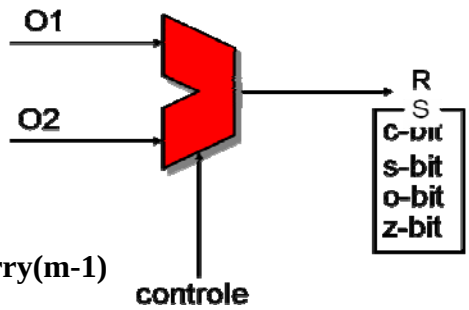
Aritmetische instructies

1/ Operatorgedeelte

ALU = circuit dat 2 operandi (O1 en O2) als input heeft, en een resultaat en statusbits als output. Statusbits bevatten toestandsinformatie over de uitgevoerde functie en worden opgeslagen in het toestandsregister van de CVE. (**c / o / s / z**)



Bij een optelling van 32 bit zal
- s=1 als R<31:31>=1
- z=1 als R=0
- c=1 als carry(31)=1

met **A<m:n>** het getal in bitpatroon A tussen de m^{de} en de n^{de} bit.
carry(m) duidt de overdracht aan bij **O1<m:m>+O2<m:m>+carry(m-1)**

2/ Overflow

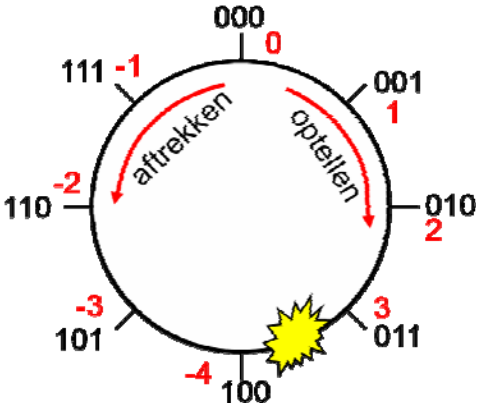
Wanneer? Als bij de optelling de carry naar de tekenbit (carry(30)) verschillend is van de carry uit de tekenbit(carry(31)), dus met andere woorden **o-bit = carry(31) XOR carry(30)**.

<u>Som van 2 + getallen</u>
Overflow: enkel carry(30), dus R<31:31> = 1 (resultaat is dus negatief! → tekenbit)
Geen overflow wanneer ze kan worden voorgesteld binnen de getalrepresentatie

<u>Som van 2 – getallen</u>
Overflow: enkel carry(31), dus R<31:31> = 0 (resultaat is dus positief! → tekenbit)
Geen overflow wanneer ze kan worden voorgesteld binnen de getalrepresentatie

<u>Som van pos en neg getal</u>
NOOIT OVERFLOW! Als pos getal de grootste absolute waarde heeft, dan is er zowel carry(30) als carry(31), zoniet is er geen van beide.

3/ Getallencirkel



Bij modulorekenen werkt men met deze getallencirkel. Alle voor te stellen getallen staan op de cirkel. Modulorekenen is een manier om overflow op te vangen.

4/ Saturatierekenen

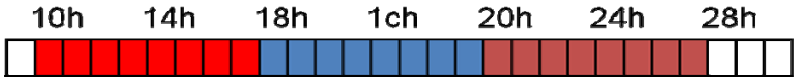
Overflow kun je ook opvangen door saturatierekenen. Het resultaat welke niet kan voorgesteld worden, zal worden vervangen door het grootste of kleinste getal dat men wel kan voorstellen. Dit kan nuttig zijn bij beeldverwerking. Wanneer we de afbeelding helderder willen maken, zullen zéér heldere pixelwaarden (bvb max 255) hoger komen te liggen dan afgesproken. We plaffoneren ze op de maximale waarde 255...

5/ Optelling & aftrekking
n bit + n bit = maximaal (n+1) bit
add doel, bron → doel = doel + bron

n bit – n bit = maximaal (n+1) bit
sub doel, bron → doel = doel – bron

Een voorbeeld wordt gegeven door een 64 bit-optelling op een 32 bit processor.

```
mov     eax,[10h]
mov     ebx,[14h]
add     eax,[18h]
adc     ebx,[1ch]
mov     [20h],eax
mov     [24h],ebx
```



De adc zorgt ervoor dat de twee meest beduidende (**little endian**) 32-bit helften worden opgeteld, met carry genomen uit de som (add) van de 2 minst beduidende 32-bit helften. Het eerste getal ligt van 10h → 18h, het tweede van 18h → 20h, en het resultaat wordt opgeslagen van 20h→ 28h.

6/ Vermenigvuldiging & deling
n bit * n bit = maximaal (2n) bit
mul bron → **edx:eax = bron * eax** (unsigned)
imul bron → **edx:eax = bron * eax** (signed)
imul d,bron → **d = d*bron**
imul d,bron1,bron2 → **d = bron1*bron2**

n bit / n bit = maximaal n bit
div bron → **eax = edx:eax / bron**
edx = edx:eax % bron (dus de rest)

Bij mul komen de 32 meest beduidende bits van het 64 bit resultaat dus in EDX, de minst beduidende in EAX. (EDX:EAX = registerpaar)
Bij div komt het afgekapte resultaat van de deling in EAX terecht, en de rest (**samen met het teken van het deeltal**) in EDX. (-4/3 EAX=-1, EDX=-1 -4/-3 EAX=1, EDX=-1)

Product in helften
Andere architecturen zoals de alpha-architectuur hebben ervoor geopteerd om het product in twee stukken te laten berekenen. De laagste 64 bit voor de courante gevallen, en de hoogste 64 bit voor die omstandigheden waarin dit vereist zou zijn.

mulq a,b,c → reg[c]:=(reg[a]*reg[b])<63:0>
umulh a,b,c → reg[c]:=(reg[a]*reg[b])<127:64>

Deling door een constante
Kan worden gespecificeerd door een vermenigvuldiging: X/15 = X * (1/15)

7/ Vergelijkingen

cmp d,s (vergelijken van de waarden <, = en >. Doet eigenlijk hetzelfde als sub, maar gooit het resultaat weg, en behoudt de statusbits → vergelijken van de waarde van 2 bitpatronen)

test d,s (testen van bits aan/uit. Doet eigenlijk hetzelfde als sub, maar gooit het resultaat weg, en behoudt de statusbits)

We achterhalen eerst wanneer a<b. Stel dat we **cmp a,b** doen. Als a<b, dan zal er moeten geleend worden uit de hoogste rang, waarbij dus **c==1**.

z		z == 1	zero	
c		c == 1	carry	
o		o == 1	overflow	
p		p == 1	parity	
s		s == 1	sign	
nz		z == 0	no zero	
nc		c == 0	no carry	
no		o == 0	no overflow	
np		p == 0	no parity	
ns		s == 0	no sign	
g	nle	((s xor o) or z) == 0	greater	2-complement
ge	nl	(s xor o) == 0	greater or equal	
l	nge	(s xor o) == 1	less	
le	ng	((s xor o) or z) == 1	less or equal	
e		z == 1	equal	
a	nbe	(c or z) == 0	above	
ae	nb	c == 0	above or equal	
b	nae	c == 1	below	Binair
be	na	(c or z) == 1	below or equal	

add d,s	d = d + s
adc d,s	d = d + s + c
sub d,s	d = d - s
sbb d,s	d = d - s - c
mul s	vermenigvuldiging (unsigned)
imul s	vermenigvuldiging (signed)
div s	deling (unsigned)
idiv s	deling (signed)
neg d	d = -d
inc d	d = d + 1
dec d	d = d - 1

Logische instructies

1/ Bitsgewijze logische operaties

```
and d,s
or  d,s
xor d,s
not d
```

d = d ‘and’ s
d = d ‘or’ s
d = d ‘xor’ s
d = ‘not’ d

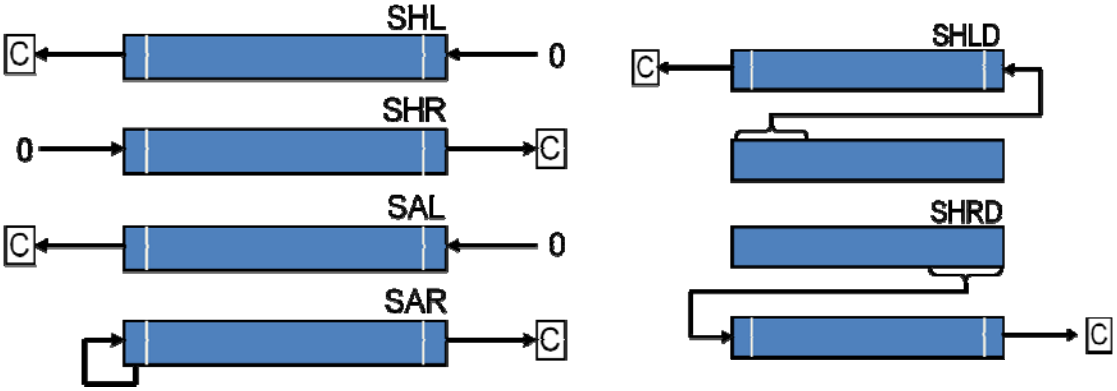
2/ Verschuivingen & rotaties

shl d,n	d = d << n	Binair
shr d,n	d = d >> n	
sal d,n	aritmetisch d = d << n	2-complement
sar d,n	aritmetisch d = d >> n	
shld d,s,n	d = (d:s << n)<63:32>	
shrd d,s,n	d = (d:s >> n)<63:32>	
rol d,n	bitrotatie naar links	
ror d,n	bitrotatie naar rechts	
rcl d,n	uitgebreide bitrotatie naar links	
rcr d,n	uitgebreide bitrotatie naar rechts	

ca4-25

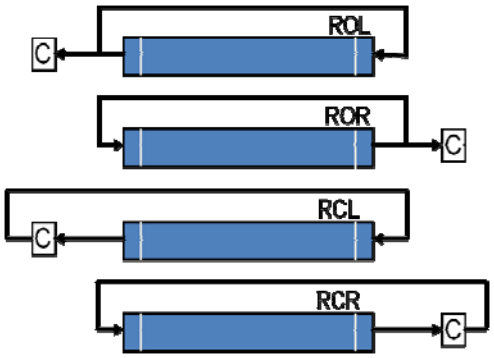
SHL en SHR zal ‘n’ 0-bits in het bitpatroon ‘d’ schuiven. **Logische schuifoperaties**
Bij een schuifoperatie naar links, worden er van rechts nullen ingevoegd. De laatste bit die er dan links uitvalt, wordt bewaard in de carrybit.
Bij logische (en aritmetische) schuifoperaties betekent een verschuiving naar links, vermenigvuldigen met 2, en een verschuiving naar rechts een **gehele** deling door 2.

SAL = SHL, SAR wijkt af doordat de ingevoegde bits dezelfde waarde hebben als de tekenbit van de verschoven waarde → kan dienen voor gehele deling door 2!! **Aritmetische schuifoperaties.** Bij het schuiven naar rechts van -1, levert dit telkens opnieuw -1 op.



SHLD en SHRD voeren logische shiftoperaties uit op een registerpaar. De in te schuiven bits zijn uit een ander register afkomstig.

ROL en ROR zijn bitrotaties. De weggeschoven bits komen er dus aan de andere kant weer bij. Bij RCL en RCR wordt samen met de carry-bit een 33bit rotatie gecreëerd.



Verschuivingen worden meestal toegepast om een eenvoudig product (in machten van 2) te berekenen, omdat deze minder tijd in beslag nemen dan het eigenlijke product.

3/ Bit test instructies

bt d, o	c = d<o:o>
bts d, o	c = d<o:o>; d<o:o> = 1
btr d, o	c = d<o:o>; d<o:o> = 0
btc d, o	c = d<o:o>; d<o:o> = ~~d<o:o>~~

Selecteer een particulier bit uit de d-operand. De bits worden geteld van rechts naar links, te beginnen bij 0 als d een register is. (aflopende bitnummering dus!!)
bt = bit test
bts = bit test and set
btr = bit test and reset
btc = bit test and complement

4/ Bit scan instructies

bsf d, s	d = minst significante 1-bit
bsr d, s	d = meest significante 1-bit

Op zoek gaan naar het eerste niet nul bit in de operand s, het resultaat komt in d. Indien er geen 1 bit wordt gevonden: z==0 en d onbepaald. (aflopende bitnummering, f = forward, r = reverse)

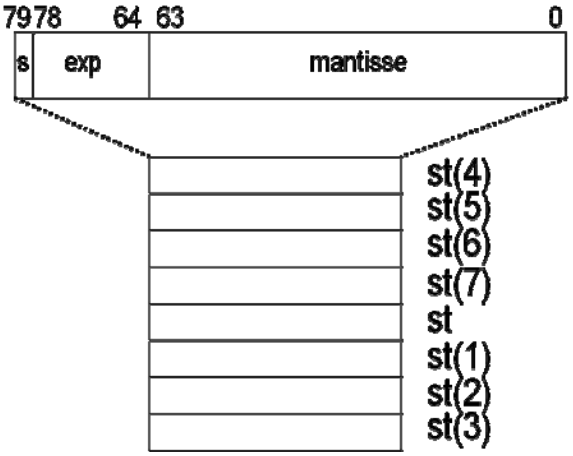
5/ Set byte on condition

set<cc> d	d = conditiecode
setge al	reg[al] = ge ? 1 : 0
seto ah	reg[ah] = o ? 1 : 0
setno al dec al	reg[al] = o ? 0ffh : 00h

Kopieert een particuliere conditiecode naar d die op basis hiervan de waarde 0 of 1 krijgt. De laatste 2 instructies zorgen er samen voor dat men nu niet de waarde 0 of 1 in d wil steken, maar wel 0 of 0ffh door de omgekeerde conditie te nemen en het resultaat met 1 te verminderen.

Vlottende komma instructies

1/ Vlottende komma registerverzameling



St = top van de stapel. St(1) zit 1 element onder de top van de stapel, enz. St(7) zit het diepst in de stapel.
Hier kan dus worden gewerkt met dubbele extended precisie vlottende kommagetallen van 80 bit, maar ook met floating points van enkelvoudie, dubbele precisie, integer woorden, dubbel- en quadwoorden, en BCD getallen van 18 nibbles en een tekenbyte (80bit).

2/ Adresseermodes

Zuivere stapeladressering

fadd	st(1) = st+st(1); pop

Registeradressering

fadd st(i)	st = st+st(i)
fadd st(i),st	st(i) = st+st(i)
fadd st,st(i)	st = st(i)+st
faddp st(i),st	st(i) = st+st(i); pop

Geheugenadressering

fadd <ae>	st = st+mem[ae]

Worden gebruikt om gegevens op de vlottende komma stapel te gebruiken. Met st(n) bedoelt men hier wel **reg[st(n)]**

3/ Vlottende kommabewerkingen

fadd	FP optelling
fsub	verschil
fdiv	deling
fprem	rest na deling
fmul	vermenigvuldiging
fabs	absolute waarde st = abs(st)
fchs	negatie st = - st
fcomi	FP compare and set eflags

4/ Wiskundige functies

fsqrt	vierkantswortel
fsin	sinus
fcos	cosinus
fptan	tangens
fpatan	arcus tangens
fyl2x	logaritme st(1) = st(1)*$\log_2$(st); pop
fyl2xp1	logaritme st(1) = st(1)*$\log_2$(st+1); pop
f2xm1	exponent st = 2^{st}-1

5/ Load-store operaties

fbld	Push BCD-getal (ld=load)
fbstp	Store BCD and pop
fild	Push integer 16/32/64 bit
fist	Store integer 16/32/64 bit
fld	Push floating point value 32/64/80 bit
fst	Store floating point value 32/64/80 bit

6/ Load constant

fld1	push 1.0
fldl2t	push $\log_2 10$
fldl2e	push $\log_2 e$
fldpi	push π
fldlg2	push $\log_{10} 2$
fldln2	push $\log_e 2$
fldz	push +0.0

MMX instructies (Pentium)

1/ Aanleiding
In de jaren 90 was er een grote opkomst van multimedia die werd gekenmerkt door een groot aantal onafhankelijke **kleine** data-elementen. De woordbreedte van de processor echter, nam toe tot 64bit. Het gevolg hiervan was dat er per 64bits maar 8 of 16 bits effectief werden gebruikt. De oplossing is verschillende data-elementen in 1 64 bit register plaatsen.

2/ MMX registerverzameling
8 registers van 64 bit (mm0 – mm7), die overlappen met de laagste 64 bit van de 80 bit brede vlottende komma registers om ervoor te zorgen dat de toestand van de processor niet toeneemt.

3/ Instructies
57 nieuwe instructies die vooral de bedoeling hebben om gegevens in geschikte vorm in de registers te krijgen, en dan simultaan een operatie uit te voeren op de verschillende registeronderdelen. Ze ondersteunen dus **subwoordparallellisme**.
De 'p' voor de instructies staat voor packed.

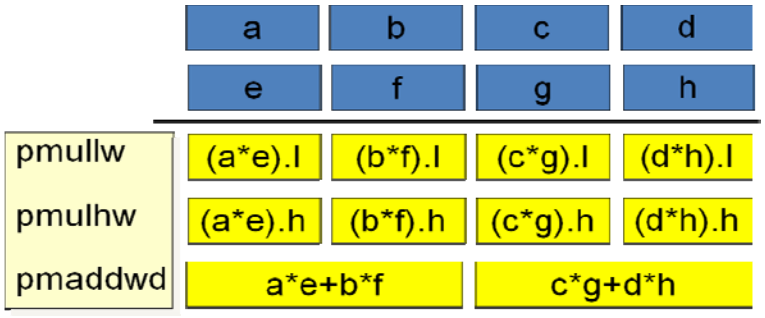
Optellen en aftrekken

padd[b/w/d/q] mm0,mm1 = gegevens byte / woord / dubbelwoord / quadwoord per ... optellen. In het geval van paddb wordt er modulorekenen per byte toegepast, en er is in dit geval geen overdracht op de bytegrenzen.
psub[b/w/d/q] mm0,mm1

padds[b/w] mm0,mm1 past saturatierekenen toe op mm0 en mm1 (in tegenstelling tot modulorekenen) en werkt met **2-complementgetallen**.
psubs[b/w] mm0,mm1

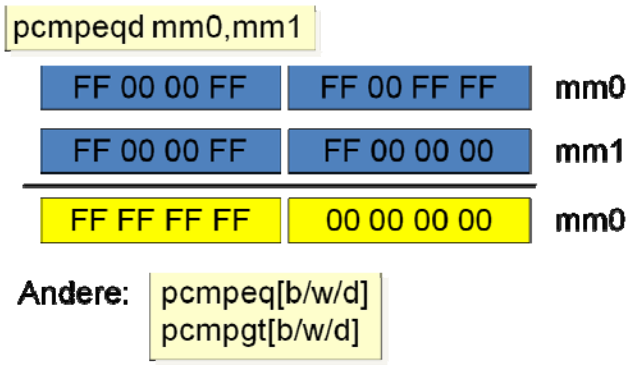
paddus[b/w] mm0,mm1 zelfde als hierboven, maar werkt op **binaire** getallen.(u = unsigned)
psubus[b/w] mm0,mm1

Vermenigvuldigen



pmull levert de laagste 16 bits van de producten, pmulh de hoogste 16 bits. pmaddwd voegt beide producten samen.

Vergelijken



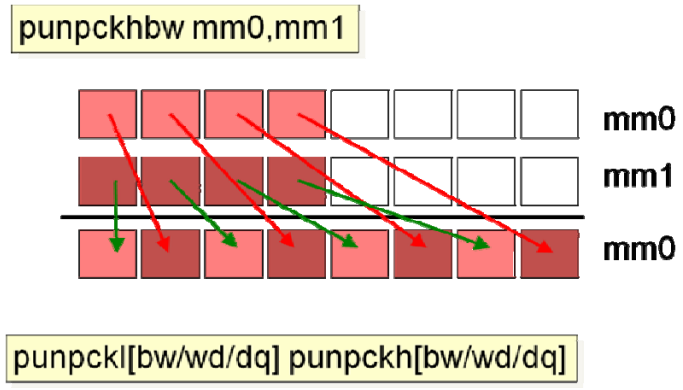
Omdat de mmx operaties niet zomaar gebruik kunnen maken van het EFLAGS register om conditiecodes bij te houden (door parallelle vergelijkingen is er 1 conditiebit per vergelijking nodig!), wordt er gebruik gemaakt van een masker. In bovenstaand voorbeeld zijn linker dubbelwoorden gelijk, dus 32 1tjes.

Logische instructies

psra[w/d]
psll[w/d/q]
psrl[w/d/q]
pand
pandn
por
pxor

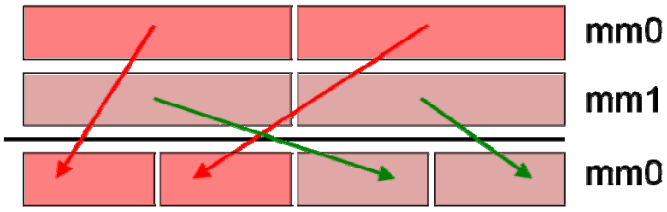
Deze instructies werden ingevoerd omdat de klassieke logische instructies inwerken op registers voor algemeen gebruik (eax,ebx,...). Dus zouden de waarden vanuit de mmx registeres eerst moeten worden getransfereerd naar de registers voor algemeen gebruik, om te bewerken, en daarna terug naar de mmx registers.

(Un)pack instructies



Ingevoegd om operaties op subwoorden te doen. Deze operaties zijn alleen bruikbaar in de gevallen waar het mogelijk is de verschillende subwoorden op de goeie plaatsen in het register te krijgen. Hiervoor heeft men pack en unpack instructies uitgevonden. De ‘b’ in de bovenste instructie staat voor “per byte”, en de ‘w’: zet de hoogste bytes van mm0 en mm1 op woordafstand van elkaar.

packssdw mm0,mm1



packss[wb/dw] packuswb

Pack instructies (zoals hierboven) reduceren dubbelwoorden naar woorden. Hierboven wordt dit saturerend gedaan op 2 complement getallen (vandaar de dubbele 's': signed).

Einde
emms : zorgt ervoor dat de st registers opnieuw goed worden geïnitialiseerd.

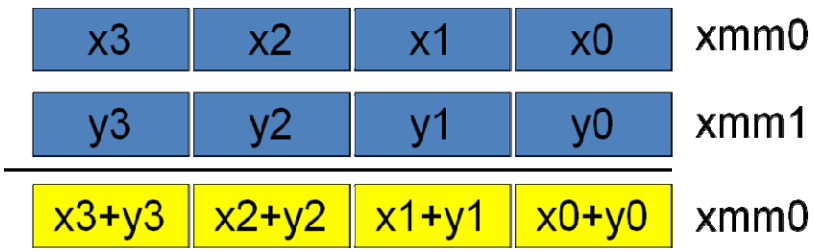
SSE/SSE2 instructies

1/ SSE extensies
8 extra XMM registers van 128 bits lang. Ook subwoord parallellisme met vlottende kommagetallen + 70 nieuwe instructies.

2/ Operaties

Packed SSE

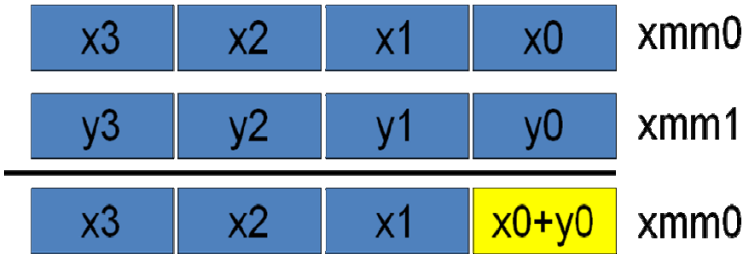
addps xmm0, xmm1



Telt 4 enkelvoudige precisie vlottende kommagetallen bij elkaar op.

Scalaire SSE

addss xmm0, xmm1

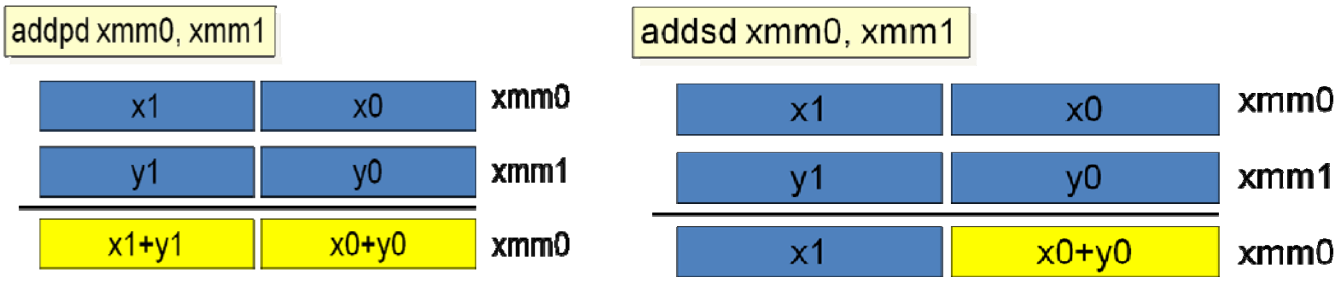


Telt 2 enkelvoudige precisie vlottende kommagetallen bij elkaar op, rest blijft onveranderd.

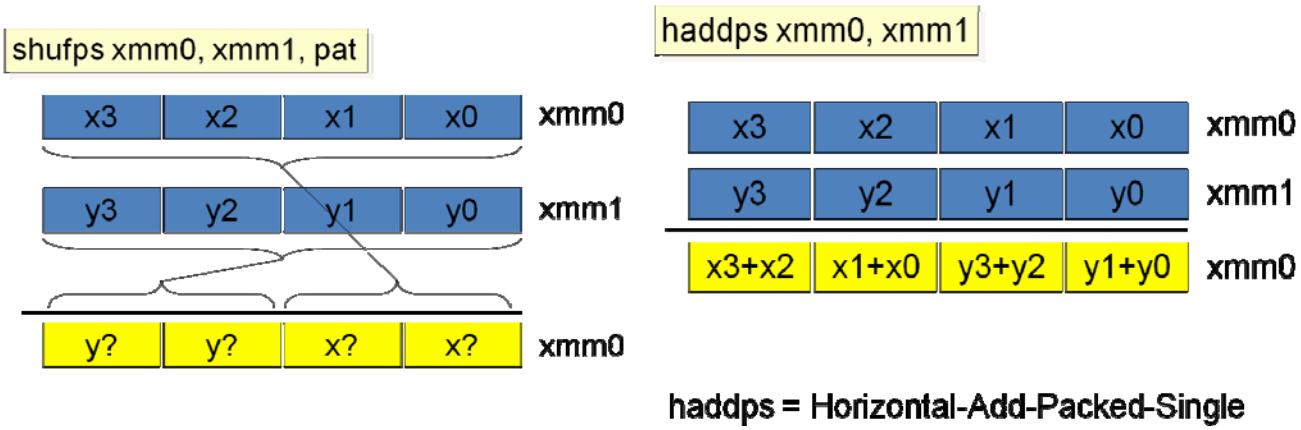
3/ SSE2 extensies
Ook subwoord parallellisme met vlottende kommagetallen **in dubbele precisie** en integers. + 144 nieuwe instructies.

4/ Operaties

Packed SSE2 Scalaire SSE2



Shuffle SSE3 extensies



Shuffle laat toe om willekeurig de vlottende komma getallen te kopiëren. Twee uit xmm0, en twee uit xmm1. De selectie wordt bepaald door het 8bit patroon **pat**. (2 bit per waarde)
In 2004 werden de SSE3 uitbreidingen toegevoegd. Bijzonder aan deze uitbreiding is dat ze naast operaties op verticale operandi (uit 2 registers), ook operaties op horizontale operandi (hetzelfde register) aanbiedt.

Varia

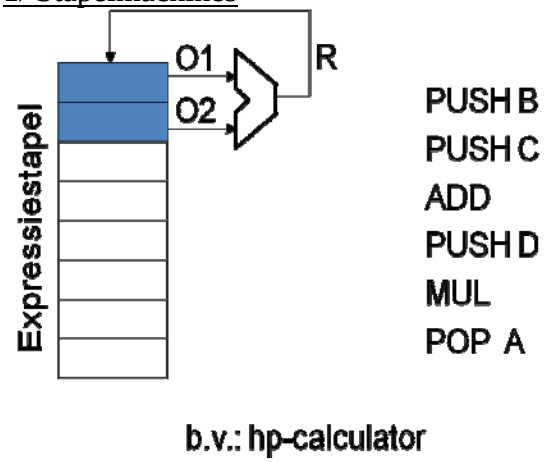
nop : doe niets, plaats reserveren in een instructiestroom (ong. gelijk aan mov eax,eax)
xadd d,s : t = d+s; s=d; d=t;
cmpxchg d,s : if (reg [eax] == d)
d=s;
else reg [eax] = d;
lock prefix : deze byte plaatst men net voor de instructie. De instructie die erop volgt wordt nu atomair uitgevoerd → read-modify-write cyclus.

stc / clc → c = 1 / 0
std / cld → d = 1 / 0
sti / cli → i = 1 / 0

Machinemodellen

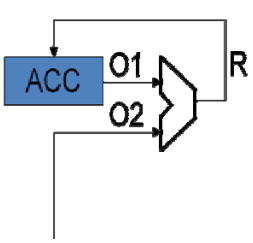
Men onderscheidt drie soorten machinetypes op basis van aantal en toegankelijkheid van de registers in de CVE.

1/ Stapelmachines



Bevat een aantal registers die georganiseerd worden als een stapel. Het plaatsen van een nieuw element op de top (**push**) zorgt ervoor dat alle andere aanwezige elementen een plaats naar beneden worden geschoven. Het omgekeerde gebeurt met **pop**. Men hoeft nooit een registernaam te vermelden, want de bovenste twee elementen worden bij een bewerking van de top genomen, en het resultaat wordt er terug op gepushed. Hier staat het voorbeeld A = (B+C)*D uitgewerkt.
Een typisch voorbeeld is de HP48 zakrekenmachine, en de JVM.

2/ Accumulatormachines

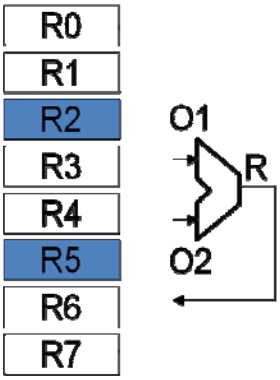



```
A := (B+C)*D

LOAD B
ADD  C
MUL  D
STORE A
```

De bewerkingen maken steeds gebruik van 1 register (**accumulator**) voor een van de operandi O1. Voorts zijn de overige registers individueel adresseerbaar. Het resultaat komt steeds in de accumulator terecht. De tweede operand moet wel expliciet worden vermeld. Net zoals bij stapelmachines zijn de instructies hier zeer kort, maar er is een geringe soepelheid mbt de bewerkingen, want voor elke bewerking moet de accumulator gepast ingesteld worden.
Deze machines hebben over het algemeen niet veel registers.

3/ Registermachines




```
A := (B+C)*D

MOV R1,B
MOV R2,C
ADD R3,R2,R1
MOV R1,D
MUL R2,R1,R3
MOV A,R2
```

Vrije toegang tot de registers, en geen impliciete O1 en O2! Bij de **twee-adresmachines** wordt het resultaat weggeschreven naar één van de operandlocaties, maar bij de **drie-adresmachines** kan men een vrij adres opgeven waar het resultaat moet terechtkomen. Als men voor de operandi enkel registerlocaties kan opgeven, spreekt men van **load/store of register-naar-registermachines**. Operandi kunnen ook uit geheugenlocaties komen, en hierbij maakt men een onderscheid als het resultaat meteen naar het geheugen verwezen kan worden (**geheugen-naar-geheugen**) of enkel in een register kan weggeschreven worden (**geheugen-naar-register**).
Hierboven : drie-adres register-naar-registermachine. Zie slide 81/82 voor andere vormen!

Samenvattend:
Men zou denken dat en geheugen-naar-geheugen machine leidt tot het elegantste, kortste en snelste programma, maar in de praktijk is dit meestal niet zo.

- de bitpatronen die A, B, C en D voorstellen zijn langer dan de registernamen
- toegang tot een register gebeurt veel sneller dan toegang tot een geheugenplaats. Als een gegeven veel gebruikt wordt, kan dit al rap leiden tot grote verschillen.
- geheugen-naar-geheugen machines zijn complexer, en werken dus trager!

Controletransferinstructies

= Instructies die verandering brengen in de zuiver sequentiële uitvoering van de instructies. Ze doen dit door de instructiewijzer een andere waarde te geven. Ze kunnen met andere woorden worden gezien als mov-instructies naar **eip**.

Sprongen

1/ Onvoorwaardelijke sprongen

```
jmp   adres
```

reg[eip] = adres

2/ Voorwaardelijke sprongen

instructie	sprong
jz	jump if zero
jc	jump if carry
jo	jump if overflow
jp	jump if parity
js	jump if sign
jnz	jump if not zero
jnc	jump if not carry
jno	jump if not overflow
jnp	jump if not parity
jns	jump if not sign

instructie		sprong	
jg	jnle	jump if greater	2-complement
jge	jnl	jump if greater or equal	2-complement
jl	jnge	jump if less	2-complement
jle	jng	jump if less or equal	2-complement
je		jump if equal	
ja	jnbe	jump if above	binair
jae	jnb	jump if above or equal	binair
jb	jnae	jump if below	binair
jbe	jna	jump if below or equal	binair

Springt slechts indien aan een gestelde voorwaarde is voldaan. (bijvoorbeeld een **if** instructie)
We kunnen verloop van een programma voorstellen door een **controleverloopgraaf** met doorvalpaden al of niet. Een controleverloopgraaf is een aaneenschakeling van een aantal basisblokken.
De linkse spronginstructies testen de bits in het EFLAGS register, terwijl de rechtse een vergelijking maakt tussen twee getallen. De vergelijking gebeurt, zoals eerder gezegd, door de twee getallen van elkaar af te trekken zodat de ALU de toestandbits zet.

3/ Berekende sprongen

Een **statische sprong** springt naar een vast adres, terwijl een **berekende sprong** springt naar een adres welke gegeven wordt door een register/geheugeninhoud. (voorbeeld van gebruik bij **switch**)

```
mov ebx,100
jmp  ebx
```

reg[eip] = reg[ebx]

Switch 1
Geneste if-lussen! Als het aantal gevallen echter groot is, dan neemt de gemiddelde tijd nodig om een geval te vinden lineair toe met het aantal gevallen in de switch instructie. Een oplossing is dan om in een binaire boom te zoeken → logaritmisch.

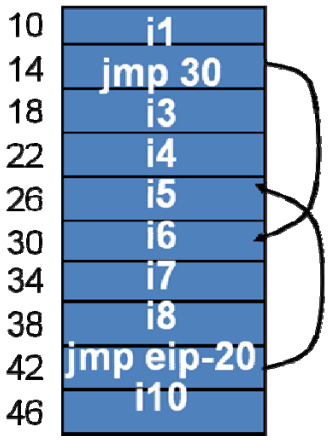
Switch 2
Berekende sprong in combinatie met een adrestabel.
Bvb mov eax, [tabel+ebx*4]
Eender welk geval kan nu in een constante tijd worden berekend. De extra kost bestaat uit de plaats nodig om de adressen in op te slaan, de controle op de geldige index, en de kost om het adres uit de tabel te halen.

```
tabel:  .long   $22
        .long   $25
        .long   $23
        .long   $24
```

4/ Absoluut versus relatief adres

- Absoluut
  - spring naar adres n
- Relatief
  - spring n bytes verder/terug

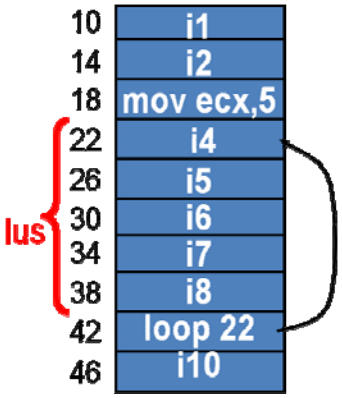
reg[eip] = reg[eip] + n



Relatieve sprongadressen zijn beter, omdat de sprong zo onafhankelijk wordt van wijzigingen elders in de code. (=**positie-onafhankelijke code**). Als men echter instructies invoegt tussen spronginstructie en doeladres, moet men de sprongafstand corrigeren!

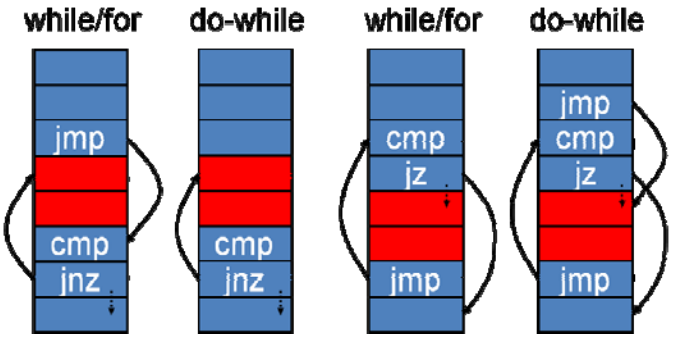
Lussen

1/ Loop
In de praktijk wordt de instructie **loop adres** niet meer gebruikt. Deze instructie decrementeert ecx en springt naar het opgegeven adres indien ecx != 0. Het gebruik ervan onderstelt dat ecx vooraf goed werd ingesteld.



2/ Voorwaardelijke sprong = geprogrammeerde lus
Men kan zelf de loopinstructie nabootsen door twee commando's:
sub ecx, 1
jnz xx

3/ Lusimplementaties



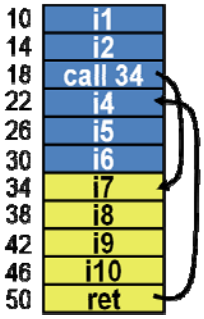
Indien het aantal iteraties vast is, en tijdens compileren gekend, dan spreekt men van een **manifeste lus**.

Procedureoproep & terugkeer

1/ Functie oproep

```
call adres
    reg[esp] = reg[esp]-4
    mem[reg[esp]] = reg[eip]
    reg[eip] = adres

ret
    reg[eip] = mem[reg[esp]];
    reg[esp] = reg[esp] + 4
```



Call en ret slaan/halen het returnadres op/van de stack op/af!

In plaats van het returnadres op te slaan op de stack, kan het ook opgeslagen worden in een register. Dit register wordt dan een **linkregister** genoemd. Dit is efficiënter, maar het werkt eigenlijk enkel maar goed in **bladroutines** (functies die geen andere functies meer oproepen).

```
call r,adres        bal  r, adres
    reg[r] = reg[eip]
    reg[eip] = adres

jmp r
    reg[eip] = reg[r]
```

Voorbeeld: vijfvoud. Zie slides 24-32 voor de code

2/ Parameterdoorgave via de stapel

```
vijfvoud:
    mov eax,[esp+4]
    cmp eax,0
    jg    positief
    xor   eax, eax
    ret
positief:
    mov ebx, 5
    imul ebx
    ret

main:
    push 6
    call  vijfvoud
    add  esp,4
    mov g, eax
```

De add esp,4 zorgt ervoor dat de plaats die ingenomen werd op de stapel door "6" weer wordt vrijgegeven. In sommige progammeertalen laat men echter de argumenten (hier de waarde 6) opruimen door de functie zelf door de **ret** instructie. Dit is echter alleen mogelijk in talen waarbij de functie het aantal argumenten kent.

```
vijfvoud:
    mov eax,[esp+4]
    cmp eax,0
    jg    positief
    xor   eax, eax
    ret   4
positief:
    mov ebx, 5
    imul ebx
    ret   4

main:
    push 6
    call  vijfvoud
    add  esp,4
    mov g, eax
```

3/ Lokale veranderlijken (in subroutines)

Worden meestal op de stapel bijgehouden door esp eentje naar boven te verschuiven en dan de temporele waarde op die plaats op te slaan... Bvb **mov [esp],eax**

4/ Uitgewerkt voorbeeld: maximum van 4 getallen

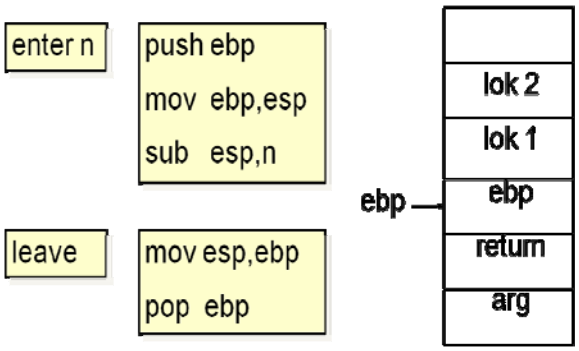
Hier wordt gewerkt met **stack frame pointers** ebp. Het is de bedoeling dat ebp nu als stack pointer wordt gebruikt voor een nieuwe functie. Dit wordt goedgezet door de volgende instructies:

push ebp
mov ebp,esp

Een stack frame breek je (natuurlijk) terug af door

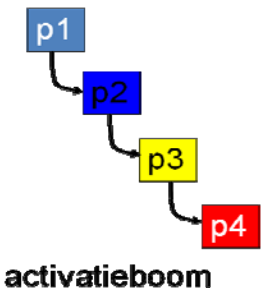
mov esp,ebp
pop ebp.

Er bestaan echter instructies die hetzelfde doen:



De "n" zal een aantal plaatsen voorzien voor lokale veranderlijken. Leave zal de esp naar de vorige stack frame pointer doen wijzen.

De stack frame pointers zullen in feiten alle stack frames aan elkaar linken (=**dynamische links**). De bijhorende stack wordt ook wel nog **call stack** genoemd.
Een **activatieboom** geeft een grafische weergave van hoe de verschillende functies elkaar hebben aangeroepen.



activatieboom

5/ Oproepconventies
Tussen oproepende code en subroutine code moeten duidelijke afspraken bestaan over de interface van beide stukken code: **oproepconventies**, welke bepalen:

- hoe de argumenten worden doorgegeven: via de stapel (in welke volgorde) of via registers (welke registers?)
- welk register is stapelwijzer?
- of registers worden bewaard door oproeper of oproepeling
- hoe het terugkeeradres wordt doorgegeven (stapel of register)

6/ Optimalisatie
Kopiepropagatie
Wanneer een waarde wordt berekend in register r1, om dan later gekopieerd te worden in r2, kan ze evengoed onmiddellijk in r2 worden berekend.

Dode waarden
Waarden die niets doen

Idempotente code
Een instructie die een resultaat berekent welke reeds bestond. Bijvoorbeeld push ebp en pop ebp, terwijl er met ebp niets wordt gedaan.

Sprong naar doorvalpad
Soms kan na optimalisatie worden gesprongen naar een doorvalpad. Dit kan zonder meer weggelaten worden.

Functiesubstitutie of inlining
Het vervangen van een functieoproep door de functiedefinitie zelf. Dus in plaats van **call functie**, zet je op deze plaats gewoon de instructies die deze functie bevat.

Onderbrekingen

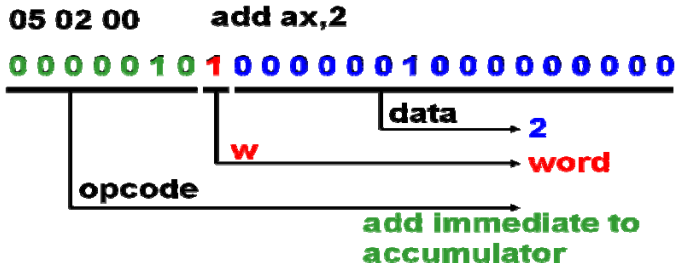
Onderbrekingen zijn eigenlijk een soort subroutines. Ze kunnen worden aangeroepen door de programmeur, maar ook door externe onderbrekingen door randapparatuur of CVE om een bepaalde situatie op te lossen. Het doeladres wordt via een **geheugenindirectie** verkregen. Aan elke onderbreking wordt een nummer toegekend, en het doeladres wordt gevonden in de **vectortabel** waarbij het nummer als index wordt gebruikt.

Stel dat tijdens een instructie een onderbreking wordt gegenereerd (int 3). Dit kan b.v. een onderbreking zijn die verband houdt met de instructie (divide by zero) of extern is (onderbreking om een I/O signaal, zoals muisbeweging, te behandelen). Als gevolg daarvan wordt gesprongen naar de subroutine waarnaar cel 3 in de vectortabel wijst. Na afloop van de onderbreking keert de uitvoering terug naar de volgende instructie en wordt de normale uitvoering voortgezet. Je kan ook expliciet het nummer van de onderbreking oproepen.
Als je nu zelf wil bepalen wat er moet gebeuren bij een welbepaalde onderbreking, zal je de vectortabel moeten overschrijven en het adres horende bij de onderbreking (stel int 2) te laten verwijzen naar de door ons gekozen subroutine in het geheugen.

Systeemoperaties zijn instructies die te maken hebben met de controle van de machine zelf en dus zeer machine specifiek. Ze bepalen o.a. instructies voor **manipulatie van de processortoestand**: onderbrekingen aan/uit, bswap,...

Instructiecodering

De instructiecodering is het finaal vertalen van de assemblercode naar binaire code. Dit gebeurt normaal door een assemblerprogramma.



De eerste 7 bits zijn de **opcode**, ofwel de opdracht die de instructie dient uit te voeren. Hier geeft ze aan dat er een constante moet worden opgeteld bij de accumulator. De 8ste bit geeft aan dat het om een woord (16 bit) gaat. De constante 2 die volgt in 2 byte (16 bit) is in little endian.

Het **databoek** bij een processor zal elke instructie in detail beschrijven. SUB zegt dat het om een subtraction gaat, 001010 is de opcode, en O D I T S Z A P C geeft aan welke vlaggen zullen worden gebruikt in het EFLAGS register.

Compilers, linkers en laders

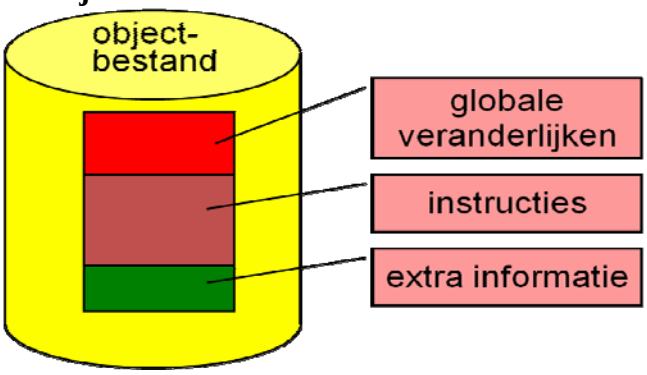
1/ Programmaontwikkeling
Het proces waarbij broncode wordt omgezet tot een uitvoerbaar bestand voor een bepaald platform (=programmaontwikkeling) bestaat uit een aantal stappen:

- **compiler** zet de broncode om in een **objectbestand**: vertaling naar assemblercode, en uiteindelijk naar binaire machinecode.
- men splitst een programma meestal op in meerdere broncodebestanden → meerdere objectbestanden. De **linker** zal alle objectbestanden samenbrengen en voegt er eventueel nog routines uit bibliotheken aan toe, om zo een uitvoerbaar bestand te maken. Een bibliotheek is een verzameling van gecompileerde routines om specifieke functionaliteit mogelijk te maken (I/O activiteiten, ...)

2/ Compiler
Het proces hierdoor uitgevoerd wordt opgesplitst in verschillende stappen:

- **lexicale analyse**: vrije tekst wordt geïnterpreteerd en omgezet tot elementaire **lexemen** van de taal. a, 12, then, while, (
- **syntactische analyse**: interpreteren van sequenties lexemen. if (a<b) then..
- **semantische analyse**: controle indien alle symboolnamen gedeclareerd zijn, en alle datatypes van alle operandi wordt gecontroleerd
- **optimalisatie**: programma optimaliseren
- **codegeneratiestap**: omzetting naar assembler
- **scheduling**: volgorde van instructies, berekeningen en geheugentoegangen veranderen om het resulterende programma nog sneller uit te voeren.

3/ Objectbestand



Bestaat uit 3 delen:

- **globale veranderlijken**
- **instructies**: binaire programmacode zelf
- **extra info**: info mbt het objectbestand nodig in de linker stap en tijdens uitvoering. (bvb naam, grootte, adres waar hoofdprog begint, ...)

4/ Linker

Verschillende objectbestanden samenvoegen tot een uitvoerbaar bestand. Het brengt dus allerlei info samen, en lost externe wijzigingen op. Met andere woorden, wanneer een bepaald objectbestand een externe routine of veranderlijke gebruikt, gaat de linker op zoek naar een uniek voorkomen van deze globale routine of veranderlijke in een lijst van andere objectbestanden. Ook wordt er nog aandacht besteedt aan **herloceerbare** adressen = adressen die nog geen finale waarde hebben gekregen en tijdens het linken nog moeten worden aangepast.

5/ Lader

Het uitvoerbare bestand wordt nu door de lader ingeladen in het geheugen. De verschillende geheugensegmenten worden opgeladen (veranderlijken,...) en bepaalde delen van het geheugen worden gereserveerd en registers worden klaargezet. De lader moet nu de finale adresaanpassingen doen. Welke adressen moeten worden veranderd, werd tijdens het compileren en linken gecreëerd en bewaard in het uitvoerbare bestand.
Het **grabbelgeheugen** is een deel van het geheugen welke beschikbaar is om dynamisch objecten te alloceren.

Secundair geheugen

Magnetische schijven

- zeer hoge opslagdichtheid, relatief snelle toegangstijd
- blijven draaien, ook wanneer inactief. Draaien aan een **constante rotatiesnelheid**
- samengesteld uit stapel van enkele platen die op enkele millimeter van elkaar zijn geplaatst. Ze zijn verbonden aan een gemeenschappelijke as in het midden, en bevatten een kop die kan bewegen van de rand naar het midden van de oppervlakken en omgekeerd.
- elke plaat heeft 2 oppervlakken (aluminium/glas) bedekt met magnetisch materiaal (ijzeroxide). 1'en en 0'en worden bewaard door magnetiseren van kleine gebieden van het materiaal.
- per oppervlak is er 1 beweegbare kop die de platen lezen en schrijven en zweven boven het oppervlak. Een oppervlak bestaat uit verschillende **sporen**, welke zijn opgedeeld in verschillende **sectoren** van typisch 512 bytes. Om de positionering van de koppen te vergemakkelijken zijn er lege ruimtes tussen de sporen (**intertrack gaps**) en tussen de sectoren (**intersector gaps**).
  De verzameling sporen (1 per oppervlak) die per armpositie van de kop kan worden gelezen, noemt men een **cilinder** → lezen van sporen binnen een cilinder vereist geen armbeweging en kan dus zeer snel.
- toegangstijd: drie factoren
  - **zoektijd:** tijd voor het verplaatsen van de kop naar de juiste cilinder (grootste tijd)
  - **latentietijd:** tijd voor de juiste sector in de cilinder te vinden
  - **transfertijd:** tijd om gegevens te lezen en te schrijven als de kop juist is gepositioneerd

- enkele begrippen:
  - **spoor/spoor tijd**: tijd nodig om de kop van het ene spoor naar het andere te brengen (/ = naar)
  - **toerental:** aantal omwentelingen per minuut van de platen
  - **latentie:** tijd nodig om een halve toer te maken
  - **BW disk:** maximale transfersnelheid van de harde schijf
  - **BW host:** maximale bandbreedte naar de host-computer
  - **interne buffer:** schijven bevatten deze omdat computers sneller gegevens naar de schijf kan transfereren dan dat ze fysiek op de schijf kunnen worden geschreven. De gegevens worden hier tijdelijk in opgeslagen vooraleer ze weggeschreven worden
  - **MTBF:** de gemiddelde tijd zonder falen
  - **MCB (Master Control Block):** gereserveerd deel van de schijf welke bijhoudt hoe de rest van de schijf eruit ziet. Bevat vaak ook een programmaatje welke wordt gebruikt om het besturingssysteem op één van de partities op te starten.

Floppy disk

- werkt volgens zelfde principe
- is veel kleiner (1,44 MB), draait niet indien niet gebruikt en de bandbreedte is ongeveer 500 kb/s ofwel 64,5 kB/s
- in tegenstelling tot hierboven, raakt de lees/schrijfkop wel contact met het oppervlak. Dit zorgt mede voor slijtage.

Tapes
- vooral voor backup en is goedkoop
- omslachtig in gebruik voor willekeurige lees/schrijftoegang want de tape moet steeds worden opgewonden naar de correcte positie.
- opslag gebeurt 2-dimensionaal: zowel in breedte als in lengte

CD Rom
- bestaan uit een plastieken onderlaag bedekt met aluminium.
- de data wordt opgeslagen als kleine vlakken en putjes in de onderlaag, die het licht anders reflecteren
- ze worden gedrukt met een **master**. Dit is van zeer hoge kwaliteit, maar data cd's zijn dit niet. Vandaar dat er gebruik gemaakt wordt van ingewikkelde foutcorrigerende code bij het schrijven van de data als bits op de CD.
- geen sporen, maar een **lange spiraal**.
- draaien aan **constante lineaire snelheid**. Dit wil zeggen: de stroom bits per seconde is dezelfde dus de rotatiesnelheid is hoger voor data in het midden van de schijf, want er geldt constante lineaire snelheid, en de putjes zijn op dezelfde afstand van elkaar geplaatst over de hele cd-rom.

DVD Rom & Blue Ray
- putjes en vlakjes worden kleiner gemaakt, en liggen dichter bij elkaar (4,7 - 27 GB ipv 650MB)

Invoerapparaten

Toetsenbord
- QWERTY / QWERTZ (Duitsland) / AZERTY / DVORAK
- bij toetsaanslag sluit de onderliggende schakelaar, en de scancode (x,y) wordt omgezet naar een letterteken.
- bevatten soms een buffer, wanneer de computer de toetsaanslagen niet tijdig kan verwerken
- soms wordt door **trilling** de schakelaar tweemaal gesloten, maar toch geeft dit maar 1 toets door
- **typematics:** karakteristiek van het toetsenbord. Het geeft aan hoe vaak een teken herhaald wordt per tijdseenheid wanneer de toets ingedrukt blijft.

Muis
- **trackball**
- **touchpad**
- **pointing stick:** in laptops tussen de toetsen
- **joystick:** xy-positie + rotatie
- **optisch/laser**

Lichtpennen en aanwijsschermen
- lichtpen vangt licht op van een scherm. Schermpixels worden gerefreshed: ze gaan oplichten en dan weer uitdimmen. De lichtpen detecteert het moment wanneer de pixels oplichten, wat kan worden vertaald in een geselecteerde positie.
- aanwijsschermen:
    - **optisch:** lichtstralen van LED's / camera's lopen over het scherm en vangen onderbroken stralen op.

- **elektrisch:** druk wordt opgevangen

Digitiseertablet

- soort muismat met een soort muis of aanwijspen. Aanraking op de mat wordt vertaald in xy coördinaten
-

Uitvoerapparaten

Laserprinter

- bestaat uit een elektrisch geladen **drum**
- een **laser** bestraalt via een roterende spiegel de drum en ontlaadt een aantal gebieden. Lijn voor lijn passeert de laser over de drum, die verder ronddraait.
- de geladen gebieden op de drum pikken elektrostatisch geladen **tonerpoeder** op
- het papier wordt opgewarmd en door druk en warmte van de rollers, wordt het tonerpoeder bevestigt op het papier.
- de drum wordt intern gekuist, en een nieuw blad kan worden afgeprint.

Ink Jet printer

- spuit kleine druppeltjes inkt op het blad → 600 dpi betekent 600 puntjes per duim, dus 0,045 mm per punt
- twee methodes om een druppeltje inkt te vormen:
  - **thermisch:** dampbelvorming door opwarming
  - **piezo-elektrisch:** kristal welke vervormt onder invloed van spanning en dit wordt gebruikt als zuiger om druppeltjes inkt naar buiten te persen

Video Display

- **CRT = Kathodestraalbuis**: elektronenstraal uit het elektronenkanon wordt lijn per lijn, van onder naar boven, van links naar rechts over het scherm bewogen. De binnenkant van het beschoten scherm bevat een fosforbevattende laag, welke oplicht wanneer het wordt geraakt. (rood, groen, blauw fosfor.. met bijhorende elektronenkanonnen).
- **LCD = Liquid Cristal Display**: opgebouwd uit vloeibare, al of niet lichtdoorlatende kristallen. Ze laten licht door afhankelijk van de aangelegde spanning. Deze kan op 2 manieren worden aangelegd:
  - **in een passieve matrix LCD** wordt ze aangebracht door rij en kolom van een pixel aan te sturen, zodat op het kruispunt een spanning ontstaat
  - **in een actieve matrix LCD** gebeurt dit via een afzonderlijke transistor per pixel. (meer betrouwbaar, nauwkeurig)

  Gewone LCD schermen werken niet zonder licht, maar de meeste zijn **backlit**.

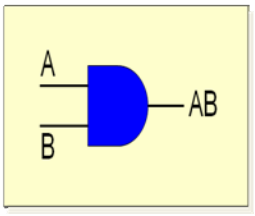
- enkele begrippen:
  - **schermafmetingen:** lengte van de diameter in inch
  - **aspect ratio:** verhouding van de horizontale & verticale lengte van het scherm
  - **oriëntatie:** standaard = landschap: breder !
  - **kleurendiepte:** aantal kleuren op het scherm
  - **true color:** 8 bits per RGB → 16,8 miljoen kleuren
  - **resolutie:** hoeveel verschillende pixels worden op het scherm getoond. **XGA** = 800x600, true color; **UXGA** = 1600x1200, true color
  - **videogeheugen:** wordt bepaald door de maximale resolutie & kleurendiepte
  - **dot pitch:** afstand tussen elementaire schermpunten

- **refresh rate:** tempo waaraan een scherm wordt ververst (zonder flikkering = >= 72Hz)
- **VGA:** binair beeld omzetten in analoog signaal welke naar de monitor wordt gezonden
- **DVI:** digitale informatie rechtstreeks sturen naar digitale monitor. Hierbij treedt er dus geen kwaliteitsverlies op tijdens transport. Monitor en grafische kaart moeten in dat geval wel samen kunnen werken

Logische poorten

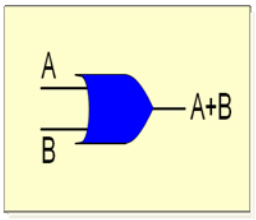
and

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1



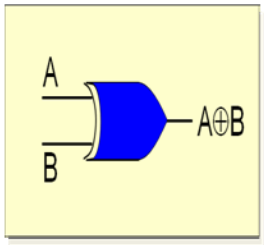
or

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1



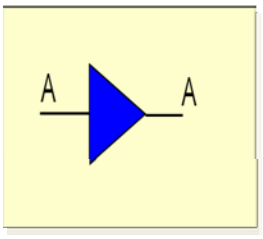
Xor (=EOF)

A	B	A⊖B
0	0	0
0	1	1
1	0	1
1	1	0



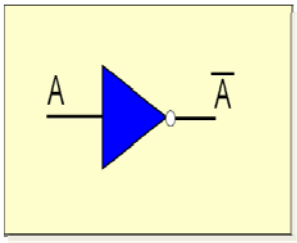
buffer **(signaal versterken over lange afstand)**

A	A
0	0
1	1



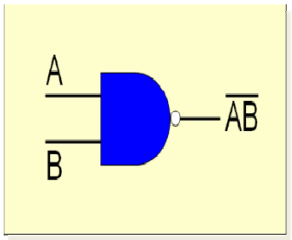
Invertor (NOT)

A	$\overline{A}$
0	1
1	0



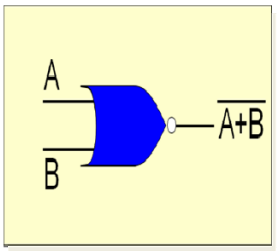
nand

A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0



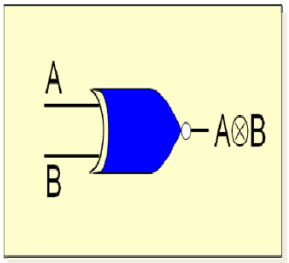
nor

A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

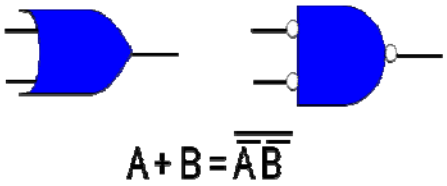


xnor

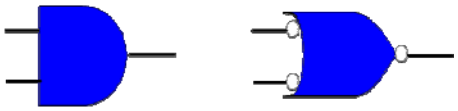
A	B	A⊗B
0	0	1
0	1	0
1	0	0
1	1	1



De Morgan



$$A + B = \overline{\overline{A}\,\overline{B}}$$



$$AB = \overline{\overline{A} + \overline{B}}$$

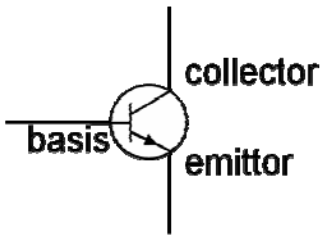
Boolese Algebra

Commutativiteit	$AB = BA$
Distributiviteit	$A(B+C) = AB+AC$
Neutraal element	$1A = A$
Complement	$A\overline{A} = 0$
Nuleigenschap	$0A = 0$
Idempotentie	$AA = A$
Associativiteit	$A(BC) = (AB)C$
Dubbele negatie	$\overline{\overline{A}} = A$
De Morgan	$\overline{AB} = \overline{A} + \overline{B}$
Consensus	$AB + \overline{A}C + BC = AB + \overline{A}C$
Absorptie	$A(A+B) = A$

Commutativiteit	$A+B = B+A$
Distributiviteit	$A+(BC) = (A+B)(A+C)$
Neutraal element	$0+A = A$
Complement	$A+\overline{A} = 1$
Eéneigenschap	$1+A = 1$
Idempotentie	$A+A = A$
Associativiteit	$A+(B+C) = (A+B)+C$
Dubbele negatie	
De Morgan	$\overline{A+B} = \overline{A}\,\overline{B}$
Consensus	$(A+B)(\overline{A}+C)(B+C) = (A+B)(\overline{A}+C)$
Absorptie	$A+(AB) = A$

Transistorniveau

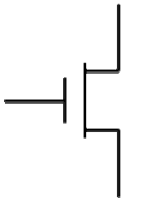
Transistor



halfgeleiderschakelaar

- Het is een schakelaar zoals het relais, maar wordt in veel kleinere oppervlakte geproduceerd, en bevat geen mechanisch, bewegende delen.
- Wordt gemaakt in halfgeleidertechnologie.
- Indien er stroom loopt door de basis, kan er stroom vloeien tussen **connector en emittor (→ schakelaar)**

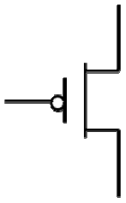
MOS transistor



NMOS

Geleidt indien input hoog

Gebruikt om output laag te brengen



PMOS

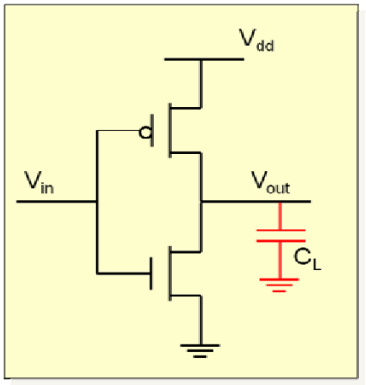
Geleidt indien input laag

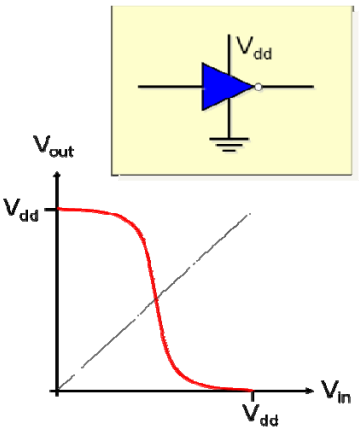
Gebruikt om output hoog te brengen

In alle chips worden tegenwoordig MOSFET transistors gebruikt (metal oxide semiconductor field-effect transistor). Hiervan bestaan twee varianten: NMOS en PMOS. NMOS blijkt bijzonder geschikt voor om een verbinding met de massa (aarding) tot stand te brengen, en PMOS voor een verbinding met de voedingsspanning. Meestal zullen ze dan ook in paren worden gebruikt. Indien beide op dezelfde chip voorkomen spreekt men van **CMOS**.

Voorbeelden

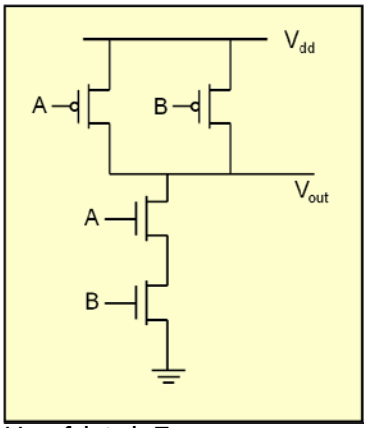
1/ Invertorpoort





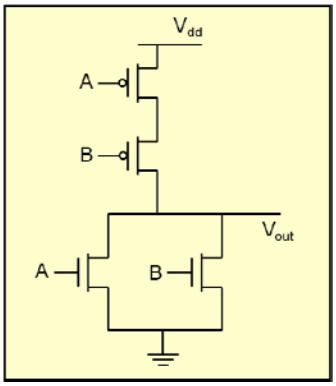
Aan de uitgang zien we: C_L (=capaciteit). Deze is in werkelijkheid niet aanwezig, maar modelleert de capaciteit van alle uitgaande bedrading. De ingang van een transistor in CMOS verbruikt *geen* stroom. De enige stromen die vloeien zijn om de parasitaire capaciteiten van de bedrading te laden en te ontladen. Eenmaal in evenwicht, stopt ook deze stroom. In rust verbruikt deze schakeling dus bijna geen stroom, bij omschakelen wel (hoe hoger de klokfrequentie, hoe meer er omgeschakeld wordt).

2/ Nand-poort



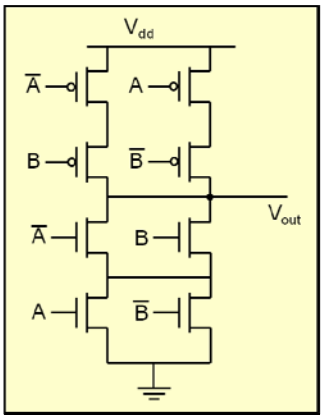
A	B	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

3/ Nor-poort



A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

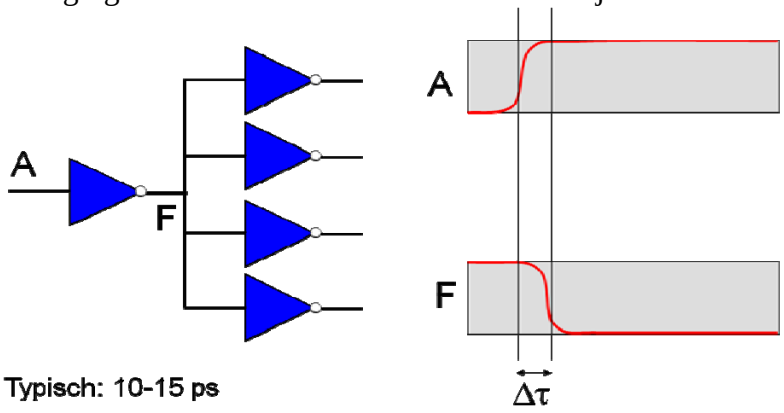
4/ Xnor-poort



A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

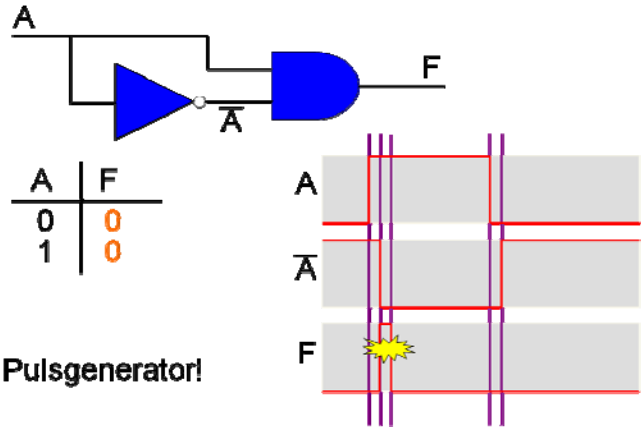
Tijdsgedrag van poortjes

Er is altijd een zekere tijd nodig vooraleer de uitgang van een poort de correcte waarde aanneemt, na een verandering aan de ingang. Dit noemen we **poortvertraging** of **propagatietijd**. Dit kan soms onaangename gevolgen hebben!
Een gangbare manier om een poortvertraging op te geven is FO4 **(fan out of 4)**. Dit is de vertraging die een invertor ondervindt indien hij vier identieke invertoren dient aan te sturen.

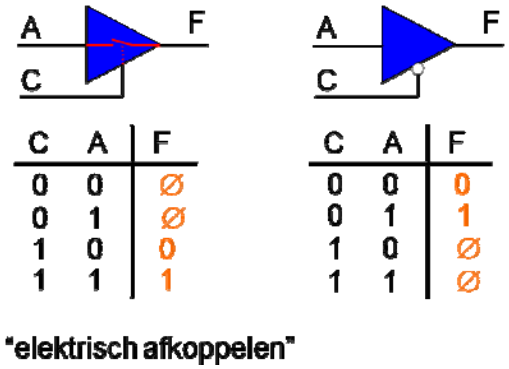


Typisch: 10-15 ps
Klokperiode > 12-16 FO4

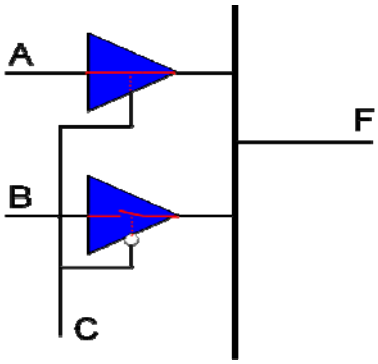
Een mogelijk gevaar van propagatietijden is de **glitch**. Dit is een ongewenste toestand of uitgang, veroorzaakt door het tijdsgedrag van poortjes. Hieronder doet zich een glitch voor wanneer A wisselt van 0 naar 1.



Tri-state buffer



Een tri-state buffer gedraagt zich wel als een gewone buffer, maar er is een extra controle ingang aanwezig die toelaat de buffer af te koppelen. Er zijn dus drie uitgangswaarden: 0, 1 of afgekoppeld. In afgekoppelde toestand is er geen uitgangswaarde omdat de uitgang **elektrisch afgekoppeld** is. We noemen de uitgang ook **hoog-impedant**. Ze bestaat in twee vormen: met **directe controle** (links) en met **geïnverteerde controle** (rechts).



We gebruiken tri-state buffers om een aantal logische poorten aan te sluiten op een gemeenschappelijke lijn (bus), zonder kortsluiting te veroorzaken, zolang er slechts 1 buffer tegelijk is aangeschakeld. Dit gebeurt door een geïnverteerd controlesignaal!

Connecties

Als verschillende verbindingen op een tekening door en over elkaar lopen, moeten we kunnen aanduiden welke kruisen, en welke zijn geconnecteerd. Met een T-kruising, of een bolletje zijn ze connected.

Productie van chips

Chips of halfgeleidercircuits worden geproduceerd op een grote, ronde siliciumplaat = **wafer**, typisch 20-30 cm diameter. Per plaat kunnen enkele tot vele tientallen circuits worden gefabriceerd (1 circuit = **die**). De chip bestaat uit vele miljoenen transistors die onderling verbonden worden door middel van metaalbaantjes in verschillende niveaus.
Tussen het kanaal waar de stroom doorloopt, en de “ingang = gate” van de transistor, bevindt zich een **oxidelaag**.

Wet van Moore

= Het aantal transistors per chip verdubbelt nagenoeg iedere 2 jaar. De geheugenchips liggen voor op de processor omwille van hun eenvoudigere structuur.

Combinatorische schakelingen

Realisatie van Boolese functies

Een meer ingewikkelde Boolese functie kan worden geïmplementeerd via een **som van producten**. Dit is het OR'en van een aantal AND resultaten.
Voor elke rij waar het resultaat een 1 bevat, stelt men de **minterm** op, dit is een productterm die elke variabele precies 1 keer bevat. De som van alle mintermen kan in de logica worden geïmplementeerd aan de hand van NOT-poorten, multi-input AND-poorten, en multi-input OR-poorten.

Je kan evengoed een Boolese functie realiseren aan de hand van een **product van sommen**. Men noteert dan de mintermen voor die rijen die 0 hebben als resultaat, en inverteert de ganse som van producten-rij (via De Morgan)

Minimalisatie

Som van producten of product van sommen kan vaak onnodig complex zijn. Dit proces gaat op zoek naar de eenvoudigste implementatievorm.

Complexiteit van een realisatie

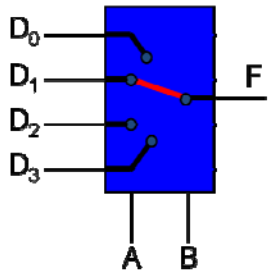
Kan kwantitatief uitgedrukt worden door een telling te maken van het totaal aantal poorten, het totaal aantal inputs, de maximale fan-out (maximaal aantal uitgaande verbindingen), en maximale fan-in (macimaal aantal inkomende verbindingen).

Computationele compleetheid

De minimale poortjes nodig om een willekeurige combinatorische functie te implementeren. **{AND,OR,NOT} en {NAND} en {NOR}** zijn computationeel compleet.

Digitale componenten

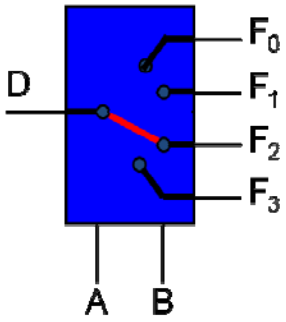
<u>1/ Multiplexer (MUX)</u>



A	B	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Zet een aantal ingangen om tot 1 enkele uitgang. Men kan hiermee Boolese functies implementeren door inputs (A,B) aan te leggen aan de controlelijnen en de resultaten uit de waarheidstabel als constanten aan te leggen aan de data-ingangen.

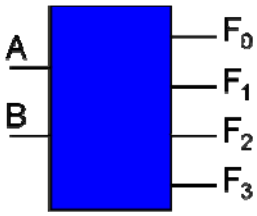
2/ Demultiplexer (DEMUX)



A	B	F_0	F_1	F_2	F_3
0	0	D	0	0	0
0	1	0	D	0	0
1	0	0	0	D	0
1	1	0	0	0	D

Eén data-ingang wordt doorgegeven aan één van de data-uitgangen. Toepassing: zenden van data van 1 bron naar 1 bestemming te kiezen uit een groep van mogelijke bestemmingen.

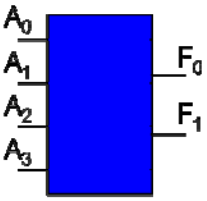
3/ Decoder



A	B	F_0	F_1	F_2	F_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Vertaalt een code in een spatiale locatie. Welke uitgang op 1 staat, wordt bepaald door de code aan de ingang. Kan gebruikt worden om andere circuits aan te sturen.

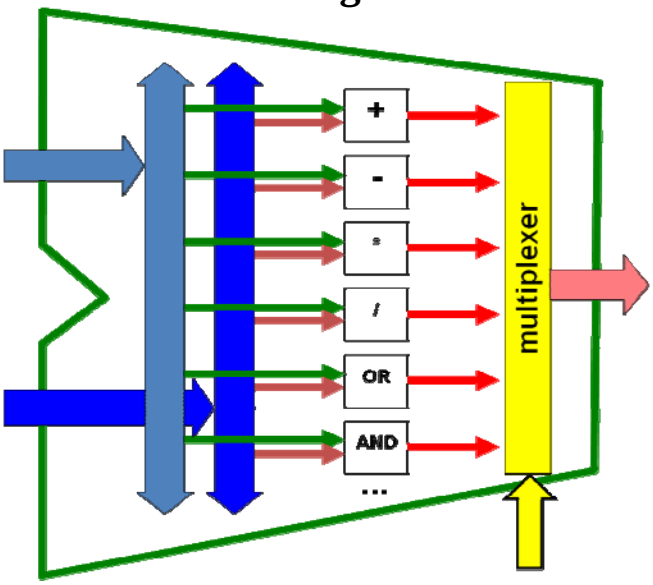
4/ Prioriteitscodeerder



A_0	A_1	A_2	A_3	F_0	F_1
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

Legt een prioriteit op aan de ingangen. De uitgang geeft de binaire waarde weer van de ingang met de hoogste prioriteit die is aangeschakeld. A0 heeft hogere prioriteit dan A1 enz.

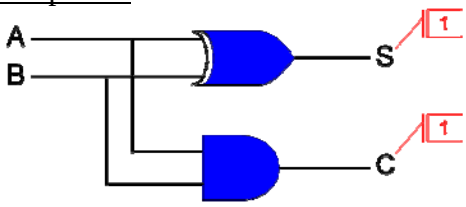
De ALU-schakelingen



Een ALU bestaat uit een parallelle berekening van de ALU-functies en dan de selectie van het gewenste resultaat door de MUX op het einde.

Optellers en aftrekkers

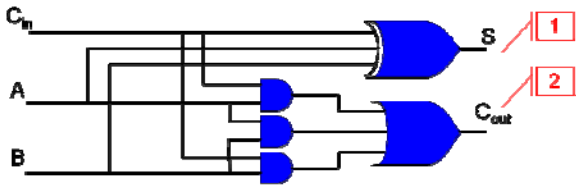
1/ Halve opteller



Berekent de som van 2 bits, en stelt het resultaat voor door 2 bits: som en carry. Zowel som en overdracht worden gegenereerd na 1 poortvertraging.

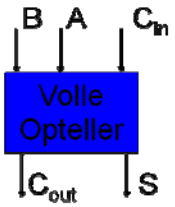
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

2/ Volle opteller

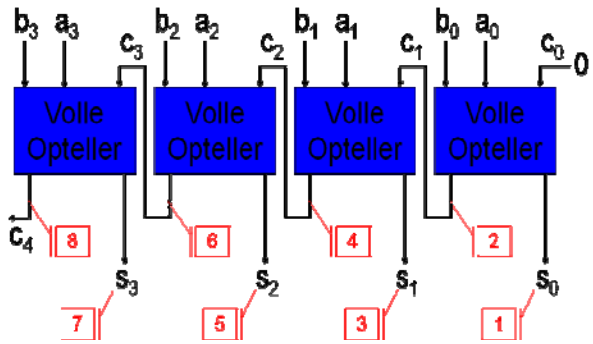


Maakt de som van 3 bits: 2 operandi, en 1 inkomend carry bit. Het resultaat duurt 1 poortvertraging, de overdacht 2.

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

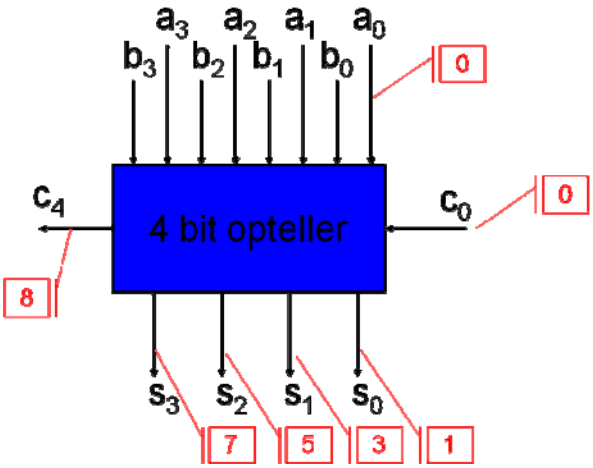


3/ Doorsijpelopteller



m-bit optelling realiseren door m Full Adders in serie te zetten. De som zal worden gegenereerd na 2*m – 1 poortvertragingen, en de overdracht na 2*m vertragingen.
Een methode om carrypropagatie te versnellen is de C_{out} herschrijven aan de hand van minimalisatie: $C_{out} = G + P\,C_{in}$ met G=AB en P = A+B. (slide 72)

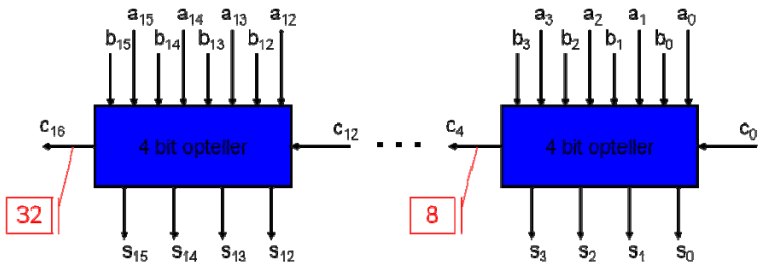
4/ 4-bit opteller

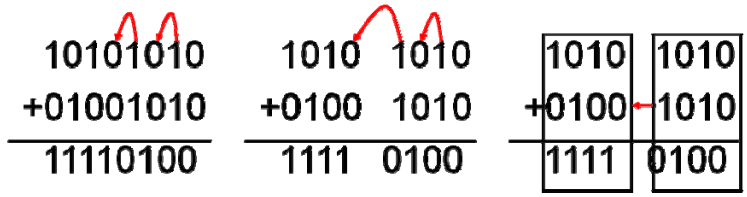


Uitgangen zijn de 4 sommatiebits en de overdracht bij de meest beduidende bit. Kan worden opgebouwd zoals hierboven.
Ook hier kan carry-propagatie worden versneld door minimalisatie.

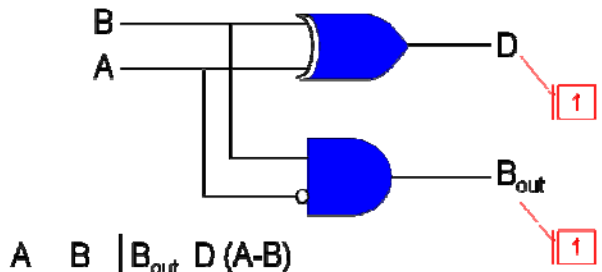
$$C_4 = G_3 + P_3 C_3$$
$$C_3 = G_2 + P_2 C_2$$
$$C_2 = G_1 + P_1 C_1$$
$$C_1 = G_0 + P_0 C_0$$

5/ Cascade van 4-bit optellers





6/ Halve aftrekker

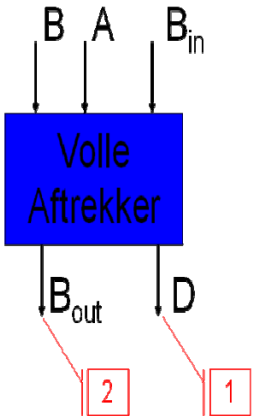


A	B	B_{out}	D (A-B)
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

A-B wordt berekend, en het resultaat wordt voorgesteld door Difference en Borrow.

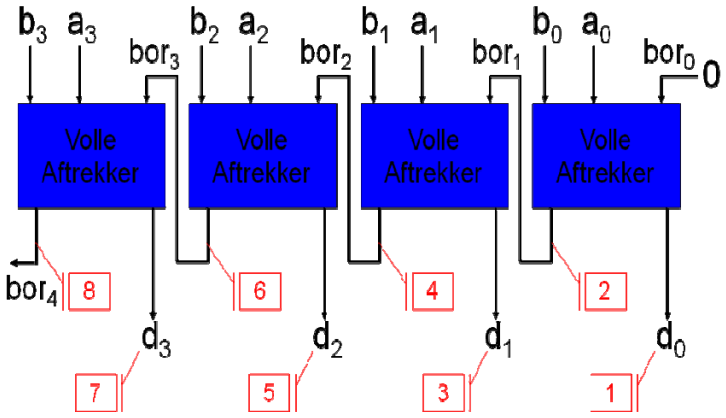
7/ Volle aftrekker

A	B	B_{in}	B_{out}	D
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

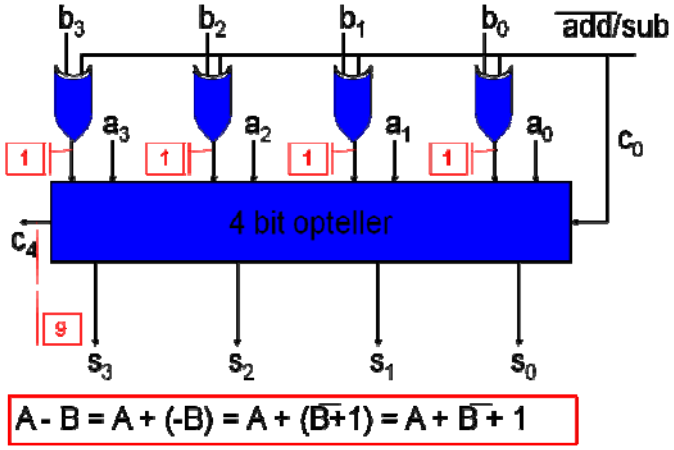


A-B-B_{in}, waarbij <B_{out}:D> het 2bits 2 complement is van A-B-B_{in}.

8/ Doorsijpelaftrekker

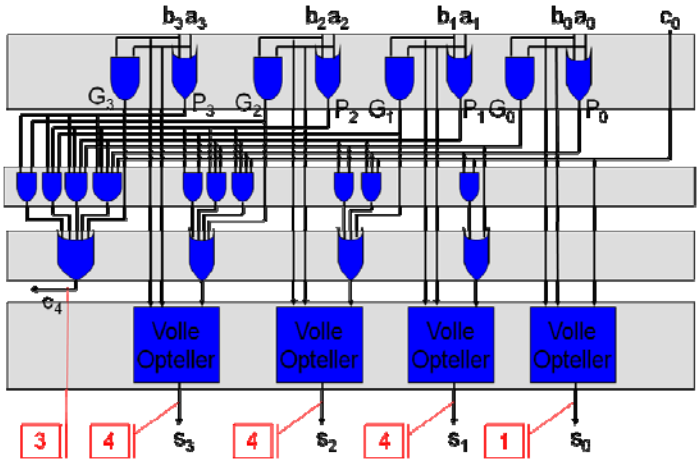


9/ Opteller/aftrekker



Uiteraard kan men A-B ook berekenen als A+(-B). De niet-add/sub lijn geeft aan indien het om een optelling of aftrekking gaat. De overige ingangen zijn A en B of (-B). Als de lijn=1, zullen de XOR poorten en c0=1 ervoor zorgen dat –B berekend wordt (invertering van alle bits + 1 (c0)). Indien lijn = 0, wordt A+B berekend.

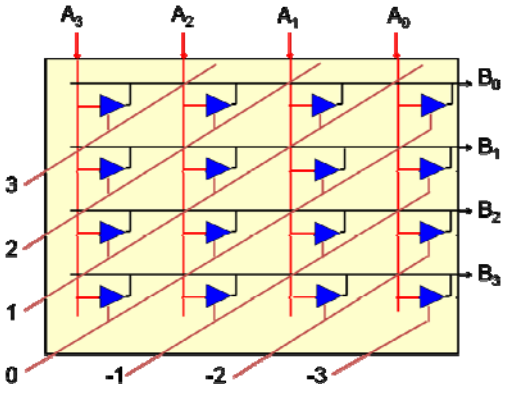
10/ Carry Lookahead opteller



De aanwezige volle optellers berekenen hier geen carry bit meer, omdat uitgaande/inkomende carry bits rechtstreeks worden berekend uit de operandi. Dit is de implementatie van minimalisatie bij doorsijpeloptellers...

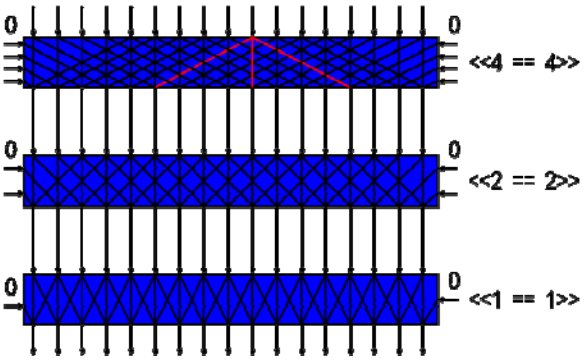
Verschuivers

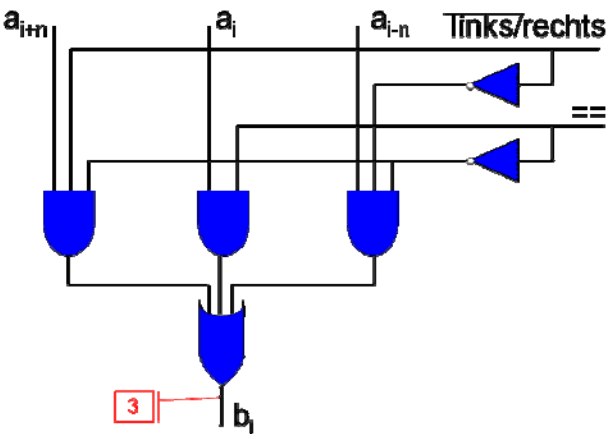
1/ Barrel shifter



In staat om in 1 poortvertraging een bitpatroon over n willekeurige bits voorwaarts of achterwaarts te verschuiven. Hiertoe moeten de correcte tri-state buffers worden aangestuurd. Voor kleine bitpatronen te verschuiven werkt dit goed, maar voor grote loopt het aantal buffers en de fan-out (van A_i) zeer sterk op. Er is echter een **verborgen kost!!** Er zal een decoder nodig zijn om de signalen 3,2,1,0,-1,-2,-3 te generen uitgaande van de 2-complementsvoorstelling van de verschuiving. (→ poortvertragingen!)

2/ Logaritmische shifter





Betere oplossing dan de barrel shifter. Kan een woord (32 bit) aan de ingang verschuiven over een willekeurige afstand afhankelijk van de ingangen.
In elke laag/stap van de shifter wordt een **log shifter cel** gebruikt om per bitpositie de nieuwe bitwaarde te berekenen. Dit gebeurt door combinatie van drie bits (zie tekening) Dit zijn dus de bits van de positie zelf, en de bits van de positie n bits naar links en n bits naar rechts. N zal dan 1,2,4,... zijn.
Daarnaast zijn er nog controlesignalen die aangeven of er geschoven wordt of niet, en in welke richting.

Vermenigvuldigers

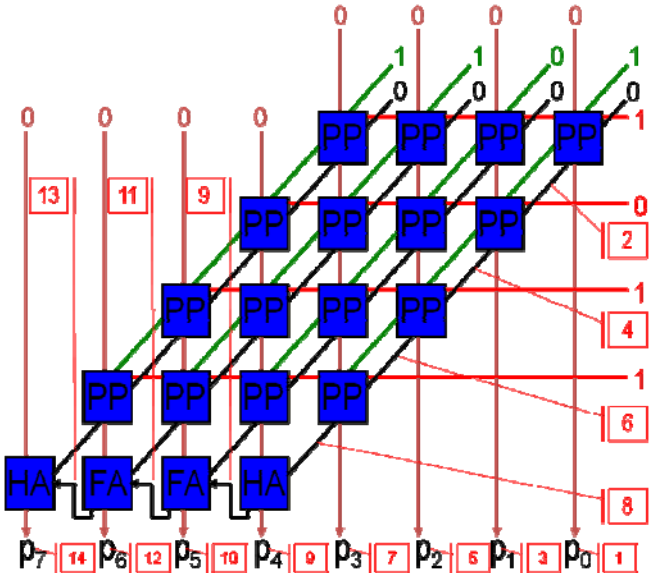
1/ Werkwijze
Hoe het in zijn werk gaat: slides 81-97!

2/ Algoritme van Booth
Indien we in de plaats van een 4 bit opteller, een opteller/aftrekker gebruiken, dan kunnen we het product in sommige gevallen aanzienlijk versnellen door gebruik te maken van het Algoritme van Booth. Indien de vermenigvuldiger uit een groep van 1 bitjes bestaat, kunnen deze worden gegroepeerd en uitgedrukt als een verschil. Men moet dus niet alleen kijken naar het bit waarmee men moet vermenigvuldigen, maar ook naar het bit waarmee men net vermenigvuldigd heeft. Wanneer er een overgang is van 0 naar 1, volgt er een rij 1tjes (soms 1, soms meerdere). Op dit ogenblik moet men het verschil maken ipv de som. Wanneer de rij 1tjes overschakelt naar een 0, moet men de som maken.

M x 00011110
= M x (16 + 8 + 4 + 2)
= M x 30
= M x (32 – 2)

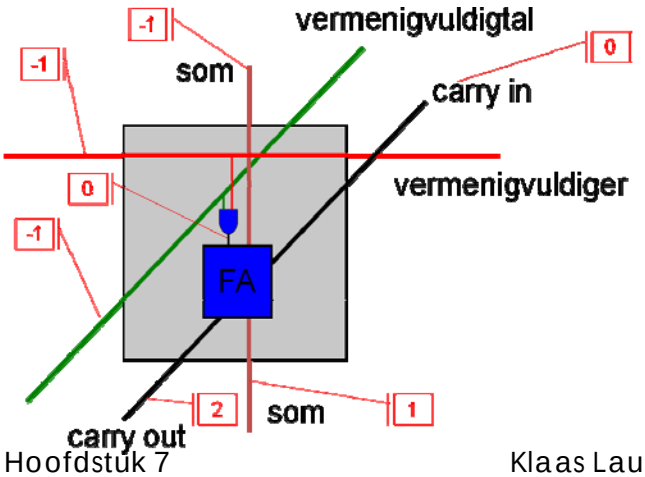
Men moet echter wel vertrekken van een 0 en een leidende 0 toevoegen om het algoritme goed te initialiseren.

3/ Carry-save vermenigvuldiger



In het groen staat het vermenigvuldigtal, in het rood (rechts) de vermenigvuldiger. Eerst wordt hier het product gemaakt van elke bit in vermenigvuldigtal, en elke bit in vermenigvuldiger, en dan worden de partiële productelementen opgeteld. Men noemt dit carry save, omdat de carry naar **links onder** ipv naar links wordt doorgegeven → doorgegeven naar de volgende partiële som. Deze implementatie kan nog worden versneld door de carry look-ahead toe te passen!

Hieronder wordt een partieel product element in detail getoond.



Vermenigvuldigen van 1 bit van vermenigvuldiger en 1 bit van vermenigvuldigtal. Het resultaat wordt opgeteld bij de andere bits in dezelfde kolom, en bij de inkomende carry. Een negatieve poortvertraging wordt gebruikt om aan te geven dat het resultaat reeds 1 poortvertraging aanwezig is.

Delers

1/ Restoring deling
Werkwijze: zie slides 105-128

2/ Non restoring deling
Het idee is dat indien (Deeltal-Deler) < 0 is, het resultaat van
(Deeltal-Deler + Deler) * 2 - Deler = Deeltal * 2 - Deler
wat hetzelfde is als:
(Deeltal-Deler) * 2 + Deler = Deeltal * 2 - Deler

Door te onthouden dat het vorige verschil < 0 was, en in een volgende stap de optelling uit te voeren, en dan de test te doen, bereikt men hetzelfde resultaat, zij het met minder operaties.
Men noemt dit een non-restoring deler.
Merk op dat de hardware nodig om te delen en te vermenigvuldigen nagenoeg dezelfde is.

3/ Iteratieve deling
De berekening van 1/b kan worden benaderd door een iteratief proces.

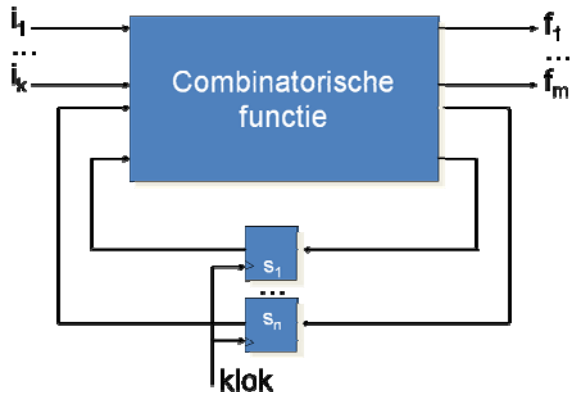
Benadering van 1/b: $x_{i+1} = x_i(2-x_i b)$

Zodoende zijn enkel vermenigvuldiging en aftrekking nodig!

Sequentiële logica

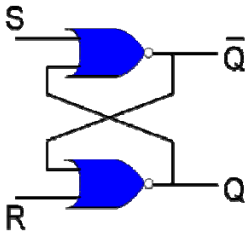
Wat is sequentiële logica

We noemen dit ook wel final state machine. Het neemt een ingang en een huidige toestand en vertaalt die via een combinatorische functie in een uitgang en een nieuwe toestand.

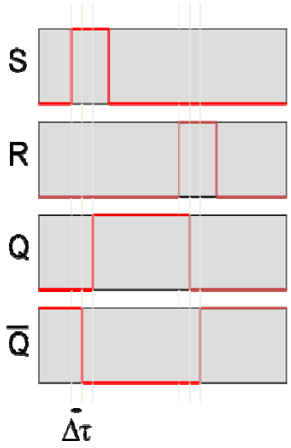


De geschiedenis van ingangen bepaalt mee de toestand en de uitgang.
Dit is het klassieke model van een toestandsautomaat. De combinatorische functie vertaalt de ingangsbits i_1 tot i_k samen met de interne toestandsbits s_1-s_n in nieuwe toestandsbits, en uitgangsbits f_1-f_n. De output van de combinatorische functie wordt bewaard door een speciale component en geeft deze op het gepaste ogenblik (klok!) door aan de input van het combinatorisch circuit. Mocht de klok niet aanwezig zijn, wordt de output meteen teruggekoppeld naar de input → ontstaan van een onstabiele situatie.

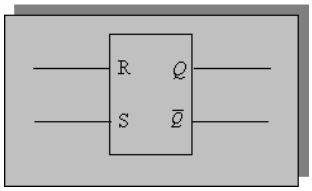
S-R Latch



A	B	A+B
0	0	1
0	1	0
1	0	0
1	1	0



Een geheugencel is een verzameling van logische poorten die een stabiele uitgang kan bewaren zonder dat de ingangen actief moeten zijn. De uitgang van de geheugencel wordt bepaald door de huidige ingangen en de geschiedenis ervan. Een 1 bit geheugencel wordt een **latch** genoemd.
In Q bevindt zich de opgeslagen bit.
Als S=R=0 dan bewaren de NOR poorten de huidige uitgangen Q en niet-Q.

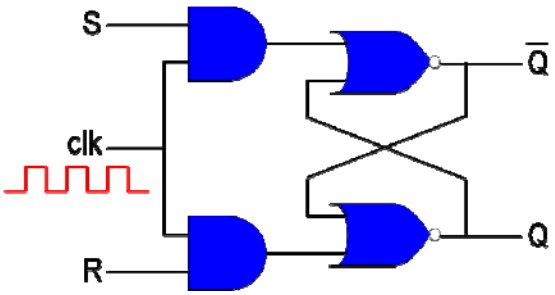


Als Q =R=S=0 en S(et) gaat naar 1, dan zal Q=1 (2 poortvertragingen) en nietQ = 0 (1 poortvertraging). Vanuit de situatie Q=1 S=R=0 kunnen we de oorspronkelijke situatie herstellen door R=1, die langer dan 2 poortvertragingen duurt.

Ingangen		Uitgangen	
R	S	Q	Q'
0	0	Willekeurig	
0	1	0	1
1	0	1	0
1	1	Vorige stand	

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

Geklokte S-R Latch



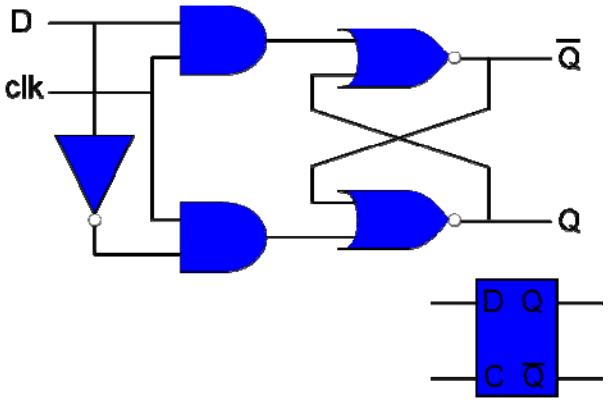
Indien S en R komen uit een ander en ingewikkeld circuit, is het mogelijk dat een aantal ongewenste overgangen gebeuren op die twee vooraleer ze de latch binnentreden. Hiervoor wordt een klok toegevoegd aan de latch, zodat veranderingen aan de bewaarde Q bit enkel gebeuren als S en R stabiel zijn telkens de klok op 1 staat.

De tijd tussen twee opgaande klokflanken = **cyclustijd**. De **kloksnelheid** is de inverse van de cyclustijd.

D Latch

Om een 1 of 0 te bewaren in de S-R latch, moet men een 1 aanleggen aan S of R. De D latch zorgt ervoor dat een 1 of een 0 slechts moet aangelegd worden aan 1 enkele input (D=Data). Het is eigenlijk niets meer als een geklokte S-R latch, waarbij S=D en R=niet-D. Wanneer de klok omhoog gaat, wordt de waarde van D bewaard.

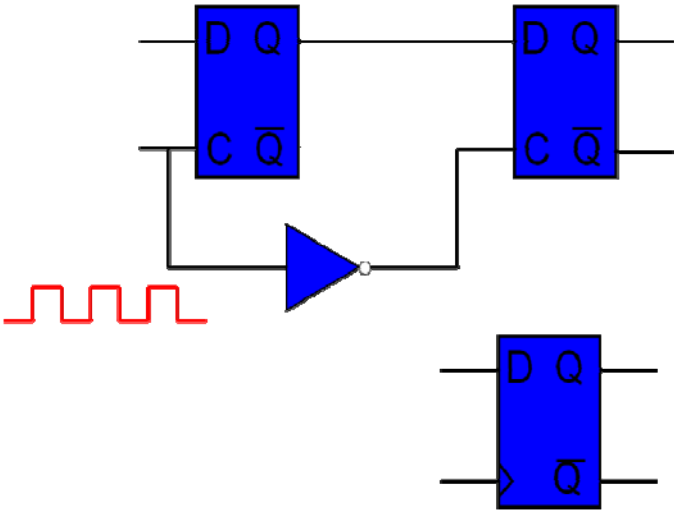
C	D	Q
0	0	Q_{-1}
0	1	Q_{-1}
1	0	0
1	1	1



Indien de klok laag staat, blijft de uitgang dus stabiel, en wanneer de klok hoog staat, volgt de uitgang de D (natuurlijk met poortvertraging).

Master-Slave D-Latch (D Flip Flop)

Om er zeker van te zijn dat de D latch slechts 1 keer per klokpuls van toestand verandert, gebruiken we geen gewone D latch, maar een **master-slave D-latch of een D flip flop**.



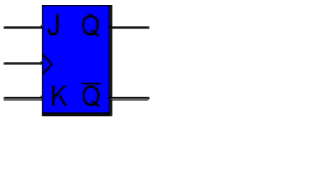
Bestaat uit 2 D latches in cascade, waarbij de tweede de geïnverteerde klok van de eerste gebruikt. De master verandert zijn toestand wanneer de klok hoog is, **de slave wanneer de klok laag** is. De klok moet dus eerst hoog gaan en dan laag vooraleer de waarde wordt opgeslagen in de slave latch.
Het driehoekje hiernaast duidt aan dat transities aan de uitgang enkel gebeuren bij stijgende/dalende klokflanken.
Op de dalende flank van de klok wordt dus de uitgang veranderd!

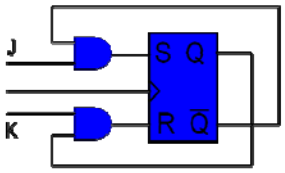
Wel is het zo dat de uitgang niet direct zal worden veranderd, omdat de flipflop wat tijd zal nodig hebben om een stabiele toestand aan te nemen. Men spreekt van de **setup tijd** van de flipflop. Gedurende minstens de setup tijd zal het signaal D stabiel moeten blijven. Sommige flipflops vereisen dat na de dalende flank het signaal D nog even bewaard wordt, we spreken van een **zeer korte!! hold-tijd**.

We spreken van **niveaugestuurde latches** als de toestand wordt gewijzigd wanneer de klok hoog (of laag) is, en van **flankgestuurde flip-flops** die de toestand veranderen tijdens kloktransities.
Een S-R flip flop kan zo ook opgebouwd worden.

J-K Flip Flop

J is ongeveer gelijk aan S, en K aan R.

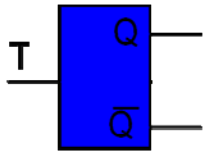




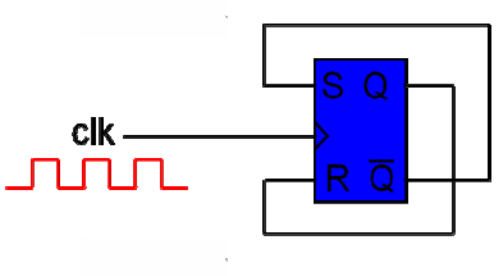
Q_t	J	K	Q_{t+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Er is wel een verschil! Wanneer J=K=1, dan slaat de toestand om (**toggle**). Deze flipflop realiseert 4 functies die men kan uitvoeren op 1 bit: omkeren, onveranderd laten, set (op 1 zetten), en reset (op 0 zetten).

T Flip Flop

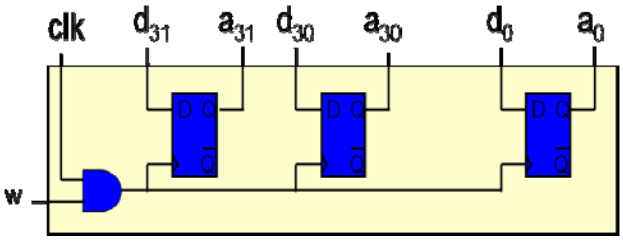


= Een J-K flip flop waarbij J=K=1. Deze flip flop zorgt dus altijd voor een toggle per klokperiode. Het signaal Q zal slechts de halve frequentie hebben van het kloksignaal.

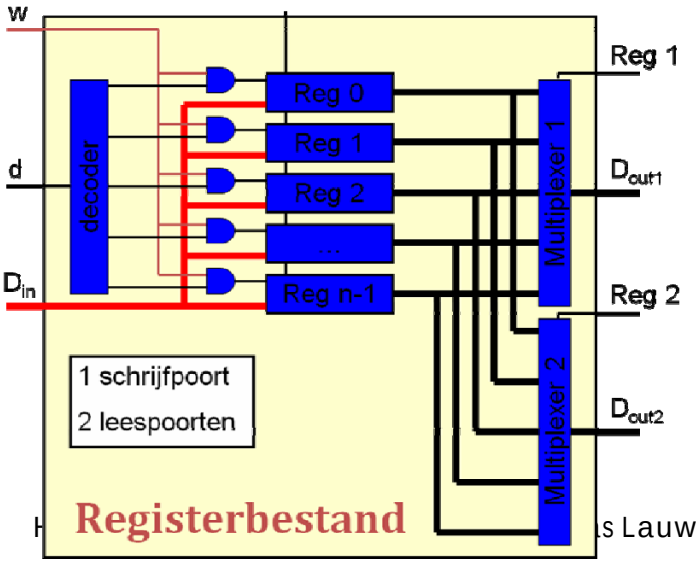


Registers

Een register is een zeer snel geheugen ter grootte van n bits. Een register bestaat uit 1 D flip flop per op te slaan bit. Het liet bij het begin van een cyclus een waarde kunnen beschikbaar stellen, en tegelijkertijd op het einde van die cyclus een nieuwe waarde kunnen opslaan. Dus een register kan binnen dezelfde cyclus belezen en beschreven worden.
Alle flip flops delen dezelfde controlesignalen, en het schrijfsignaal is een combinatie van klok en schrijfcommando “w”; indien w=1, dan wordt de waarde op de D-bus ingeschreven in het register bij dalende klokflank.



Een **registerbestand** bestaat uit een lijst van registers. Men spreekt hierbij van schrijf- en leespoorten. (hier: 1 schrijf, 2 lees)
“d” is het nummer van het register dat moet worden beschreven. Welke waarde moet worden beschreven zit in D_{in}. De decoder zendt dan signalen naar alle afzonderlijke registers, en samen met het AND-en van het “w”-signaal, zal dan in het gepaste register worden geschreven. (**bij dalende klokflank**)



Leespoorten zijn hier aangegeven door een MUX. Het aantal leespoorten kan men dus gemakkelijk uitgebreid worden, ware het niet dat ze registertoegang sterk vertragen. Ook schrijfpoorten kunnen worden uitgebreid, maar dat is al wat complexer omdat je dan ook MUX’en nodig hebt om het gepaste D_{in} te kiezen.

Het datapad

De klok

Geeft het tempo aan waarmee interne en externe controlesignalen worden gegenereerd. Het kloksignaal van de processor wordt in de praktijk afgeleid van het kloksignaal van de **front side bus**. Deze is doorgaans lager dan die van de processor, maar er wordt een frequentievermenigvuldiger opgesmeten. Dit garandeert dat de klok van de processor en de bus aan elkaar gekoppeld blijven.

De processor

De kern van de processor bestaat uit 2 essentiële onderdelen:
- Datapad met registers & ALU
- Controle-eenheid: regelt de aansturing van de controlepunten in het datapad en zet alle bewerkingen in de processor in gang + controleert alle processen.

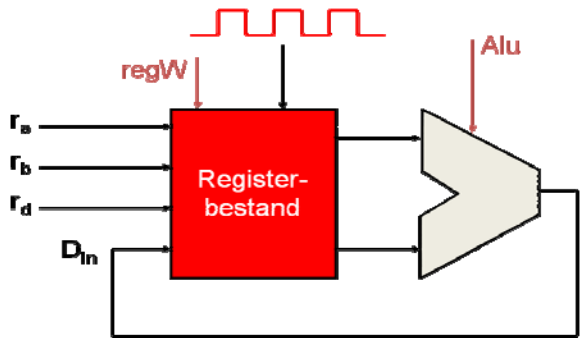
We zullen stap voor stap een datapad opbouwen, en beginnen daarbij bij het inladen van een instructie.

Inladen van een instructie

Per klokcyclus zal deze schakeling een instructie ophalen uit het geheugen. Op de dalende flank van de klok zal de PC een nieuwe waarde krijgen, welke hij meteen naar het geheugen doorstuurt. Van zodra het geheugen het adres binnenkrijgt, zal het de instructie produceren aan de output die op dit adres opgeslagen ligt.
Simultaan wordt de waarde van PC ook naar een ALU gestuurd die altijd een constante 4 erbij optelt. Het resultaat gaat terug naar PC, waar het bij de volgende dalende klokflank de waarde van de vorige PC zal vervangen.

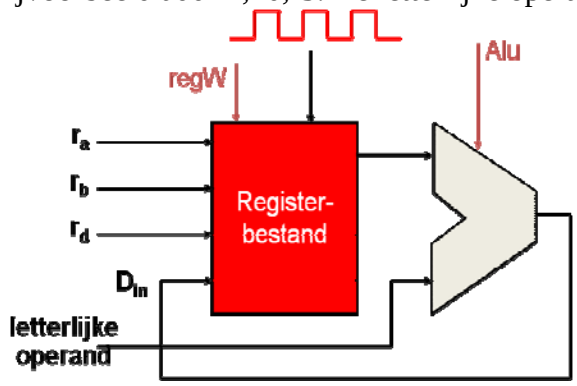
ALU instructie RRR (2 bronnen, 1 doel uit register)

Uit de instructiebits worden de nummers van de 2 bronregisters en het doelregister van de instructie afgezonderd en naar de twee leespoorten, respectievelijk schrijfpoort van het registerbestand gestuurd. Via de leespoorten stelt het registerbestand dan de twee waarden ter beschikking aan een ALU die het resultaat berekent (bvb add r1,r2,r3). De operatie die moet worden uitgevoerd wordt aan de ALU meegedeeld door een aantal operatiebits.
Het resultaat wordt aan de poort D_{in} van het registerbestand aangelegd. Als regW aanstaat en D_{in} is stabiel , zal bij dalende klokflank het resultaat worden weggeschreven.



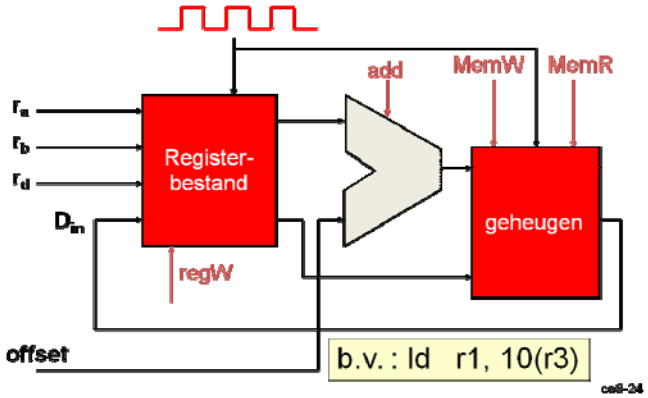
ALU instructie: RRI (1 letterlijke operand)

Bijvoorbeeld add r1,10,r3. De letterlijke operand komt uit de instructie zelf.

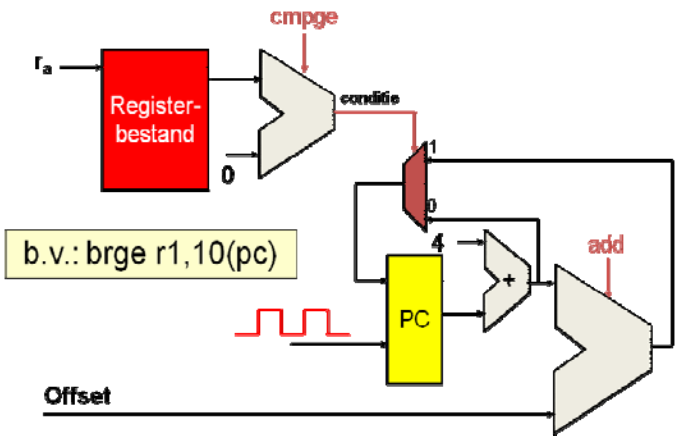


MEM instructie

Hier wordt de ALU gebruikt om een adresberekening te maken. Het resultaat ervan wordt dan aangelegd in het datageheugen. Bij het lezen zal MemR actief zijn, en zal het geheugen na een tijd een waarde produceren die aangelegd wordt aan de D_{in} ingang van het registerbestand. Via de tweede leespoort wordt nu ook (simultaan met de adresberekening) de tweede operand ingelezen, en bij het schrijven aan het datageheugen aangelegd. MemW wordt actief, en de waarde wordt weggeschreven bij dalende klokflank.



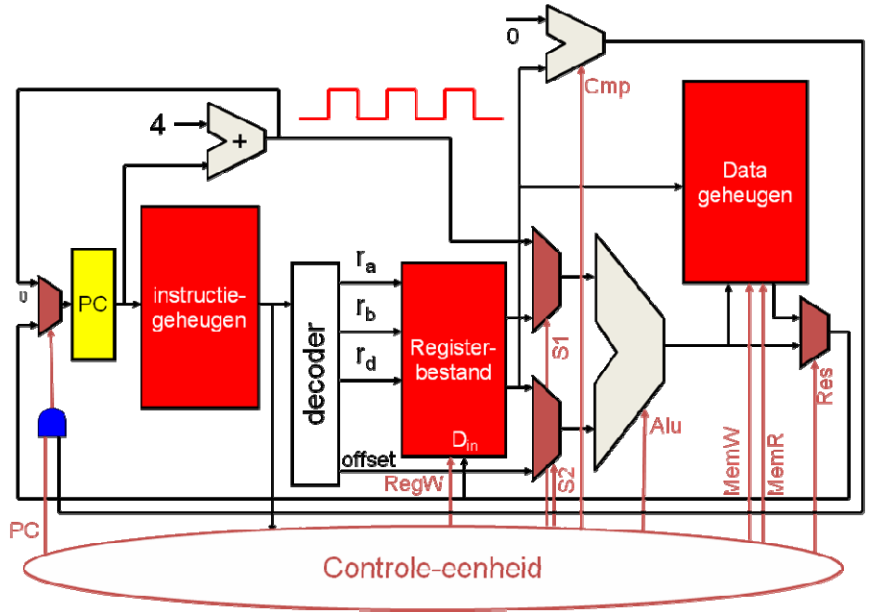
Controletransfer instructie



2 ALU operaties: ééntje om de sprongvoorwaarde vast te stellen, en een andere om het sprongadres vast te stellen. Aangezien de nieuwe waarde van PC nu PC+4 of PC+4+offset kan zijn, moet een MUX worden toegevoegd om een keuze te maken. Als er een 1tje staat aan de MUX, wil het zeggen dat dit wordt gekozen als de voorwaarde opgaat.

Een cyclus per instructie machine

De zonet besproken datapaden kunnen allemaal worden samengevoegd tot 1 groot datapad, dat per cyclus 1 instructie uitvoert. Om dit mogelijk te maken werden er een aantal MUX'en toegevoegd, een decoder om registernummers en constanten uit de instructies te isoleren, en tenslotte ook nog een **controle-eenheid** die de werking van alle ALU's, MUX'en, geheugens en registers controleert. Normaal gezien is 0 de bovenste uitgang bij een MUX.



Hieronder vind je de controletabel om de verschillende controlepunten aan te sturen. Deze tabel kan gemakkelijk worden geïmplementeerd aan de hand van een geheugen met baucellen van (hier althans) 13 bits.

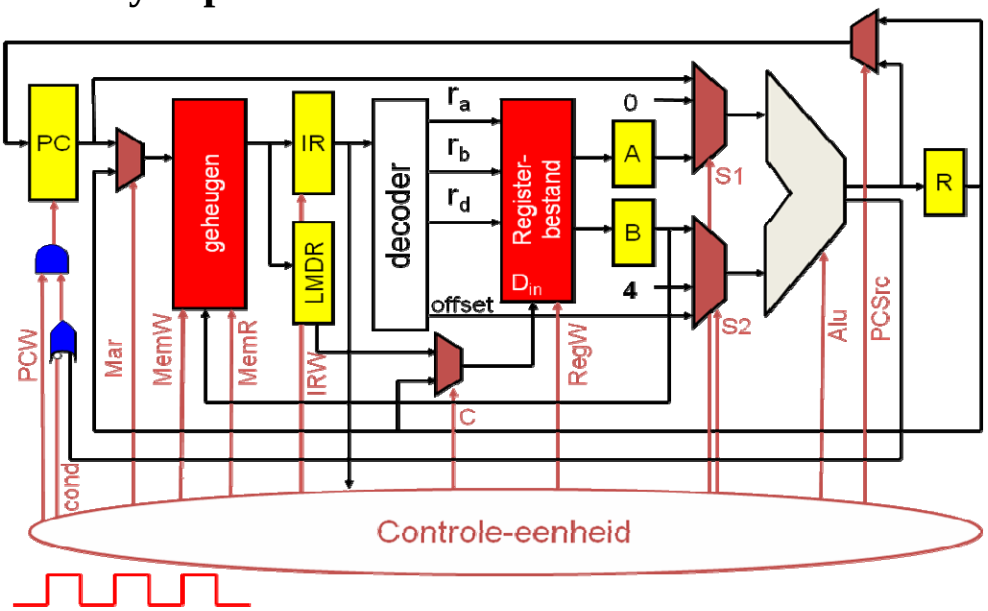
	PC	RegW	S1	S2	Alu	Cmp	MemW	MemR	Res
add	0	1	1	0	001	xxx	0	0	1
addi	0	1	1	1	001	xxx	0	0	1
load	0	1	1	1	001	xxx	0	1	0
store	0	0	1	1	001	xxx	1	0	x
jump	1	0	0	1	001	111	0	0	1
brge	1	0	0	1	001	110	0	0	1

Add 000	0110001xxx0 01
Addi 001	0111001xxx001
Load 010	0111001xxx010
Store 011	0011001xxx10x
Jump 100	1001001111001
Brge 101	1001001110001

2 grote nadelen:

- Sommige onderdelen komen gedupliceerd voor (ALU)
- Alle instructies voeren even snel/traag uit, want de klokperiode kan niet kleiner gemaakt worden dan de tijd die de traagste instructie nodig heeft om uit te voeren.

Meer cycli per instructie machine



Alle onderdelen komen hier nu slechts éénmaal voor, dus sommige zullen meermaals moeten gebruikt worden tijdens uitvoering van 1 instructie.

De eerste stap tijdens de uitvoering is identiek voor alle instructies. Tijdens de dalende flank van de klok wordt PC aangelegd aan het geheugen die een instructie ophaalt en doorspeelt aan zowel IR (instructieregister) als aan LMDR (load memory data register). Hierbij wordt ook PC doorgespeeld naar de ALU en telt er 4 bij op, en stuurt resultaat terug naar PC.
Bij de volgende dalende klokflank zal alles zijn uitgevoerd
Actieve signalen: MemR, IRW, S2, Alu=add, PCSrc, PCW
Niet actieve signalen: Cond, S1, Mar, C, RegW

De tweede stap is ook identiek. De instructie die is ingeladen, wordt nu gecodeerd door de instructiedecoder, die de registernummers en constanten in de instructie eruit haalt, en doorspeelt naar het registerbestand (registernummers, welke waarden leveren aan A en B) en controle-eenheid (controle-informatie ophalen nodig voor de uitvoering van de instructie). De gedecodeerde offset, samen met de aangepaste instructie wordt doorgegeven aan de Alu om te vermijden dat hij niets zou doen. We weten nog niet of er gesprongen dient te worden, dus deze berekening is louter speculatief. Het resultaat wordt aangeboden aan register R.
Bij volgende dalende klokflank zijn al deze operaties uitgevoerd, en waarden in A, B en R opgeslagen. Deze registers zullen immers reageren bij elke klokcyclus, PC en IR doen dit enkel wanneer hun controle-ingang hoog is.
Actieve signalen: S1=1, S2=2, Alu=add

VERVOLG ADD/ADDI
Add Derde Cyclus
Registers A en B worden naar de Alu gestuurd, opgeteld, en in R gestoken.
Actieve signalen: S1=2, S2=0, Alu=add

Addi Derde Cyclus
Enkel register A + offset naar Alu, resultaat in R.
Actieve signalen: S1=2, S2=2, Alu=add

Add/Addi Vierde Cyclus
Via de gepaste controlesignalen wordt de inhoud van R doorgestuurd naar de D_{in}-ingang van het registerbestand. Het signaal r_d bepaalt dan waar de waarde moet worden opgeslagen. Dit stond al een hele tijd klaar, maar wordt nu pas gebruikt doordat RegW=1. Zoals eerder gezegd zijn ook r_a en r_b nog actief, waardoor A en B steeds dezelfde waarde krijgen.
Actieve signalen: RegW=1,C=1

VERVOLG LOAD/STORE
Load/Store Derde Cyclus
Berekening van de adresseermode, hier de som van register+hard gecodeerde offset. Resultaat komt in R.
Actieve signalen: S1=2, S2=2, Alu=add

Store Vierde Cyclus
Inhoud van B (weg te schrijven waarde) wordt naar ingang van het geheugen gebracht. Via Mar=1 wordt het effectief adres aan de i,gang van het geheugen gelegd.
Bij dalende klokflank zorgt MemW = 1 dat de waarde wordt weggeschreven in het geheugen.
Actieve signalen: Mar=1, MemW=1

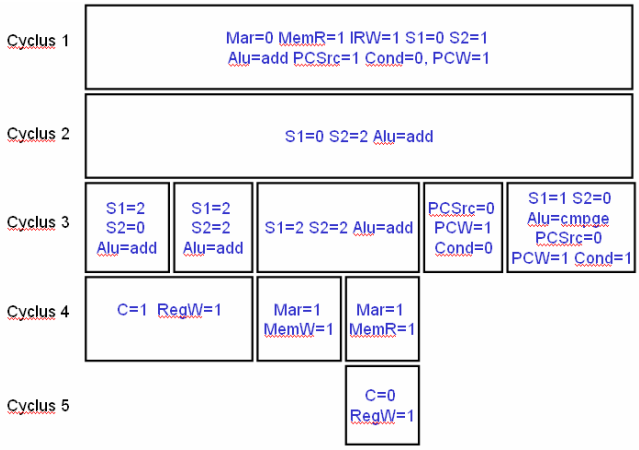
Load Vierde Cyclus
Analoog, maar er wordt niet geschreven, maar gelezen en het resultaat komt in LDMR terecht. Omdat IRW=0 komt het niet in IR terecht! Anders zou er info worden overschreven.
Actieve signalen: MemR=1, Mar=1

Load Vijfde Cyclus
Gelezen waarde wordt weggeschreven in een register.
Actieve signalen: RegW=1, C=0

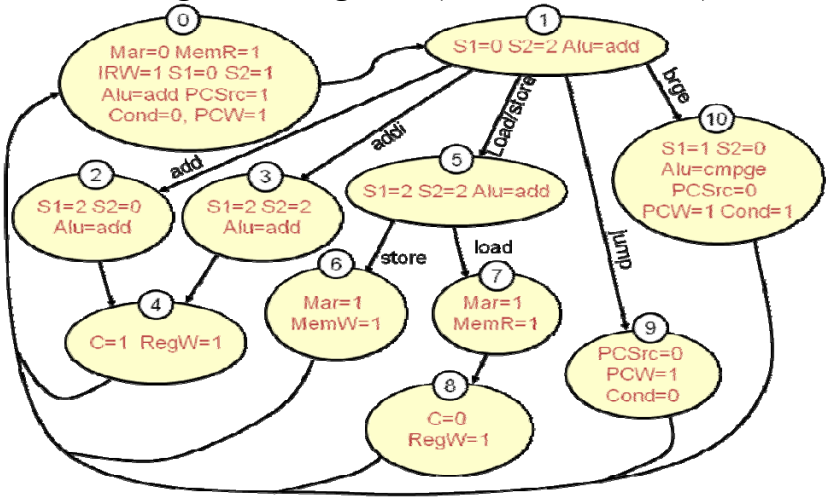
VERVOLG JUMP/BRGE
Jump Derde Cyclus
Het enige wat men hier hoeft te doen is de berekende speculatieve waarde doorsturen naar PC.
Actieve signalen: PCScr=0, PCW=1, Cond=0

Brge Derde Cyclus
Alu wordt gebruikt om te vergelijken met 0, en de resulterende conditiecode wordt gebruikt om het schrijfsignaal van PC te sturen. Cond wordt normaal op 0 gelaten, tot het verandert op 1, en pas dan zal de AND poort het schrijfsignaal naar de PC sturen.
Actieve signalen: PCSrc = 0, PCW=1, Cond=1, Alu=cmpge, S1=1, S2=0

Samenvatting controlesignalen



Samenvatting controlesignalen (toestandsautomaat)



De labels op de pijlen geven aan welke voorwaarden er moeten voldaan zijn om een bepaalde overgang te maken.

Naast de controlesignalen per toestand moet er ook informatie worden bijgehouden over het verloop van de controle doorheen de automaat: de overgangen tussen de verschillende toestanden. Onderstaande tabel geeft aan op welke voorwaarde er van één toestand naar een volgende kan gesprongen worden. In 8 bits zal dus begintoestand en operatie kunnen worden gecodeerd, indien we ervan uitgaan dat de operaties ook in 4 bits kunnen worden gecodeerd. Uitgaande van deze 8 bits kan men dan ondubbelzinnig de volgende toestand en alle controlesignalen afleiden. De generatie van al deze signalen gebeurt aan de hand van een tabel.

toestand	voorwaarde	volgende toestand
0 0 0 0		0 0 0 1
0 0 0 1	op = add	0 0 1 0
0 0 0 1	op = addi	0 0 1 1
0 0 0 1	op = load/store	0 1 0 1
0 0 0 1	op = jump	1 0 0 1
0 0 0 1	op = brge	1 0 1 0
0 0 1 0		0 1 0 0
0 0 1 1		0 1 0 0
0 1 0 1	op = store	0 1 1 0
0 1 0 1	op = load	0 1 1 1
0 1 0 0		0 0 0 0
0 1 1 1		1 0 0 0
1 0 0 0		0 0 0 0
1 0 0 1		0 0 0 0
1 0 1 0		0 0 0 0

ROM implementatie

Indien de tabel in Read-Only Memory wordt opgeslagen, zijn er lichte wijzigingen. De controlesignalen hangen enkel af van de toestand, niet van de operaties, dus beschouwen we ze beter afzonderlijk. Er zijn dan 16 mogelijke toestanden (11 daadwerkelijk gebruikt), en in elke toestand moeten 16 signalen worden gegenereerd → 256 mogelijke gevallen, die elk een nieuwe toestand van 4 bits moeten genereren.
Alle don't cares in bvb ROM1 worden vervangen door 0'en. De lijnen 12-16 worden nagenoeg niet gebruikt! (dus allemaal 0)

Microcodering

Indien we gebruik maken van een sequencer, proberen we ervoor te zorgen dat daar, waar er geen keuzen worden gemaakt, de toestanden sequentieel oplopen. Het volstaat dan om een opteller met 1 te definiëren om te gaan naar de volgende toestand. Twee gevallen waar dit niet mogelijk is: **terugspringen naar 0 en waarbij er meer dan 1 opvolger is**.

→ Het terugspringen naar 0 kan worden opgelost door een reset-signaal te genereren.
→ Meer dan 1 opvolger: gebruik maken van sprongtabellen die uitgaande van de operatiebits de nieuwe toestand gaan bepalen.

De keuze tussen terugspringen naar 0, kiezen uit meer dan 1 opvolger, en de gewone sequentie moet worden gemaakt door een MUX aan het begin van de ROM. Hiervoor dienen echter 2 nieuwe controlebits worden gezonden naar de MUX, welke de aard van de operatie aanduiden. Hiervoor is het nodig dat ADD en ADDI worden opgesplitst.

Indien we nu niet willen dat ADD en ADDI worden opgesplitst, en indien we niet willen gebonden zijn aan de sequentiële nummering van de toestanden, dan kunnen we de **volgende toestand ook opnemen in de controlesignalen**.
In plaats van de toestand nu op 0 te dwingen, kunnen we hem eender welke waarde geven, welke gecodeerd staat in ROM1. **(escape-simulator model!)**
De ROM1 tabel wordt dan het microcodeprogramma genoemd, en de toestandsveranderlijke de microcodePC. De controle-eenheid voert dan het microcodeprogramma uit.

RISC vs. CISC

Evolutie van de instructiesets

Eerst was er in de jaren 70 de CISC: complex instruction set computer, welke werkte met complexe microgecodeerde instructiesets. Vervolgens in de jaren 80 de RISC: reduced instruction set computer.
De RISC herkent men aan volgende kenmerken:

- Voornamelijk “echt nodige”, effectief gebruikte, eenvoudig decodeerbare instructies
- Groot aantal registers, RRR machinemodel
- Geen complexe adresseermodes

Instructiegebruik & Complexiteit

Een groot deel van de uitgevoerde instructies zijn héél eenvoudige instructies, waardoor de complexe CISC dus vervangen werd door een gereduceerde instructieset computer, de RISC. Ook is voor de meeste instructies de complexiteit niet erg groot; voor toewijzingen is 80% van de eenvoudigste vorm met 2 parameters: X=Y;

Strafverhouding

= een maat voor hoeveel beter een assemblerprogrammeur een bepaald probleem kan oplossen. In de CISC tijd was deze verhouding zeer gunstig, want:

- Optimaliserende compilers waren een uitzondering
- Programmeurs konden met succes gebruik maken van de particulariteiten van de processor, terwijl de compiler dit niet kon.

Bij de RISC viel het tweede weg, en door de regelmatige structuur in de instructieset, kon ook de compiler verder gaan optimaliseren. Het loont dus tegenwoordig niet langer om in assembler te programmeren, uitgezonderd de binnenste kern van spelletjes bijvoorbeeld, waar iedere extra cyclus telt.
De strafverhouding = **uitvoeringstijd machinetaal / uitvoeringstijd hoge-niveautaal**

SPARC registervensters

SPARC werkt met een zéér groot aantal registers. De processors met veel registers zorgen ervoor dat er bvb maar 32 per keer simultaan zichtbaar zijn, omdat veel registers niet echt goed is voor systeemsoftware, die bij een onderbreking deze in een register moet bewaren.

SPARC werkt met register**vensters** van 32 registers per keer:

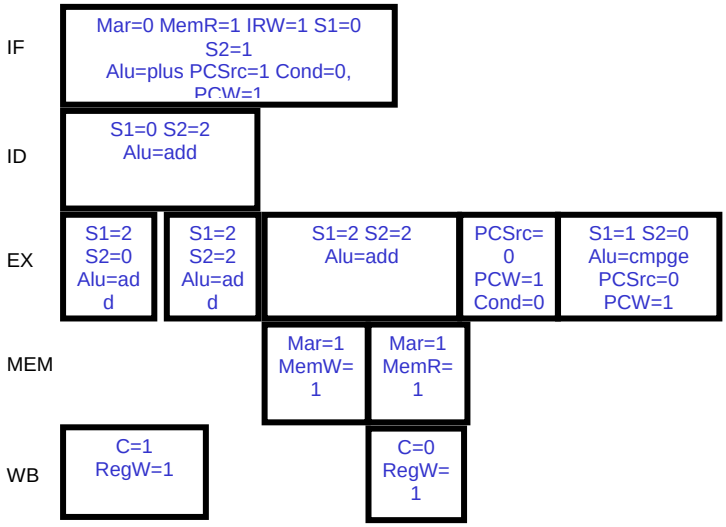
- 8 globale
- 8 lokale
- 8 input (parameters voor de huidige functie)
- 8 output (parameterdoorgave)

Per procedureoproep worden 16 registers bewaard (lokale), en komen er 16 andere in de plaats (output). Tegelijk worden de outputregisters van niveau n, inputregisters van niveau n+1.

Sequentiële gepijplijnde machine

Samenvatting controlesignalen

Als we de controlesignalen van vorig hoofdstuk nemen, kunnen we per cyclus een naam geven aan de stap. We plaatsen hiervoor wel de registerbeschrijving RegW van cyclus4 in cyclus 5. M.a.w. de WB voor ALU instructies wordt slechts uitgevoerd in de vijfde cyclus!

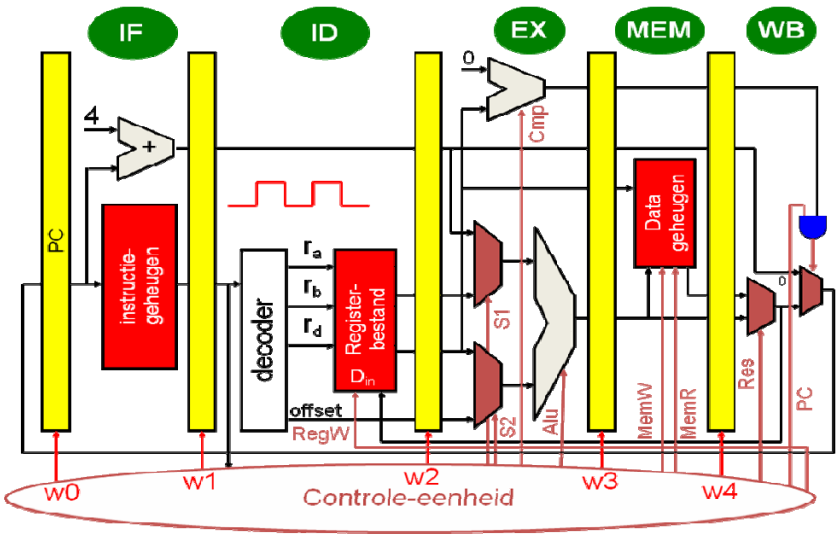


IF = Instruction Fetch, haalt de instructie op en verhoogt PC
ID = Instruction Decode, decodeert de instructie, haalt registerwaarden op en berekent speculatief het sprongadres.
EX = Execute, voert de semantiek van de instructie uit (betrekking van de ALU)
MEM = Memory, interactie met het datageheugen
WB = Write Back, resultaten wegschrijven in de gepaste registers

Indien we bij de meer-cycli-per-instructiemachine er willen voor zorgen dat iedere instructie in 5 trappen wordt uitgevoerd, zullen we er voor moeten zorgen dat:

- bij overgang van 1 cyclus naar de volgende, alle waarden die een overgang moeten overleven, ergens tijdelijk kunnen worden opgeslagen in een register (A,B,LDMR,IR,PC,R)
- moeten we extra registers bijplaatsen op het eind van elke trap.

Omdat we weten dat de bestemming van een sprong pas bekend is na de EX trap, of zelfs na de MEM trap (indien geheugenindirectie), wordt de berekening van de PC volledig verhuisd naar de WB trap!



Deze opstelling wordt een **pijplijn** genoemd. De instructies vertrekken aan de linkerzijde in de pijplijn, en verplaatsen zich naar rechts in een vast tempo. De registers tussen 2 pijplijntrappen, worden **pijplijnregisters** genoemd, aangestuurd door extra controlesignalen w0-w4, welke zullen bepalen wanneer een pijplijnregister waarden zal opnemen en doorgeven aan de volgende trap. Dus er wordt een 1tje gezonden naar w1 in de IF trap, w2 in de ID trap, w3 in de EX trap, w4 in de MEM trap, en w0 in de WB trap.

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

Aan de controletabel verandert ook iets! Het is nu niet meer nodig om de signalen gedurende de ganse uitvoering actief te houden. Er is toch maar 1 trap per keer actief, en tijdens die trap moeten de controlesignalen die bij die trap horen, geactiveerd worden.

Het **RegW** signaal wordt pas aangestuurd in de WB trap, ofschoon het een blok in de ID trap aanstuurt! Dit komt omdat het registerbestand 2 maal wordt gebruikt: éénmaal in de ID, en éénmaal in de WB trap.
De **IF en ID** trap staan hier niet in vermeld, omdat deze toch onafhankelijk van de instructie verlopen, en de controle kan dus vast zijn.

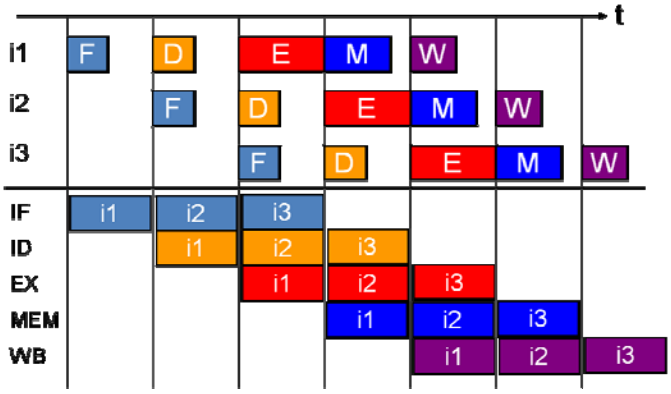
We kunnen de sturing van de controlesignalen ongelooflijk vereenvoudigen, indien we de signalen laten genereren in de ID trap, en de signalen via de pijplijnregisters doorgeven naar de juiste locatie. Merk op dat we het **RegW** signaal artificieel vertragen tot in de WB trap, om dan terug te sturen naar het registerbestand. We zullen hierbij ook het r_d signaal vertragen van aan de decoder tot in de WB trap, zodat het nergens moet worden bewaard tot aan de WB trap.

Voor het volgen van de verschillende ALU instructies, zie slide 9-15

De langst durende trap bij een sequentiële pijplijncyclus zal bepalen hoe groot een klokperiode mimimaal moet zijn. Een **volgende instructie** wordt pas aangevat **wanneer de vorige instructie volledig is verwerkt**. Indien we naar de individuele trappen van de pijplijn kijken, dan zien we dat een trap gemiddeld gezien maar om de 5 cycli eens gebruikt wordt, en ongebruikt blijft tijdens de overige cycli.

Parallelle gepijplijnde machine

We kunnen vermijden dat de pijplijntrappen ongebruikt blijven tijdens die overige cycli door de verschillende instructies na elkaar te verwerken. Dit werkt dankzij het feit dat in een pijplijn slechts 1 trap per keer actief is. Bovendien werkt dit enkel voor instructies die geen controletransfer instructies zijn, want de nieuwe pc wordt pas in EX (of MEM) bekend. Dit kan worden opgelost door **4 NOP instructies** te plaatsen na de controletransfer. (zie later)



Kenmerken van parallelle pijplijncyclus:

- **PC + 4** wordt rechtstreeks doorgestuurd naar de WB trap, zodat in een volgende cyclus onmiddellijk een nieuwe instructie kan worden opgehaald.
- Hier bespreken we slechts een model met 5 trappen, maar er kunnen er veel meer zijn. We kunnen door deeltrappen te beschouwen, de maximale uitvoeringstijd per trap verder reduceren, en dus de klokfrequentie op te drijven.
- De instructies voeren **niet sneller uit**! Per instructie blijft de uitvoeringstijd 5 cycli! Men spreekt van **CPI = clocks per instruction** en **IPC = instructions per clockcycle**

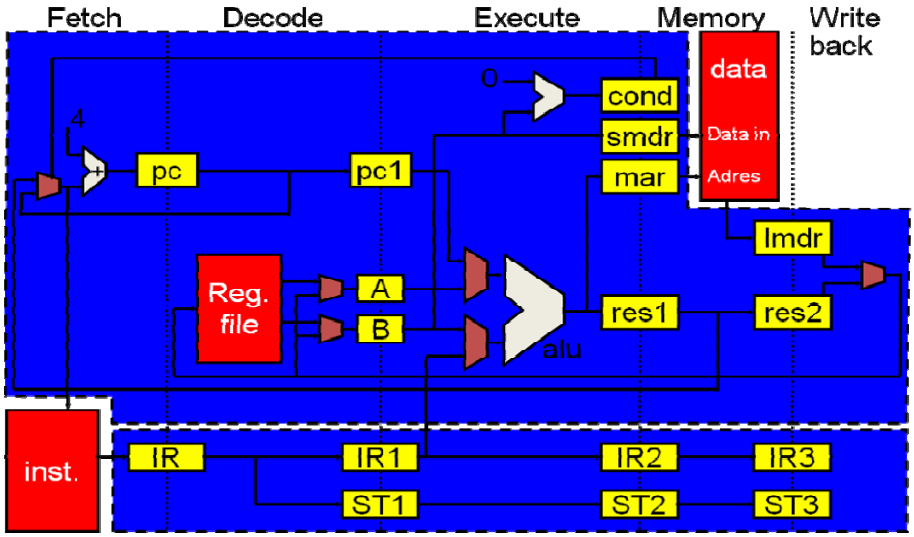
Aanpassingen:

- Eigenlijk maakt de berekening van de nieuwe pc enkel gebruik van informatie die in de EX trap wordt gegenereerd. De selectie van de PC kan nu worden verplaatst naar de EX trap, en meteen worden opgeslagen in het PC register. Nu moeten we geen 5 cycli meer wachten op de PC, en zijn er maar **2 NOP's** meer nodig bij controletransfers.

Het model van de Escape Simulator

Fetch: instructie ophalen & PC berekenen.
Decode: registernummers uit instructie gehaald, en juiste registers bepaald.
Execute: sprongadres / adresseermode / resultaat van de ALU operatie berekenen. Sprongadressen worden dus na 3 cycli al teruggevoerd naar PC.
Memory: interactie met het datageheugen
Writeback: resultaat naar register geschreven. Dit resultaat kan tegelijk in het registerbestand, en in registers A en B worden geschreven → registers kunnen in dezelfde cyclus een waarde opnemen, en in diezelfde cyclus (aan het eind) diezelfde waarde beschikbaar stellen. Dit is mogelijk door de klok van het registerbestand wat te vervroegen, zodat de op te slane waarde opgeslagen wordt en al zichtbaar wordt aan de uitgang, net voor het invullen van A en B. (cfr. de extra MUX)

Het bovenste blauwe vierkant is het datapad, het onderste het controlepad.



Hazards

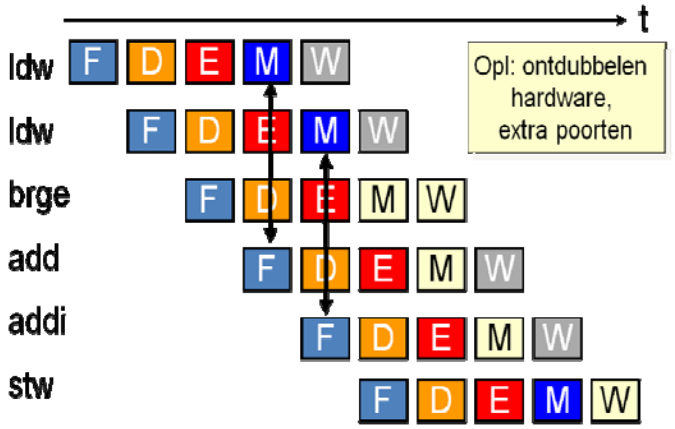
Een hazard is een afwijking van de standaard sequentiële interpretatie van de programmacode. Ze worden veroorzaakt door de overlappende uitvoering van instructies die na elkaar komen.

Structurele hazards

Oorzaak

- Schaarse hardware: bijvoorbeeld wanneer registerbestand niet simultaan zou kunnen lezen en schrijven, dan ontstaat een structurele hazard tussen WB en ID.
- Voornamelijk een probleem in micro architecturen waar niet alle EX trappen even lang duren, en parallel kunnen uitvoeren. Hierbij is het dus niet statisch te voorspellen wanneer een bepaalde trap een bepaald onderdeel van de processor zal nodig hebben.

Situatie



Hier wordt natuurlijk gesteld dat er slechts 1 geheugen is, en dat het geheugen maar 1 aanvraag per keer kan bedienen.

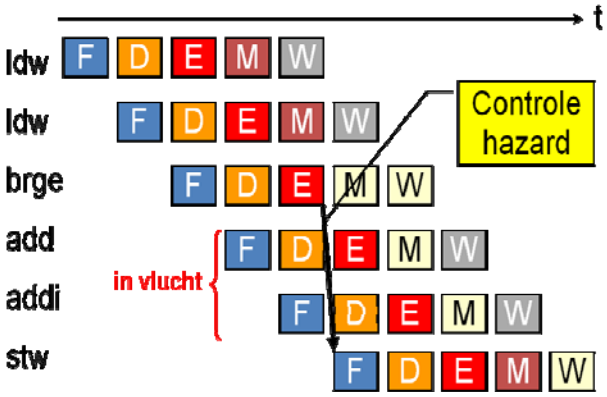
Oplossing
Onderdelen van de processor ontdubbelen, zodat 2 simultane aanvragen wel gewoon door kunnen gaan.

Controlehazards

Oorzaak:

- Vertraging bij berekenen van een sprong

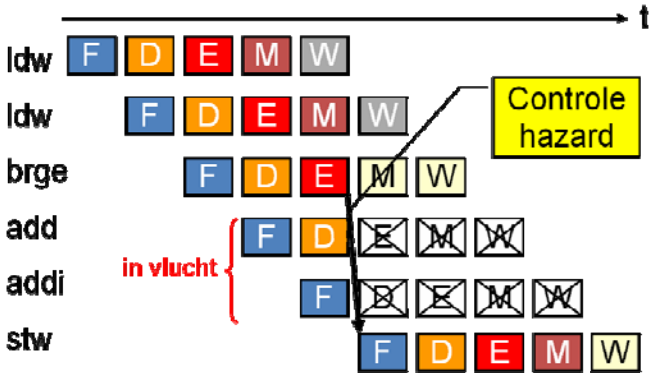
Situatie



Eigenlijk mogen add en addi hier niet worden uitgevoerd, maar dat weet de processor niet, want pas op het einde van de EX trap van het brge commando weet men pas dat er degelijk moet worden gesprongen. Instructies die aan hun uitvoering begonnen zijn noemen we **in vlucht**.

Oplossingen

- **Vervangen door NOP**: instructies die onterecht in uitvoering zijn, kunnen worden vervangen door NOP instructies, ze worden m.a.w. omgezet in **pijplijnbellen**. Dit is net op tijd vooraleer een register of geheugen wordt veranderd van toestand. In de Fetch en Decode trap gebeuren geen wijzigingen van de processortoestand, dus deze kunnen blijven bestaan. Het creëren ervan is eenvoudig. Het signaal welke aangeeft dat er moet gesprongen worden, zal de Fetch en Decode trap van de instructie in de **pijplijnregisters** annuleren door ze bvb op 0 te zetten. De reeds ingelezen en gedecodeerde instructies, zullen dus worden vervangen door NOP's, die ongestuurd verder door de pijplijn kunnen gaan.



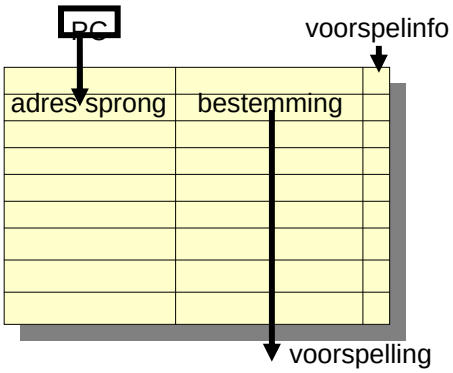
- **Vertraagde controletransfer/sprong**: het probleem wordt verschoven naar de programmeur door aan te duiden dat 1 of 2 instructies na de sprong wel degelijk worden uitgevoerd, zodat men in de processor geen zorgen moet maken of controlehazards. De programmeur zal dus deze **instructieslots / delay slots** moeten invullen met 2 bewerkingen die sowieso mogen uitgevoerd worden. Als men geen gepaste bewerkingen vindt, kan men nog altijd NOP invoegen.

  We kunnen dit op een aantal manieren doen.
  Bij een **onvoorwaardelijke sprong** kun je de instructies die net voor de spronginstructie komen, of van op de bestemming van de sprong erin plaatsen.
  Bij een **voorwaardelijke sprong** mag je meestal geen instructies nemen van net voor de sprong, want ze helpen meestal mee de sprongconditie vast te leggen. Op de sprongbestemming dienen de instructies ook slechts ENKEL uitgevoerd te worden ALS de sprong wordt genomen. De SPARC architectuur biedt een oplossing: men schrijft een instructie van de bestemming in het instructieslot, maar bij de vertraagde sprong (bvb **breq next**) zet men een annulatiebit **breq, a next** zodat men een cyclus wint indien de sprong genomen wordt!

  Men poogt omwille van het laag houden van het aantal delay slots om de selectie van de volgende PC reeds in de ID-trap te laten gebeuren. We moeten in de architectuur dan op een aantal zaken letten wanneer we de twee ALU's betrokken bij berekenen van de nieuwe PC willen verplaatsen naar de ID stap:
  1. De optellingsALU geeft geen probleem, daar de offset reeds kort na het begin van de ID cyclus bekend is, en de PC bij het begin.
  2. De comparator is iets moeilijker omdat hij moet wachten op het resultaat van het registerbestand (B). Comparator kan alleen worden verplaatst naar de ID als de vergelijking eenvoudig is, en dus de waarde nog op tijd binnen dezelfde cyclus naar de MUX te sturen voor de PC.

- Soms lukt het dus niet om de **comparator** naar de ID trap te verplaatsen, en dan maakt men gebruik van **sprongvoorspellers**.

  Een sprongvoorspeller zal proberen, uitgaande van het voorbije gedrag van een programma, te voorspellen wat de volgende in te laden instructie zal zijn (op een precisie van 95%). Ze zal in de IF fase voorspellen of een sprong al dan niet genomen wordt, en het adres zal worden berekend in de ID trap. In sommige gevallen levert ons dit geen tijdswinst bij een vijftrapspijplijn, want de uitkomst kan soms worden berekend in 2 trappen. Bij een complexe vergelijking, levert ons dit wel winst op, omdat de sprong anders slechts in de 5^{de} trap is bekend.
  Het berekenen van een adres kan ook in de IF trap. Hiervoor maken we gebruik van een **Branch Target Buffer**. Hiervoor dienen we naast info over het al dan niet nemen van een sprong (zie later) ook info bijhouden over de bestemming van een sprong. Dit neemt wel heel wat meer plaats in beslag per entry dan een klassieke voorspeller.



1. Statische sprongvoorspellers

1/ Voorspel niet-genomen

Een voorwaardelijke sprong wordt in een lus meestal in het begin geplaatst zodat de voorspelling in de meeste gevallen juist is, want de instructie na een voorwaardelijke sprong wordt meestal altijd ingeladen, en de sprong wordt dus voorspeld als niet genomen.
Een onvoorwaardelijke sprong wordt als genomen voorspeld.

2/ Branch backward taken/ forward not taken

Voorspelling wordt hier gemaakt in de richting waarin de sprong gaat. In dit geval wordt de voorwaardelijke sprong beter achteraan geplaatst in de lus.

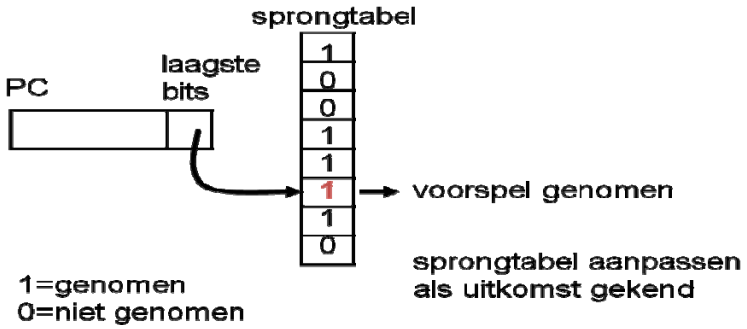
2. Dynamische sprongvoorspellers

1/ Eenvoudige dynamische voorspeller

Maakt een voorspelling op basis van de inhoud van een sprongtabel. De tabel houdt hier bij of de sprong op een gegeven adres de vorige keer al dan niet genomen werd. Bij elke foute voorspelling wordt de waarde in de tabel aangepast.

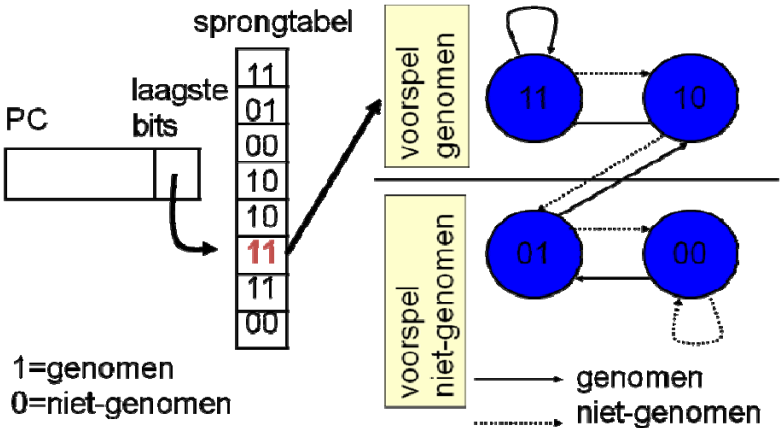
Probleempjes:

- Bij het verlaten van de lus, en bij het eerste keer heruitvoeren van de lus zal er verkeerd voorspeld worden. Indien de lus 10 keer wordt genomen, wordt dus 20% verkeerd voorspeld.
- De sprongtabel is véél kleiner dan het aantal adressen → waarden worden overschreven = **aliasing of verwarring**



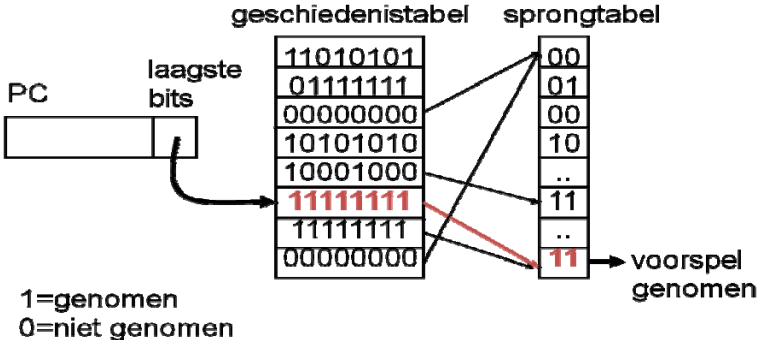
2/ 2-bit dynamische voorspeller

Er worden 2 bits bijgehouden in de sprongtabel. We gebruiken hier een **saturerende 2 bit opteller**.

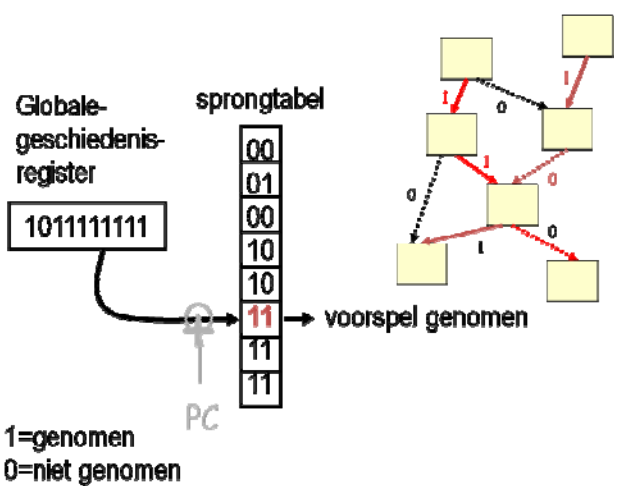


3/ Lokale dynamische voorspeller

We gaan nu de sprong voorspellen aan de hand van de geschiedenis van de sprongen. (hier bvb 8 bits) Per uitgevoerde sprong wordt in de geschiedenistabel een 1 ingeschoven. Op basis van dit patroon wordt er dan gekeken in de sprongtabel wat er bij dergelijk patroon gebeuren moet.



4/ Globale dynamische voorspeller



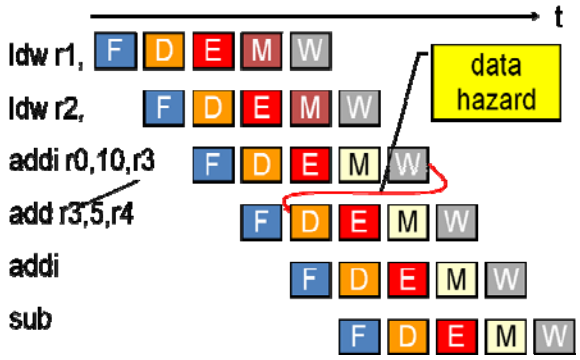
Op dezelfde manier kan er ook vertrokken worden van een globale geschiedenis zoals de processor ze aan zich ziet voltrekken. Per sprong die wordt ontmoet, wordt een bit toegevoegd aan het globale geschiedenisregister, welke dan wordt gebruikt om een waarde uit de sprongtabel aan te duiden.
Probleem: om te vermijden dat twee globale geschiedenissen (afkomstig van 2 verschillende plaatsen in het programma) dezelfde voorspelling moeten opleveren, zullen we ze XOR'en met een aantal bits van PC, en zo geraken de indices in de sprongtabel ook meer verstrooid.

Datahazards

Oorzaak:

- Resultaten worden pas in de vijfde cyclus bewaard, maar er zijn soms instructies die die resultaten al vroeger nodig hebben

Situatie

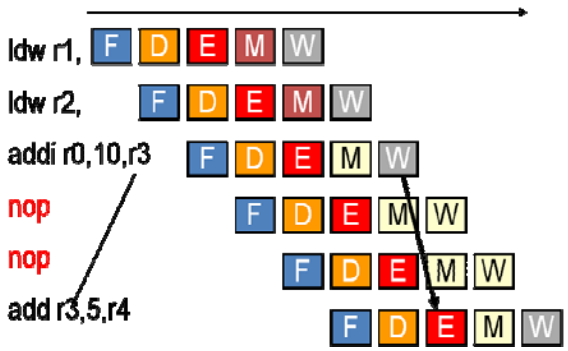


Wanneer twee nabije instructies hetzelfde object lezen of schrijven, ontstaat een afhankelijkheid.

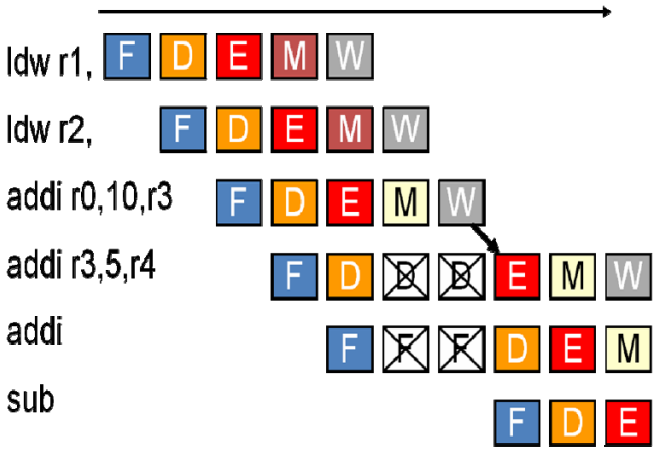
- **RAW**: object wordt eerst geschreven, dan gelezen (echte afhankelijkheid). Lezen moet wachten op schrijfoperatie
- **WAR**: object wordt eerst gelezen, dan geschreven (anti-afhankelijkheid). Schrijfoperatie moet wachten op leesoperatie, maar kan zijn waarde ook elders wegschrijven en op deze manier het wachten vermijden
- **WAW**: object wordt 2 maal geschreven (anti-afhankelijkheid). Tweede schrijfoperatie mag niet vóór de eerste gebeuren maar kan zijn waarde ook elders wegschrijven en op deze manier het wachten vermijden.
- **RAR**: object wordt 2 maal gelezen (**geen** afhankelijkheid). Leesoperaties mogen in een willekeurige volgorde worden uitgevoerd!

Oplossingen

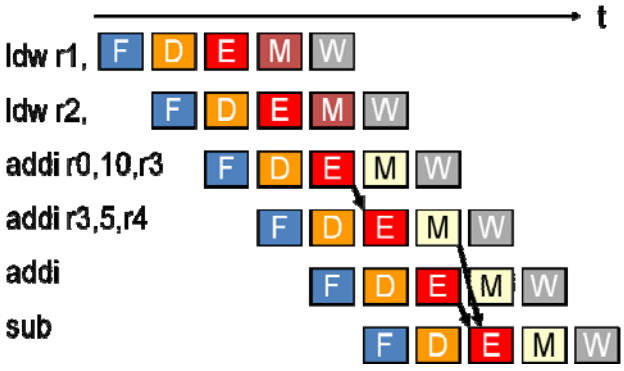
- Afhankelijke instructies ver genoeg uit elkaar zetten door invoegen van NOP.



- Pijplijn blokkeren tijdens de uitvoering telkens zich een datahazard voordoet (**stalls**). De volgende instructies worden onderbroken tot het resultaat beschikbaar is.



- Snelste oplossing: **forwarding**. Van zodra een resultaat bestaat op de processor zal ze beschikbaar zijn voor de volgende instructies. Forwarding kan gebeuren vanuit de output van de EX trap of vanuit de output van de MEM trap.



Om forwarding te realiseren, moeten een aantal extra verbindingen worden gelegd. De uitgang van de ALU na S1 en S2, wordt teruggekoppeld naar de ingang van S1 en S2; dus aangeboden aan dezelfde ALU. Ook wordt de uitgang van de MEM trap terug aangeboden aan diezelfde ALU (terugkoppeling van 2 van WB naar EX).

Principes van een in-order gepijplijnde processor

- IPC = 1 (eigenlijk een stuk onder 1 door hazards), CPI = 5
- 5 instructies tegelijkertijd in uitvoering
- Klokfrequentie hangt af van de **trapvertraging**: tragere trappen kan je opsplitsen in meerdere kleine, maar snellere delen, en aldus een hogere klokfrequentie maken.
- Indien men IPC > 1 wenst zal men moeten toelaten dat:
  - Instructies in een andere volgorde worden uitgevoerd dan deze van het prog
  - Per cyclus meer dan 1 instructie opgehaald & uitgevoerd wordt.

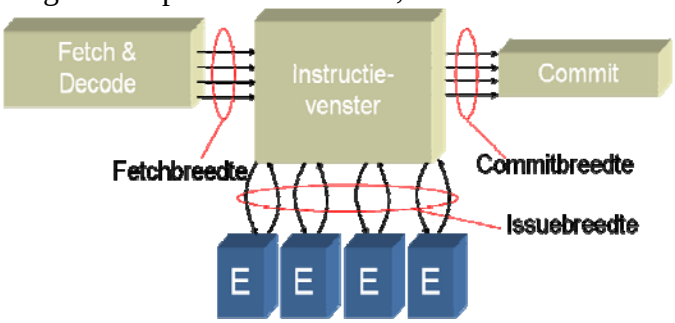
Andere microarchitecturen

Superscalaire architecturen

Zal meer dan 1 instructie per cyclus ophalen en uitvoeren → IPC > 1.

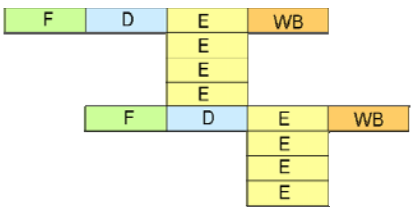
- **Ophalen:** Stel bvb 2 instructies ophalen per cyclus, dan moet geen 32 bit, maar 64 bit in 1 keer worden ingelezen en verwerkt.
- **Uitvoeren**: de trappen moeten nu meer dan 1 instructie per cyclus uitvoeren. De sequentiële semantiek mag hierbij niet verloren gaan! We zullen moeten rekening houden met **echte afhankelijkheden**, anti-afhankelijkheden zullen weggewerkt worden door het resultaat van die instructies tijdelijk op een andere plaats bij te houden en pas weg te schrijven in het registerbestand als alle voorgaande instructies zijn afgerond.

Er zullen – bij superscalaire architecturen – asynchroon instructies worden opgeladen, gedecodeerd en opgeslagen in een **instructievenster**. Hier wachten de instructies op beschikbaarheid van al hun argumenten, om dan uitgevoerd te worden op een van de functionele eenheden (E). Na uitvoering verhuizen ze naar de **commit trap**, waar ze terug in volgorde de processor verlaten, na hun waarde te hebben geproduceerd in registers & caches.



VLIW : Very Long Instruction Word Processors

Een andere manier om van de functionele eenheden nuttig gebruik te maken is om de hulp van de **compiler** in te roepen. Een instructie bestaat dan uit een aantal operaties die simultaan moeten kunnen worden uitgevoerd. De processor leest dan een instructie in, en stuurt de operaties in volgorde naar de betrokken functionele eenheid. De processor zal stellen dat de compiler heeft nagegaan op welk ogenblik de resultaten van de vorige instructie beschikbaar zullen komen in de registers, en neemt gewoon de waarde die op dat moment in de registers zitten. (**Philips Trimedia)**



EPIC : Explicitly Parallel Instruction Computing

Een EPIC instructie bestaat uit 2 operaties, waarbij het onderlinge parallellisme aangegeven wordt in een template, welke kan aangeven welke instructies parallel mogen worden uitgevoerd, en welke niet.

- Heel wat minder NOP nodig!
- Versneld de uitvoering door bvb een zeer groot aantal registers.