adres van de *word*. Deze structuur heet ***direct mapped***. De bijna alle *direct-mapped caches* gebruiken de volgende *mapping* om een blok te vinden:

- (Blok adres) modulo (het aantal blokken in de cache).

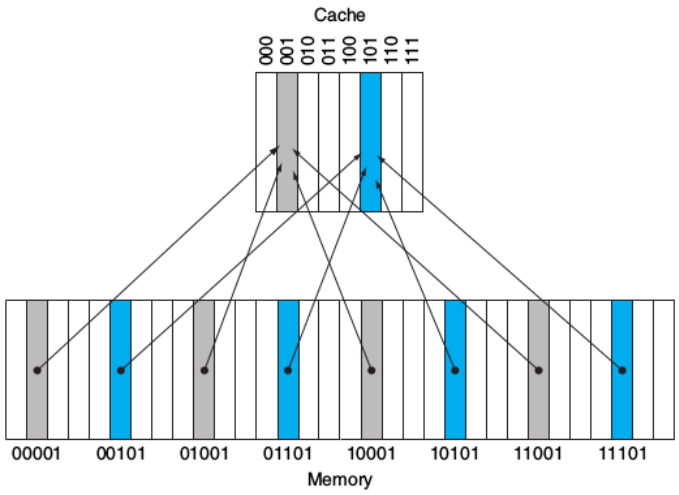


FIGURE 5.8 A direct-mapped cache with eight entries showing the addresses of memory words between 0 and 31 that map to the same cache locations. Because there are eight words in the cache, an address X maps to the direct-mapped cache word X modulo 8. That is, the low-order $\log_2(8) = 3$ bits are used as the cache index. Thus, addresses 00001_{two}, 01001_{two}, 10001_{bro}, and 11001_{two} all map to entry 001_{bro} of the cache, while addresses 00101_{bro}, 01101_{bro}, 10101_{bro}, and 11101_{bro} all map to entry 101_{bro} of the cache.

Als het aantal blokken in een cache een macht van twee is, bijvoorbeeld 16, dan gebruikt de cache de laagste $\log_2 16 = 4$ bits van een blok voor het adres. Omdat hierdoor nu bij een adres in de cache meerdere dingen aanwezig kunnen zijn, kunnen we de juiste data vinden door middel van ***tags***. Om te kijken of een cache leeg is gebruik je een ***valid bit***. Als deze 1 is, dan is de data aanwezig. 0 betekent niet aanwezig.

Er moet een balans worden gevonden tussen de blok grootte en de hoeveelheid blokken in de cache. Als de blok grootte te groot wordt gaat de *miss rate* omhoog omdat er dan minder blokken in de cache kunnen.

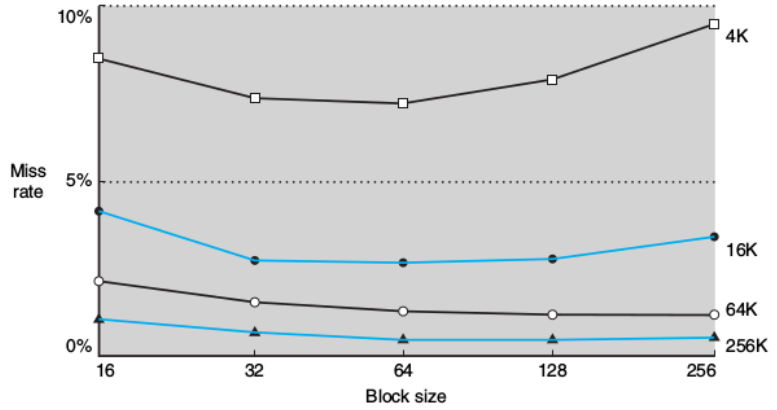


FIGURE 5.11 Miss rate versus block size. Note that the miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. (This figure is independent of associativity, discussed soon.) Unfortunately, SPEC CPU2000 traces would take too long if block size were included, so this data is based on SPEC92.

Bij een cache miss wordt de CPU pijplijn uitgesteld. Hierna wordt het blok uit de memory geladen en in de cache opgeslagen. Als data vanuit het geheugen alleen in de cache wordt opgeslagen, dan staat er na een tijd andere data in de cache dan in het geheugen (inconsistente data). Hierdoor moet dezelfde data in de cache én het geheugen worden aangepast, waardoor de data weer consistent wordt. Dit heet ***write-through***.

Als dit elke keer moet gebeuren dat er data verandert kost dit veel tijd. Een oplossing hiervoor is een ***write buffer***. Dit is een rij aan (aangepaste) data die wacht tot hij kan worden opgeslagen in het geheugen.

Een alternatief voor een *write-through* is een ***write-back***. Hier wordt data alleen aangepast in het blok in de cache. Op het moment dat het blok uit de cache wordt gehaald, wordt de data in het geheugen overschreven.

R. van den Berge

5.4 Het meten en verbeteren van cache performance

De cache performance hangt van een aantal dingen af:

- $CPU\ time = (CPU\ execution\ clock\ cycles + Memory\text{-}stall\ clock\ cycles) \cdot Clock\ cycle\ time$
- $Memory\text{-}stall\ clock\ cycles = \frac{Memoryaccesses}{Program} \cdot miss\ rate \cdot Read\ miss\ penalty$
  $= \frac{Instructions}{program} \cdot \frac{Misses}{Instruction} \cdot Miss\ penalty$
- $Memory\text{-}stall\ clock\ cycles = (Read\text{-}stall\ cycles + Write\text{-}stall\ cycles)$
  - $Read\text{-}stall\ cycles = \frac{Reads}{Program} \cdot Read\ miss\ rate \cdot Read\ miss\ penalty$
  - $Write\text{-}stall\ cycles = \left(\frac{Writes}{Program} \cdot Write\ miss\ rate \cdot Write\ miss\ penalty\right) + Write\ buffer\ stalls$
- $AMAT\ (average\ memory\ access\ time) = Time\ for\ a\ hit + Miss\ rate \cdot Miss\ penalty$

Om de cache performance te verbeteren kun je een aantal dingen doen:

- De *hit time* verlagen door:
  - Kleinere cache
  - *Direct mapped* cache
  - Kleinere blokken
  - *Write buffers* gebruiken
- De *miss rate* verlagen door:
  - Grotere cache
  - Flexibelere plaatsing (*associativity* gebruiken)
  - Grotere blokken

  - Een 'slachtoffer' cache gebruiken om de meest recent weggegooide cache blokken in op te slaan.

- De *miss penalty* verlagen door:
  - Kleinere blokken
  - *Write buffers* gebruiken
  - De *write buffer* bekijken bij een miss, misschien zit het daar wel in voordat je naar het geheugen gaat
  - Bij grotere blokken kijken naar het belangrijkste *word*
  - Meerdere cache lagen gebruiken die niet gebonden zijn aan *clock rate*
  - Bandbreedte van geheugen verbeteren.

Direct mapped cache had dus maar één adres om blokken in op te slaan. Bij een ***fully associative*** cache kunnen blokken op elke plek worden opgeslagen. Hierbij moet dus wel elk blok afgezocht worden voor een hit, omdat blokken op elke plek kunnen worden opgeslagen. Een middenweg is ***set associative***. Bij een *set associative* cache is er een aantal locaties waar een blok geplaatst kan worden. Een *set-associative* kan een n aantal sets hebben. Deze bestaan ook weer uit een n aantal blokken. Blokken worden dus gekoppeld aan een set, en daarin kan een blok dus op elke plek worden opgeslagen. Hierbij geldt de dat de set waarin een blok is opgeslagen gevonden kan worden door

- (Blok nummer) modulo (aantal sets in de cache)

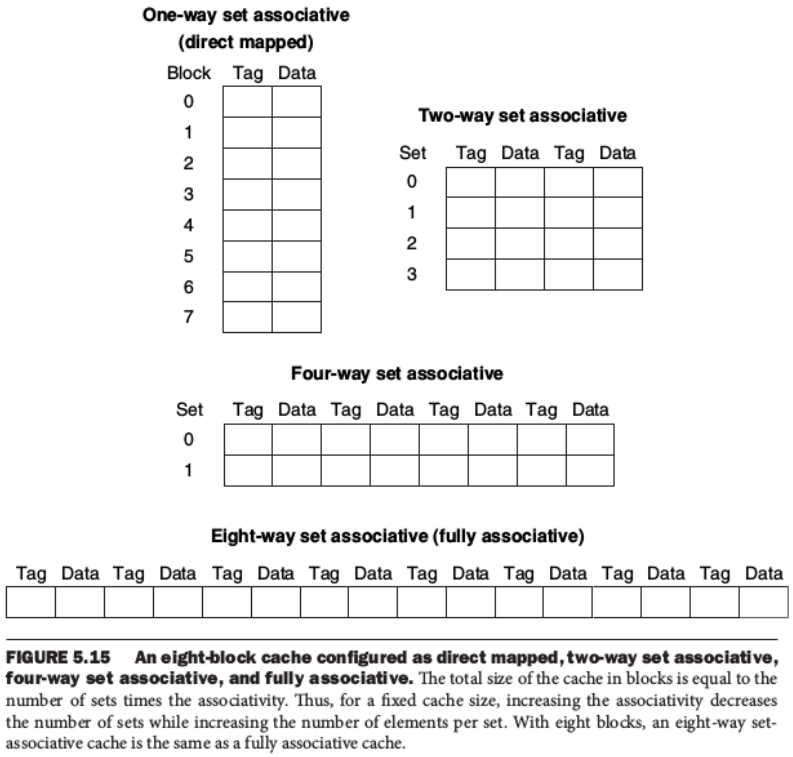


FIGURE 5.15 An eight-block cache configured as direct mapped, two-way set associative, four-way set associative, and fully associative. The total size of the cache in blocks is equal to the number of sets times the associativity. Thus, for a fixed cache size, increasing the associativity decreases the number of sets while increasing the number of elements per set. With eight blocks, an eight-way set-associative cache is the same as a fully associative cache.

Dus, bij *fully associative* moeten alle tags worden afgezocht, en bij *set associative* slechts alle tags binnen een set. Bij het vervangen van een blok wordt de **LRU (*least recently used*)** vervangen. De computer kijkt dus welk blok relatief het minst belangrijk is en vervangt deze op het moment dat dat nodig is.

R.A.J. Wacanno

5.5 Virtueel Geheugen

Nu de bovenste lagen van de geheugenhiërarchie uitvoering behandeld zijn, is het ook van belang om te bespreken hoe de onderste lagen van deze piramide geoptimaliseerd kunnen worden om zo tot de gewenste prestaties van een architectuur te komen.

Een principe welke een cruciale rol speelt bij het *RAM (Random Access Memory)* en de secundaire opslag, in de vorm van een *SSD-* (*Solid State Drive*) of harde schijf, is die van het **virtuele geheugen**. Dit virtuele geheugen is in de loop der tijd tot stand gekomen ter oplossing van de volgende problematiek:

- **Het veilig delen van het geheugen tussen verschillende processen**

  Tijdens het schrijven en compileren van een programma kan niet voorspeld worden met welke programma's het bij het uitvoeren het geheugen zal moeten delen. Sterker nog, zelfs tijdens het draaien van het programma kunnen deze omstandigheden sterk veranderen.

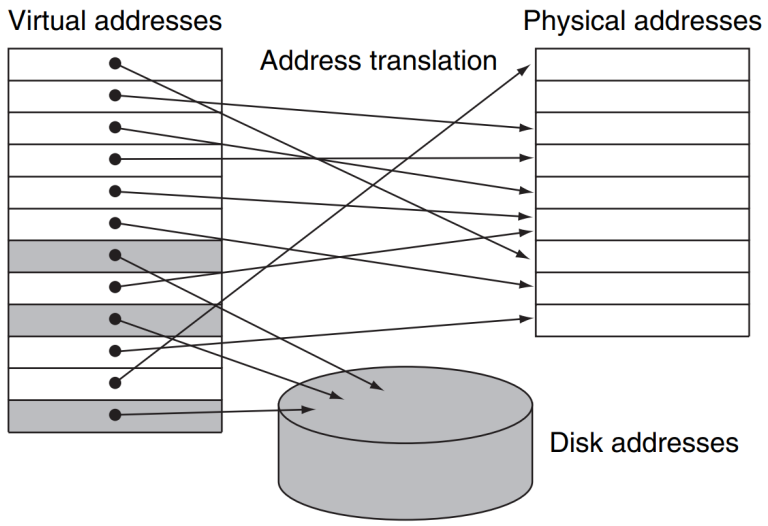
  Om dit soort situaties te voorkomen stelt het virtuele geheugen de individuele programma's in staat een eigen **adresseringsruimte** te benutten. Het virtuele geheugen draagt zorg voor het omzetten van locaties binnen deze adresseringsruimte naar **fysieke geheugenadressen**.

- **Het oplossen van problematiek als gevolg van een gelimiteerde hoeveelheid geheugen**

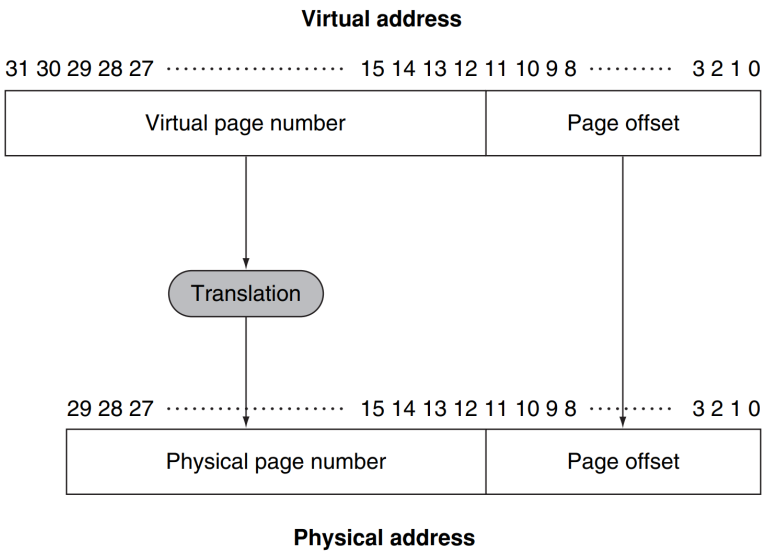
  Dikwijls komt het voor dat een programma meer geheugen nodig heeft—zij het voor data, zij het voor instructies—dan dat er binnen het systeem beschikbaar is. Ook hiervoor biedt het virtuele geheugen een oplossing. Het eerdergenoemde omzetten van geheugenadressen geldt namelijk niet alleen tussen virtuele en fysieke adressen, maar ook tussen **virtuele adressen** en adressen van secundaire opslagapparaten. Zo kan een draaiend programma dus niet alleen gebruikmaken van het RAM, maar ook van een naar alle waarschijnlijkheid aanwezige secundaire opslageenheid.

Om het virtuele geheugen beter te kunnen begrijpen is het verstandig een aantal termen en hun bijbehorende definities te belichten. Allereerst is er de **pagina** (*page*). Dit is een blok geheugen dat in het virtuele geheugen aanwezig is. Wanneer een pagina die is opgevraagd met behulp van een virtueel adres niet in het *RAM* aanwezig is ontstaat er een **pagina-fout** (*page fault*).

Een voorbeeld van het verbinden van een virtueel en een fysiek adres is in het onderstaande plaatje schematisch uitgebeeld:



In het nu volgende plaatje is de opbouw van zowel virtuele als fysieke adressen en de tussenliggende omzetting geïllustreerd:



Zoals te zien is, is het virtuele adres opgebouwd uit een **virtueel paginanummer** en een **pagina-afstand**. Het fysieke adres is opgebouwd uit een **fysiek paginanummer** en dezelfde pagina-afstand.

Het is van belang het een en ander in beschouwing te nemen wanneer men kijkt naar dit figuur. Allereerst is aan de lengte van de paginanummers te zien dat het virtuele adres een groter pagina-adresseringsbereik heeft dan het fysieke adres. Dit spreekt voorzich, aangezien—zoals eerder gezegd—de adresseringsruimte groter kan zijn dan het geheugen dat in een systeem aanwezig is. Wanneer er een virtueel adres opgevraagd wordt die niet aan een locatie in het geheugen is gekoppeld zal deze dus uit de secundaire opslag geladen moeten worden (een pagina-fout).

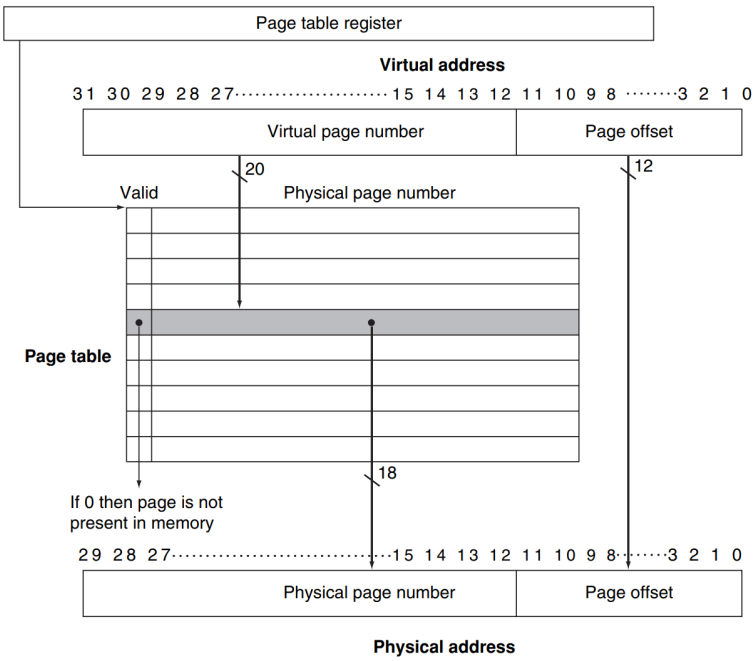
Dan is er ook te zien dat dat de pagina-afstand in beide adresvormen met elkaar correspondeert; hetgeen logisch is omdat pagina's in het virtuele en het fysieke geheugen dezelfde grootte hebben. Het is daarom dat deze nummers niet omgezet hoeven te worden en een locatie binnen een pagina in het virtuele en het fysieke geheugen met het zelfde pagina-afstandsnummer opgevraagd kan worden.

Bij het ontwerpen van een virtueel geheugensysteem zijn er een aantal zaken die in acht genomen moeten worden, aangezien het geheugen in sommige gevallen wel 100.000 maal sneller kan zijn dan de secundaire opslag en een pagina-fout dus zeer kostbaar kan zijn.

- Het loont om pagina's groot genoeg te maken zodat de lange opvragingstijd uit de secundaire opslag teniet wordt gedaan.
- Veelal worden pagina's volledig associatief geordend om de frequentie van pagina-fouten te verminderen.
- Pagina-fouten kunnen door de *software* worden afgehandeld omdat de tijd die dit kost niet significant is vergeleken met de tijd die het kost om de betreffende pagina uit de secundaire opslag te laden. Ook is *software* in staat om pagina's dusdanig te ordenen om zo de kans op pagina-fouten te verminderen.
- Omdat het schrijven naar de secundaire opslag te lang duurt maakt virtueel geheugen gebruik van *write-back* in plaats van *write-through*.

5.5.1 Het Plaatsen en Vinden van Pagina's

Het omzetten van de virtuele naar fysieke adressen gebeurt met behulp van de **pagina-tabel** (*page table*). Dit is een stuk geheugen dat geïndexeerd is op basis van de virtuele pagina-nummers. Deze index is vervolgens gekoppeld aan het corresponderende fysieke pagina-nummer. Ieder programma bezit zijn eigen pagina-tabel en deze is bij uitvoer te vinden dankzij het **pagina-tabelregister**. Dit register wijst naar het begin van de desbetreffende pagina-tabel.

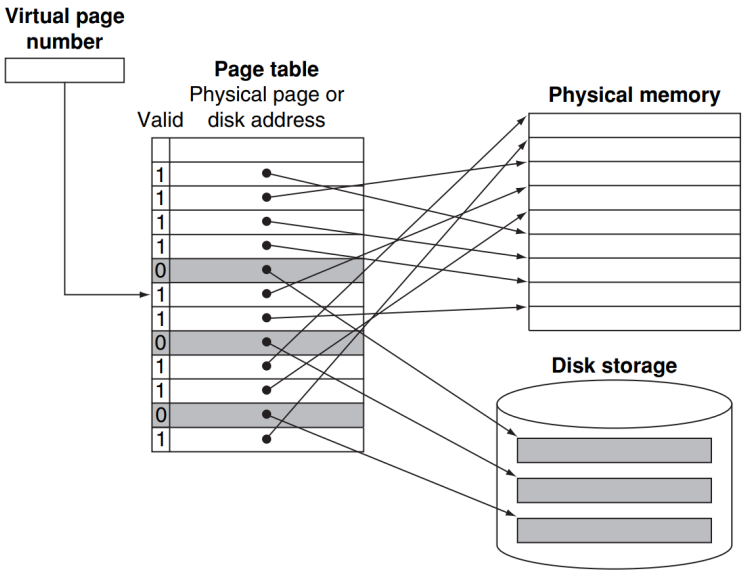


Het bovenstaande figuur toont de eerder geziene omzetting van virtuele naar fysieke adressen, maar nu is hier de pagina-tabel en het pagina-tabelregister aan toegevoegd. Ook is hierin de zogenaamde ***valid bit*** weergeven. Deze bit geeft aan of de met het fysieke pagina-nummer corresponderende pagina in het geheugen aanwezig is. Wanneer deze bit 1 is, is dit het geval. Indien deze 0 is, zal er een pagina-fout optreden.

5.5.2 Pagina-fouten in Detail

Bij het omgaan met pagina-fouten is de centrale rol van het **besturingssysteem** goed te zien. Wanneer een dergelijke fout namelijk optreed is het de taak van het besturingssysteem om dit in goed banen te leiden. Het besturingssysteem laad de gewenste pagina uit het geheugen maar kan dit niet alleen met het virtuele adres.

Wanneer een programma wordt gestart, maakt het besturingssysteem een ***swap space*** aan in de secundaire opslag waarin alle pagina's van het programma kunnen staan. Tevens houdt het besturingssysteem bij welke virtuele adressen naar welke locaties in deze *swap space* verwijzen. Dit alles is in het onderstaande figuur weergegeven:



Ook registreert het besturingssysteem welke processen en virtuele adressen aan welke fysieke adressen gekoppeld zijn.

Het kan zo nu en dan voorkomen dat het geheugen vol is, maar er voor een bepaald proces toch een pagina uit de secundaire opslag geladen moet worden. In dat geval zal het besturingssysteem op basis van principe van het **minst recent gebruikte** (LRU) een pagina vervangen. Het idee hierachter is dat op deze manier de kans op een pagina-fout geminimaliseerd wordt.

5.5.3 Terugschrijven naar Secundaire Opslag

Zoals eerder gezegd baseert het virtuele geheugen het schrijven naar secundair geheugen niet op een ***write-through*** maar op een ***write-back***. Dit betekend dat dat iedere individuele pagina-aanpassing naar het geheugen geschreven wordt en de desbetreffende pagina naar de secundaire opslag wordt gekopieerd wanneer deze in het geheugen wordt vervangen door een andere pagina.

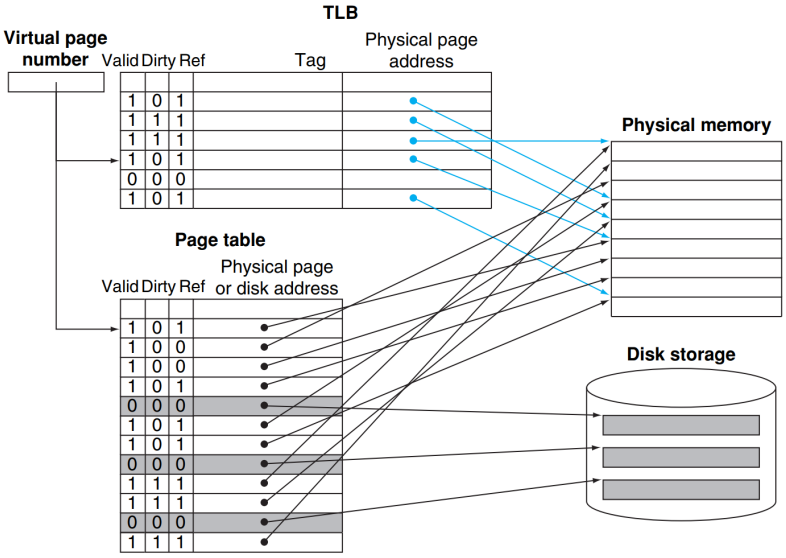
5.5.4 Versnelling van Adresomzetting

Aangezien het opvragen van data uit een pagina op basis van het geheugen tweemaal de opvragingstijd van het geheugen kost—omdat zowel het fysieke adres als de data uit het geheugen moeten worden geladen—word in systemen gebruik gemaakt van een speciale *cache* die dit proces versneld. Deze cache, onder de noemer ***translation-lookaside buffer*** (TLB), fungeert op basis van tijdelijke en ruimtelijke geheugennabijheid. Verondersteld wordt namelijk dat, wanneer een virtueel paginanummer is omgezet, de kans groot is dat deze pagina in de toekomst weer gebruikt zal worden

Bij het opvragen van data zal dus eerst in de TLB worden gekeken. Om dit goed te laten verlopen zin naast de *valid bit* nog een aantal andere controle-bits nodig.

- ***Dirty bit***: Deze bit krijgt een waarde van 1 als er een schrijfactie plaatsvindt op de geheugenlocatie van dit adres.
- **Referentie-bit**: Deze bit krijgt een waarde van 1 als het desbetreffende adres in de TLB aanwezig is.

In het figuur hieronder wordt deze samenhang weergegeven:

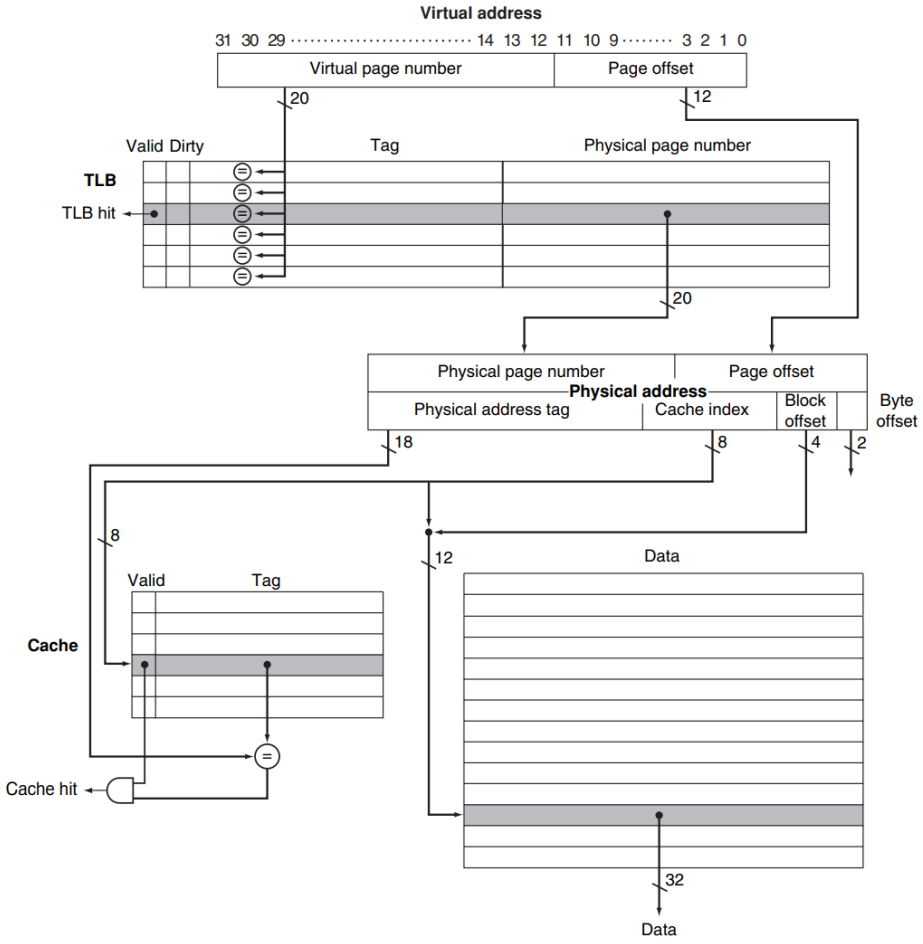


Wanneer een adres in de *TLB* wordt opgevraagd en deze is niet aanwezig is er sprake van een *TLB miss* en dit kan twee oorzaken hebben:

- **Het adres mist enkel in de *TLB*:** In dit geval wordt de processor aan het werk gezet om het missende adres alsnog in de *TLB* te laden.
- **Er is sprake van een pagina-fout:** Indien dit gebeurd is het—zoals eerder beschreven—de taak van het besturingssysteem om de pagina-fout af te handelen.

5.5.5 Samenbrengen van Virtueel Geheugen, *TLB* en Cache

Uiteindelijk is het dan nog zaak om de hogere en de onderste lagen van de geheugenhiërarchie met elkaar te verenigen. Dit wordt in onderstaande afbeelding schematisch toegelicht:


Virtual address
31 30 29 ····· 14 13 12 11 10 9 ····· 3 2 1 0
Virtual page number
Page offset
20
12
Valid Dirty
Tag
Physical page number
TLB
TLB hit
20
Physical page number
Page offset
Physical address
Physical address tag
Cache index
Block offset
Byte offset
18
8
4
2
8
12
Data
Valid
Tag
Cache
Cache hit
32
Data

6 Parallelle Processors van Client tot Cloud

C.S. Bras

6.1 Introductie

Hoge performance van een computersysteem kan betekenen dat de doorvoer van onafhankelijke taken hoog is, dit is waar ***multicore processors*** van pas komen. In dit hoofdstuk worden toepassingen van deze processors besproken en manieren hoe deze het best te programmeren zijn.

6.2 De Moeilijkheid van Parallelle Programma's Maken

Het probleem met parallellisme is niet zo zeer de hardware die hiervoor nodig is maar het feit dat programmeren voor parallelle doeleinden vele malen lastiger is dan voor niet parallelle versies. Een van de redenen hiervan is dat er een dusdanige verbetering moet zijn in de performance of het energie verbruik, ten opzichte van een sequentiële versie, om het nuttig te stellen de extra moeite te nemen om een programma parallel te laten uitvoeren. Een ander probleem dat komt kijken bij het veelvoudig parallel maken van processen is de overhead die nodig is om communicatie tussen meerdere processor cores goed te laten verlopen.

Een snelheidsverbetering bij parallelle programmatuur is makkelijker te verkrijgen wanneer het probleem mee groeit met het aantal parallelle processen. De termen **strong scaling** en **weak scaling** worden gebruikt om de snelheidswinst te beschrijven bij het niet mee schalen van een probleem en het proportioneel mee schalen van een probleem met het aantal processors, respectievelijk.

C.S. Bras

6.3 SISD, MIMD, SIMD, SPMD en Vector

Parallelle hardware kan opgedeeld worden in de volgende categorieën:

- ***SISD*** *Single Instruction stream Single Data stream*, dit is een standaard *singlecore processor*.
- ***MIMD*** *Multiple Instruction streams Multiple Data streams*, dit is een standaard *multicore processor*.
- ***SPMD*** *Single Program, Multiple Data streams*, dit is een manier van een *MIMD* processor te programmeren waarbij een programma op meerdere processors tegelijk draait die vervolgens samenwerken door middel van conditionele statements om zo te bepalen wanneer welke processor welk stuk code uitvoert.
- ***SIMD*** *Single Instruction stream Multiple Data streams*, bij een *SIMD* computer wordt dezelfde instructie uitgevoerd op meerdere stukken data, ofwel een *vector*, waardoor deze allemaal op het zelfde moment parallel uitgevoerd worden. *SSE instructies* voor de *x86 instructieset* zijn hier een vorm van.

Alle parallelle *SIMD* units staan onderling gesynchroniseerd en luisteren allemaal naar dezelfde instructie die afkomstig is van een enekele *program counter*. Dit lijkt vanuit het standpunt van een programmeur erg veel op *SISD*.

6.3.1 Vector

Een zogenaamde *vector architectuur* is een oudere interpretatie van *SIMD*. In plaats van een grote hoeveelheid *ALU's* te gebruiken om parallel addities uit te voeren, wordt de data in *vector registers* geladen waarna dit sequentieel wordt bewerkt door *pipelined execution units* om vervolgens weer terug gestopt te worden in het geheugen.

6.3.2 Vector versus Scalar

Vector instructies hebben enkele belangrijke eigenschappen ten opzichte van *scalar* instructies.

- Een *vector* instructie bevat veel werk, wat normaal gesproken een loop zou zijn wordt nu bevat in een instructie, zo is er minder bandbreedte nodig voor het *fetchen* en *decoden* van instructies.
- Door te specificeren dat er een *vector* berekening moet gebeuren laat de programmeur weten dat de berekeningen niet afhangen van de resultaten binnen de *vector* en zo is het niet nodig om *hazards* af te vangen bij elk onderdeel binnen een *vector*. Ook hoeven *loop branch hazards* niet meer afgevangen te worden omdat een *vector* instructie een gehele loop beschrijft.

- *Vector* architecturen en compilers maken het makkelijker efficiënte programma's te schrijven wanneer parallellisme op data niveau aanwezig is.
- Geheugen benadering van een *vector* instructie heeft een bekend patroon, wanneer de onderdelen naast elkaar in het geheugen liggen kunnen deze met slechts een keer het geheugen benaderen uitgelezen worden. Hierdoor wordt er veel *latency* bespaard door niet voor elk *word* apart op het geheugen te hoeven wachten.

C.S. Bras

6.4 Hardware Multithreading

Bij *hardware multithreading* kunnen meerdere *threads* gebruik maken van de functionele delen van een enkele *processor core* om zo de aanwezige hardware zo efficiënt mogelijk te gebruiken. Wanneer een *thread* stil staat, stapt de processor over naar het verwerken van een ander *thread*. Om dit delen mogelijk te maken moet de processor een kopie hebben van de staat van elke individuele *thread*. Ook moet er een manier zijn om snel tussen textitthreads te kunnen schakelen.

Twee veel gebruikte manieren van *hardware multithreading* zijn **fine-grained multithreading** en **coarse-grained multithreading**.

- ***Fine-grained multithreading*** schakelt tussen *threads* na elke instructie. Dit resulteert in een doorschoten uitvoer van de *threads*, dit wordt wel zo gedaan dat vastgelopen *threads* worden overgeslagen. Om deze manier nuttig te maken moet de processor elke klok cyclus van *thread* kunnen wisselen. Het nadeel van deze manier is dat individuele *threads* worden uitgesmeerd en dus langer de tijd nodig hebben om uitgevoerd te worden.
- ***Coarse-grained multithreading*** is een alternatief op *fine-grained multithreading* waarbij er alleen van *threads* gewisseld wordt bij langdurige wachttijden. Hierdoor is er geen noodzaak om *threads* wisselen binnen een klok cyclus te laten vallen. Een nadeel van deze manier is dat elke keer dat er een langdurige *stall* voorkomt, de *pipeline* geflushed moet worden en weer gevuld moet worden met instructies van een ander *thread* voordat deze uitgevoerd worden, daarom is dit alleen bruikbaar bij langdurige *stalls* waarbij vergeleken de tijd voor het vullen van de *pipeline* verwaarloosbaar klein is.

Naast deze vormen is er nog een variant, ***Simultaneous multithreading (SMT)***. Dit maakt gebruik van *thread-level* parallellisme.

C.S. Bras

6.5 Mutlicore en Andere Gedeeld Geheugen Multiprocessors

Hoewel *hardware multithreading* voor een verbetering in performance hebben gezorgd tegen lage kosten, blijft het lastig om oude programma's om te zetten zodat deze efficiënt draaien op deze hardware. Een oplossing hiervoor is het hebben van een enkele fysieke geheugen locatie die benaderd kan worden door alle processors. Hiermee kunnen alle variabelen van een programma bereikt worden door elke processor op elk willekeurig moment. Deze vorm van een multiprocessor wordt ook wel ***shared memory multiprocessor (SMP)*** genoemd.

Single address space multiprocessors komen in twee stijlen:

1. ***Uniform Memory Access (UMA)*** hierin is de *latency* van elk *word* in het geheugen hetzelfde ongeacht welke processor het opvraagt.
2. ***Nonuniform Memory Access (NUMA)*** hierin ligt de *latency* aan welke processor een stuk geheugen aanroept.

Aangezien parallelle processors bewerkingen uitvoeren op dezelfde data moet er een mechanisme zijn die er voor zorgt dat twee processors niet tegelijk aan dezelfde data werken. Deze coördinatie wordt **synchronisatie** genoemd. Een manier om dit te doen is om er voor te zorgen dat een van de processors een gedeelde variabele op slot kan zetten zolang als dat deze een bewerking hier op uitvoert.

C.S. Bras

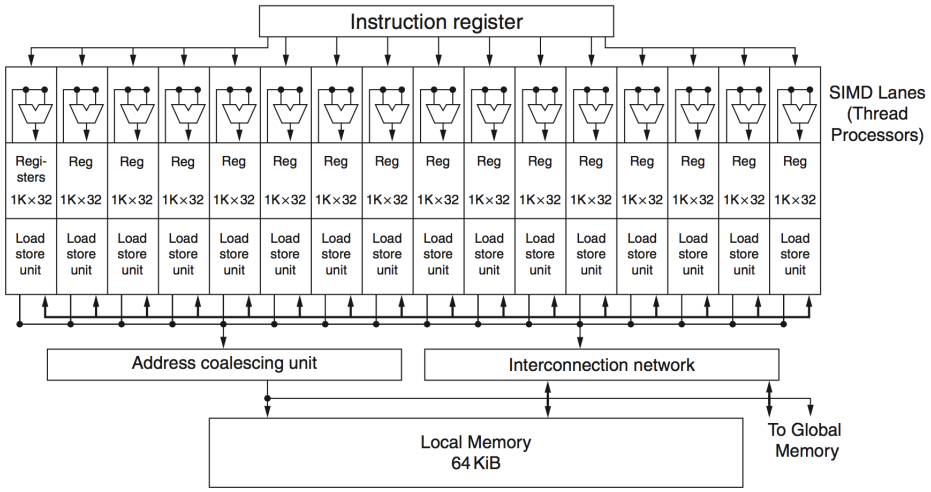
6.6 Introductie tot Grafische Processing Units

Graphics Processing Units (GPU's) verschillen in de volgende opzichten van standaard *CPU's*:

- Een *GPU* is een *accelerator* die als supplement wordt toegevoegd aan een *CPU* en hoeft daarom ook niet alle taken te kunnen doen die een *CPU* kan. Hierdoor kan de *GPU* al de resources focussen op beeldverwerking en de *CPU* de rest laten doen.
- Een *GPU* werkt met honderden megabytes tot gigabytes in plaats van honderden gigabytes tot terabytes. Een *GPU* gebruikt dan ook geen *multi level caching*. In plaats hiervan gebruiken ze *hardware multithreading* om *latency* van het geheugen te overkomen.
- *GPU* geheugen is gebouwd rondom hoge bandbreedte in plaats van lage *latency*.
- Doordat er gerekend wordt op veel *threads* om de geheugen bandbreedte hoog te houden heeft een *GPU* veel parallelle processors, veel meer dan een standaard *CPU*.

6.6.1 Introductie tot de NVIDIA GPU Architectuur

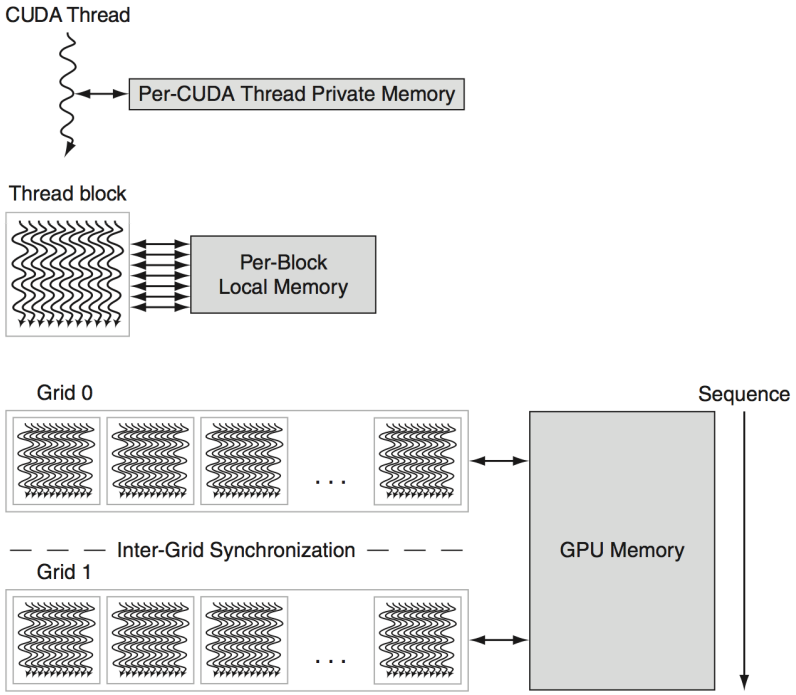
Net zoals met een *vector architectuur* werken *GPU's* alleen optimaal met data-level parallelle problemen. In tegenstelling echter tot de meeste *vector architecturen*, rekent een GPU ook op hardware *multithreading* binnen een enkele *multithreaded SIMD processor*. Aangezien NVIDIA meerdere producten aanbiedt met verschillende hoeveelheden *multithreaded SIMD processors* wordt *Thread Block Scheduler* hardware gebruikt om te regelen welke processor welk blok aan *threads* ontvangt, zie figuur 9 voor een schematische tekening van een *multithreaded SIMD processor*.



Figuur 8: Schematische tekening van het datapad van een *multithreaded SIMD processor*

6.6.2 NVIDIA GPU Geheugen Structuren

In figuur 10 wordt het geheugen structuur van een *NVIDIA GPU* schematisch weergegeven. Het geheugen op de chip dat lokaal is voor elke *multithreaded SIMD processor* wordt *Local Memory* genoemd. Dit geheugen wordt gedeeld door SIMD Lanes binnen een SIMD processor maar niet door meerdere SIMD processors. In plaats van een grote caches gebruikt een GPU kleinere *streaming caches* waarbij er gerekend wordt op veel *multithreaded SIMD* instructies om de *latency* van het *DRAM* te verbergen.



Figuur 9: GPU geheugen structuur

6.7 Clusters, Magazijn Grootte Computers, en andere Message-Passing Multiprocessors

C.S. Bras

Een alternatief op het delen van adres ruimte is dat elke processor zijn eigen fysieke adres ruimte heeft. Nu moeten deze processors echter wel uitdrukkelijk communiceren via ***message passing***, ook wel gezien als de naam van dit soort processors.

Clusters zijn een collectie computers die via een netwerk aan elkaar verbonden staan om zo samen een multiprocessor te worden. Elke computer draait zijn eigen besturingssysteem en eigen processor wat weer een multiprocessor kan zijn zoals hiervoor beschreven. Het grote voordeel van een eigen geheugens hebben ten opzichte van een *shared memory multiprocessor* is dat het gehele systeem niet stil gelegd hoeft te worden wanneer een individuele *node* vervangen moet worden, ook is het bij een cluster makkelijk om uit te breiden zonder het programma wat hier op draait stil te hoeven leggen.

Tegenwoordig hebben grote internet bedrijven niet meer genoeg aan normale clusters, daarom zijn er *Warehouse-Scale Computers (WSC)* die qua grootte een geheel magazijn innemen. Hoewel deze *WSC's* overeenkomsten hebben met servers, zijn er drie grote verschillen:

1. *Ruim, makkelijk parallellisme:* Een server ontwerper moet rekening houden of de toepassing van het systeem wel genoeg parallellisme zal hebben om een grote hoeveelheid parallelle hardware noodzaak te maken. Ook moet er rekening gehouden worden met de kosten van de communicatie hardware om gebruik te maken van dit parallellisme. Bij een *WSC* echter, worden vele stukken verschillende data aangeleverd die allemaal apart bewerkt moeten worden waarbij geen verdere communicatie tussen hardware nodig is en ook verder niet afhangen van elkaar.
2. *Operationele kosten:* Normaal gesproken ontwerpen server architecten een systeem voor maximale performance binnen een bepaald budget en wordt er alleen rekening gehouden met de

koelingscapaciteit van de behuizing zonder dat er rekening wordt gehouden met de kosten voor de benodigde energie van het draaien van de server. Bij WSC's ligt dit echter anders aangezien deze in veel langere tijd worden afgeschreven waardoor de kosten voor het operationeel houden veel meer oplopen dan bij traditionele servers.

3. *Grootte en de bijkomende problemen en voordelen hiervan:* Doordat voor een *WSC* veel servers nodig zijn kan dit in bulk met korting gekocht worden. Zo is er ondanks het feit dat er weinig *WSC's* zijn toch nog spraken van schaalvoordeel. Deze schaal brengt echter als problemen mee dat er rekening gehouden moet worden met het aantal falende systemen. Ook als zal er maar een klein percentage dagelijks falen, wanneer er honderdduizenden systemen zijn komt dit uiteindelijk toch neer op een relatief groot absoluut aantal.

C.S. Bras

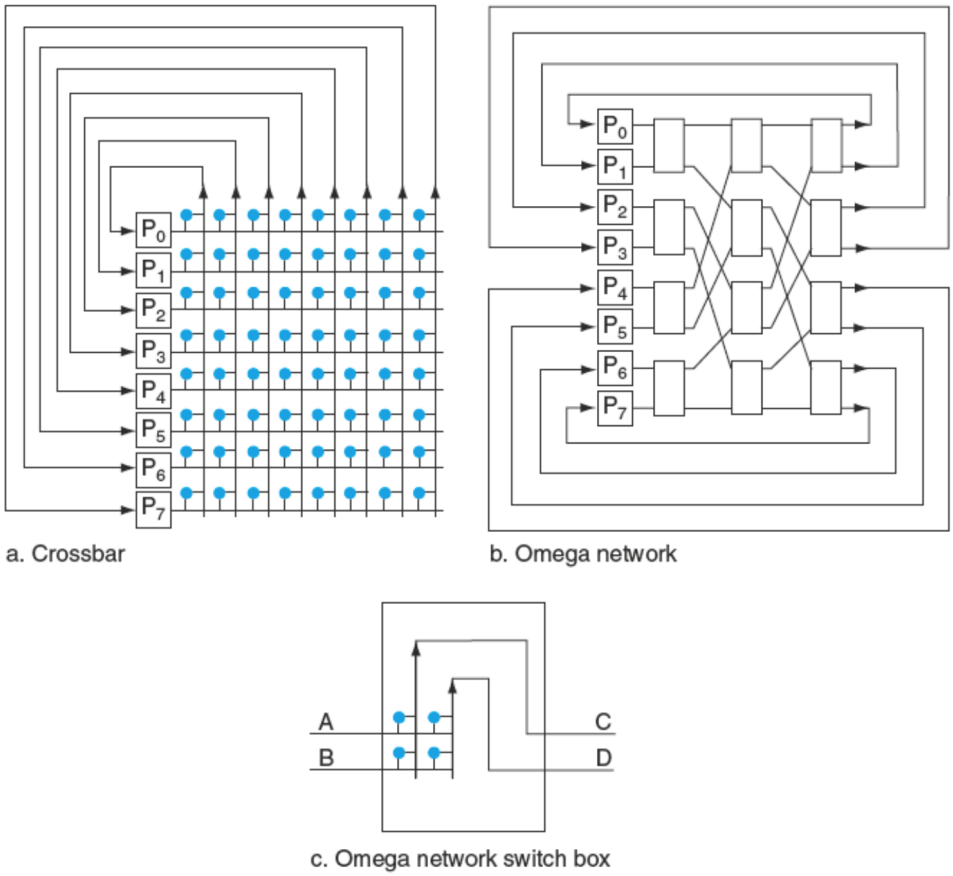
6.8 Introductie van Multiprocessor Netwerk Topologieën

Multiprocessor chips hebben netwerken op de chip nodig om met meerdere cores te kunnen communiceren, clusters hebben lokale netwerken nodig om tussen de verschillende servers te kunnen communiceren. Dit stuk kijkt naar de voor en nadelen van beide interconnectie topologieën. In het figuur hieronder is een schematische weergave te zien van een *ring* topologie, de processor knooppunten zijn zwart gekleurd en de netwerk switches worden aangegeven met blauwe cirkels. In tegenstelling tot een *bus*, kan een ring meerdere data uitwisselingen tegelijk afhandelen.

Figuur 10: Schematische weergave van een ring netwerk topologie met processor knooppunten als zwarte vierkanten en netwerk switches als blauwe cirkels

Doordat er veel verschillende soorten netwerktopologieën bestaan moeten er punten zijn waarop de performance kan worden vergeleken. Het eerste punt is ***totale netwerk bandbreedte***, dit is de bandbreedte van elke link vermenigvuldigd met het totaal aantal links. Dit is wat in optimale omstandigheden de hoogst haalbare bandbreedte is. Een ander punt waarop vergeleken kan worden is de ***tweeëndeling bandbreedte***, dit gaat uit van minder gunstige omstandigheden en wordt berekend door de machine in twee helften te verdelen om vervolgens de bandbreedte van de twee links die deze denkbeeldige lijn doorkruisen bij elkaar op te tellen.

Een netwerk met een kleine switch op elk knooppunt wordt een ***multistage netwerk*** genoemd. In *figuur 12* zijn enkele populaire vormen van *multistage netwerken* te zien.



Figuur 11: Populaire *multistage* netwerktopologieën voor acht knooppunten. De links zijn een richting, data komt links binnen en gaat rechts weer naar buiten.