Digitale electronica en processoren

Hfdst 2) Digitaal ontwerp

Digitaal ontwerp in grote lijnen

Wanneer we een **digitaal ontwerp** maken, doorlopen we een **aantal stappen en niveaus** van **abstractie** om tot het eindproduct te komen. Dit proces is afhankelijk van

- kennis en de mogelijkheden van het bedrijf dat de chip maakt,
- de mensen die het ontwerp moeten maken
- waarvoor de chip moet dienen
- hoe snel die af moet zijn enzo

⇨ **Documenteren** : Van elk systeem dat we maken moeten we uiteindelijk opschrijven wat het **doet**, hoe het intern **werkt** en hoe het **bediend** moet worden. We kunnen achteraf niet in de chip te kijken.

- voor de **gebruiker**, om het digitale systeem te kunnen gebruiken
- voor de **hersteller**, omdat die moet weten hoe iets ineen steekt om het te kunnen herstellen
- om het ding verder te kunnen **ontwikkelen** en verbeteren

De ontwerpcyclus zelf : Specificatie ⇨ Synthese ⇨ Analyse

1) Specificatie : een beschrijving van de **functionaliteit** van het digitale systeem (wat het doet)
Daarbij hoort de **interface** : de interactie tussen het systeem en de buitenwereld (bediening & control)
⇨ Eerste stap is een **beschrijving** in een natuurlijke taal die ruwweg zegt wat het ding moet doen :

- ⇨ dit impliceert een **blokschema** (flow-chart) waarbij elke blok een eigen functionaliteit heeft met in- en uitgangen en waarbij we logische interconnecties hebben tussen de blokken. (= soort van weg met ja-nee-vraagjes)
- ⇨ De specificaties in een natuurlijke taal zijn dikwijls zeer **onvolledig** : taal is dubbelzinnig en soms onduidelijk. Daarom wordt de specificatie vaak later **aangevuld** en **gewijzigd** tijdens het ontwerpproces wanneer men op dubbelzinnigheden of onduidelijkheden stuit.
- ⇨ Men moet oppassen dat men **niet** reeds vormen van **implementatie of realisatie** beschrijft, omdat de **onnodige beperkingen** oplegt : Specificatie mogen alleen zeggen **wat** het ding moet doen, **niet hoe** het dat moet doen !

2) Synthese : we gaan over van het niveau specificatie, dat zegt **wat** het systeem moet doen, naar een basic beschrijving over **hoe** we dat gaan doen en **waarmee**.
We zullen beschrijven welke **fysische middelen** we zullen gebruiken en hoe we de functionaliteiten zullen **organiseren**.
⇨ We zullen dit doen **in stappen** : telkens naar een niveau dat lager ligt (meer specifiek). We blijven elk component opsplitsen tot alle component voorkomen als basic-bouwblokken in de bibliotheek.

- SysteemSynthese : onderverdelen in **blokken** met eigen functionaliteit en de **interconnectie** en interactie tussen de blokken. (geheugens, processors, ASIC's)
- ArchitectuurSynthese : elke blok heeft een eigen algoritme of flowchart en worden opgebouwd uit een aantal componenten op RTL-niveau. (opteller, teller, registers, optellers)
- sequentieel ontwerp : Finite-state-machine bschrijving omzetten in poorten en flipflops
- combinatorisch ontwerp : booleaanse uitdrukkingen maken met poorten
- circuit-ontwerp : poorten maken met transistoren
- fysisch ontwerp : transistors maken met halfgeleideropppervlakken

Bibliotheek van componenten : **verzameling** ontwerpen van **basis-functionaliteiten** die op een chip gemaakt kunnen worden. Bedoeling is om eenvoudige basiscomponenten **niet steeds opnieuw** te moeten ontwerpen. (economisch) We **herbruiken** dus ontwerpen van eerder ontworpen functionaliteiten en combineren die tot een groter ontwerp. Ontbrekende functionaliteiten kunnen we dan zelf maken of kopen.

⇨ bibliotheken bevatten ontwerpen op **verschillende niveau's** :
- processoren, meer specifieke processoren
- optellers, tellers : meest optimale combinaties van registers en poorten
- logische poorten : manieren om een logische poort te maken met transistors
- transistor

We zien dat de **niveau's** van de bibliotheken steeds **hoger** worden omdat de schaal steeds groter wordt (integratie van grotere delen).

⇨ De ontwerpen in de bibliotheken moeten zeer **goed gedocumenteerd** zijn. Zo weten ontwerpers direct hoe ze het ontwerp moeten gebruiken, zonder de interne werking te moeten bestuderen

3) Analyse : testen of het systeem **voldoet aan de specificaties**. Dit doen we na elke synthesestap.
We testen op :

⇨ functionaliteit : doet het wat het moet doen

⇨ kostprijs : deze is evenredig met de **oppervlakte** van de chip en het **aantal pinnen**

⇨ vermogenverbruik : $C^*f^* V^2$: dit is in de loop der jaren sterk **toegenomen**. Men probeert dit sterk terug te dringen voor (1) **mobiele toepassingen** met batterij
(2) de **warmteproductie** (al dit vermogen wordt omgezet in warmte)
- **C** is de **capaciteit** : oppervlakte waarop gewerkt kan worden 0.25 naar 4 cm^2 (we kunnen meer op één chip zetten)
- **f** is de **frequentie** : vooral deze factor is sterk toegenomen : 1MHz naar 1GHz (we doen meer bewerkingen per seconde)
- **V** is het **voltage** waarop gewerkt wordt : dit is gedaald in de tijd van 5 naar 1.5 volt (we gebruiken minder stroom omdat we nauwkeuriger werken)

⇨ snelheid : hoe **snel** de chip zijn functionaliteit uitvoert :
- **vertraging** tussen ingangsignaal en een uitgangrespons (het kritische pad)
- **klokperiode** (= frequentie)
- tijd die nodig is om een bepaald algoritme of programma uit te voeren
  of de **throughput** (aantal resultaten per seconde)

⇨ testbaarheid : we testen elke chip wanneer de gemaakt is op **foute responsie** op testsignalen
- we willen zoveel mogelijk **fabricagefouten kunnen ontdekken**
- we willen die **zo snel mogelijk** kunnen ontdekken (met zo weinig mogelijk tests)

Gegevensvoorstelling

1) Getallen digitaal voorstellen :

Maak in dit stuk goed onderscheidt tussen getallen en cijfers !

In digitale systemen zullen we alle vormen van getallen verwerken en opslagen als **binaire getallen** bestaande uit bits.

⇨ **Positieel numeriek systeem** : we schrijven een getal D in een bepaald talstelsel als een opeenvolging van **cijfers**. De waarde van het getal is de gewogen som van de waarde van de cijfers. De **waarde** of het gewicht van elk cijfer d_i hangt af van zijn **positie** in het getal : elke positie komt overeen met een macht van het grondtal r : d^*r^i.

Dit **grondtal** r noemt de **radix** en is thevens het aantal gebruikte cijfers. De positie van het cijfer (dus zijn macht van het grondtal) bepalen we t.o.v. het **radixpunt** : dat is het cijfer op positie nul dat gewoon zijn eigen waarde heeft (macht 0 van het grondtal). We schrijven het talstelsel waarin een getal geschreven is rechts onder het getal wanneer dit nodig is.

$$D_r = d_{m-1}\cdots d_0, d_{-1}\cdots d_{-n} \quad \equiv \quad D = \sum_{i=-n}^{m-1} d_i \cdot r^i$$

Bv 1234,5610 (in decimale notatie)

$= 1\cdot10^3 + 2\cdot10^2 + 3\cdot10^1 + 4\cdot10^0 + 5\cdot10^{-1} + 6\cdot10^{-2}$

$= 1\cdot1000 + 2\cdot100 + 3\cdot10 + 4\cdot1 + 5\cdot0{,}1 + 6\cdot0{,}01$

MSB (most-significant-bit) is het cijfer vooraan dat dus het **grootste gewicht** en dus het meest bepalend is voor de waarde van het getal. De **LSB** (least-significant-bit) is het cijfer achteraan en heeft dus het **kleinste gewicht**.

Verschillende talstelsels :

- ⇨ **Binair** : radix 2 / cijfers 0 en 1
- ⇨ **Octagonaal** : radix 8 / cijfers 0 tot 7
- ⇨ **Hexadecimaal** : radix 16 / cijfers 0 tot 9 en A tot F (A=10 B=11 C=12 D=13 E=14 F=15)

⇨ **Radix-conversie** : Overgaan van het ene talstelsel naar het andere doe je best door een **klein gemeen veelvoud** van beide radixen te zoeken, daarnaar om te zetten, de cijfers te hergroeperen en dan kan je vlot naar de beoogde radix overgaan.

- ⇨ bv. overgaan tussen **octale of hexadecimale radices** doen we door om te zetten naar binair talstelsen (radix 2, klein gemeen veelvoud van 8 en 16), dan de bits te groeperen in **groepen van 3 of 4,** en dan kunnen we elk groepje appart vlot omzetten in een hexadecimaal resp. octagonaal cijfer.

Wanneer er **geen klein gemeen veelvoud** is, gaan we best over naar decimaal talstelsel en zetten we dan verder om naar het beoogde talstelsel.

We maken gebruik van de formule $D = ((\ldots((d_{m-1})\ r + d_{m-2})\ r + \ldots)\ r + d_1)\ r + d_0$

- ⇨ overgaan van **radix r naar decimaal** : beginnend bij het meest beduidende cijfer nemen we telkens een cijfer, vermenigvuldigen met de radix r en tellen het volgende cijfer op, … tot we uiteindelijk het laatste cijfer opgetelt hebben.
- ⇨ overgaan van **decimaal naar radix r** : telkens delen door radix r, de rest opschrijven als meest beduidend cijfer, en dan het quotiënt nog eens delen door radix r, weer rest noteren, …

(modulo = bewerking)

2) Rekenen met binaire natuurlijke getallen

⇨ **Optelling en aftrekking** zijn analoog met het decimale systeem. Steeds de 2 cijfers met hetzelfde gewicht (dezelfde macht van de radix) optellen of aftrekken.

- **Overdracht** (= carrey) : wanneer we 1+1 krijgen wordt dit nul en dragen we een 1 over naar de volgende macht van het grondtal.
- **Lenen** (= borrow) : wanneer we 0 – 1 krijgen, moeten we een 1 gaan lenen bij de hogere macht (die heeft waarde 2) waar we dan dus 1 van aftrekken en nog 1 overhouden.

⇨ **Vermenigvuldiging en deling** is ook analoog aan het decimaal systeem. Voor elke 1 in de vermenigvuldiger herhalen we het vermenigvuldigtal onder de anderen, en elke vermenigvuldiger keer zodanig verschoven dat zijn positie overeenkomt met de positie van de overeenkomstige 1 (de juiste macht). Optellen van al deze verschoven vermenigvuldigers geeft het product.
Het aantal optellingen is het aantal bits van de vermenigvuldiger. (kan ook efficiënter)

bv. Optellen :

```
overdracht  0 0 1 1 1 1 1
          x 1 0 0 1 1 0 1 1
          y   1 0 1 0 1 1 1
        som 1 1 1 1 0 0 1 0
```

Aftrekken :

```
lenen       1 1 1 0
          x 1 1 1 0 1
          y   1 1 1 1
  resultaat 0 1 1 1 0
```

Product :

```
    1 0 1 1 0
  x     1 0 1
  -------------
    1 0 1 1 0
  0 0 0 0 0
+ 1 0 1 1 0
  -------------
  1 1 0 1 1 1 0
```

Deling :

```
1 0 1 1 1 0 1 0 | 1 1 1 0
- 1 1 1 0       | -------
  -------       | 1 1 0 1
  1 0 0 1 0 1 0
-   1 1 1 0
  -------
    1 0 0 1 0
-   0 0 0 0
  -------
    1 0 0 1 0
-     1 1 1 0
  -------
      1 0 0
```

3) Negatieve getallen

Met n bits kan je 2^n getallen binair voorstellen. Wanneer we nu ook hun 2^n negatieve equivalenten willen gebruiken, hebben we 2^{n+1} getallen en hebben we zowiezo een extra bit nodig. We kunnen die bit nu op 2 manieren toevoegen :

1) Sing-magnitude voorstelling :
We zetten hier **voor elke getal** een **tekenbit**. Het binaire getal bestaat dan uit zijn **waarde** (magnitude) en een extra bit als **teken** (sign). De tekenbit staat vooraan (MSB) en is 0 voor + en 1 voor -.
Merk op dat we nu **2 notaties** hebben **voor 0** (als -0 : 00…0 en +0 : 10…0).

⇨ Dit is **geen goed systeem** voor de optelling (en vermenigvuldiging, wat neerkomt op een som). We moeten immers voor elke bewerking eerst de **tekens vergelijken** en in functie daarvan handelen (tel de tekenbits niet zomaal op !).
We kunnen op basis hiervan een optel-algoritme opstellen, dat traag en weinig gebruikt is :

- **zelfde tekens** : behoud teken en tel op
- **verschillend teken** : als zelfde getallen : geeft nul /
  als verschillende getallen : trek kleinste van grootste af en neem teken grootste

2) <u>Twee-complement</u> voorstelling :

De tekenbit wordt niet expliciet genoteerd maar wordt verweven in het **2-complement**.

⇨ ***(cijfer)-complement D'*** van getal D bekomen we door in D **elk cijfer te vervangen door zijn complement**, dat is het cijfer dat erbij opgeteld (r-1) geeft. We vervangen het grootst mogelijke cijfer door het kleinst mogelijke, het 2de grootste door het 2de kleinste, enz.

- voor **radix 10** is dit het **9-complement** :
  we vervangen 9 door 0, 8 door 1, 7 door 2, 6 door 3, 5 door 4, en omgekeerd
- voor **radix 2** is dit het **1-complement** : we vervangen 0 door 1 en omgekeerd

⇨ ***radix-complement D**** van een getal D is $\mathbf{r^m - D}$ met m het aantal cijfers in D.

- voor **radix 10** is dit het 10-complement : bv. Voor D=591 is $D^*=10^3-591=1000-591=409$
- voor **radix 2** is dit het **2-complement** : bv. Voor D=101 is $D^*=2^3-101=1000-101=011$

⇨ Eigenschap : **D*=D'+1** : Dit is een heel makkelijke manier om het **radix-complement van een getal te zoeken**. (handige manier om de negatieve versie van een binair getal te berekenen)

- Wanneer we nu met een **vast aantal bits (n)** werken, zien we dat $\mathbf{D^*} = r^n-D = 0-D = \mathbf{-D}$
  (r^n bestaat volledig uit nullen als we de n laatste cijfers bekijken).
  Daarom is D^* een logische voorstelling voor -D
- Wanneer we het **radix-complement van 0** nemen, zien we dat we **opnieuw nul** krijgen, buiten het feit dat we overdracht van 1 krijgen bij de meest beduidende bit. Laten we die overdracht weg (= we behouden altijd n cijfers) dan is het radix-complement van 0 gelijk aan 0.
  We zijn nu verlost van -0 en +0

⇨ Een **negatief binair getal** wordt nu voorgesteld door zijn **2-complement** : dit is eenvoudig te berekenen door alle cijfers om te draaien, 1 op te tellen en de eventuele overdracht van de meest beduidende bit weg te laten. **0 wordt nu op zichzelf afgebeeld**, en elk cijfer heeft een unieke negatieve voorstelling. Deze voorstelling van negatieve getallen is vooral handig bij optellen en aftrekken.

- Let wel dat elke voorstelling van een negatief getal in 2-complement de positieve voorstelling is een ander getal in sign-magnitude. We moeten dus **afspreken** waar we over bezig zijn (**2-complement of 'sign-magnitude'**)
- het 2-complement van een getal van n bits **ligt tussen** (-2^{n-1}) en $(2^{n-1}-1)$.

⇨ We kunnen dus met **n bits** ofwel **n positieve getallen** afbeelden, ofwel **(n/2)-1 positieve en n/2 negatieve**. Het 2-compement kan nu gebruikt worden voor de (n/2) negatieve getallen, daar elk positief getal een n-complement heeft dat niet tot de (n/2)-1 positieve getallen behoort.

Bv. voor 4 bits : getallen tss 7 en -8 voorstelbaar met 4 bits.

0…7 = 0000…0111 en -1…-8 = 1111….1000

- merk op dat elke binaire combi gebruikt is
- merk op dat elk negatief getal een 1 heeft als meest beduidend cijfer, en positieve getallen en nul een 0. Dit zal later handig blijken (heeft dus dezelfde voordelen als sign-magnitude).

<u>Optellen en aftrekken</u> is met het 2-complement veel handiger dan met sing-magnitude.

Regels waarop we moeten letten bij het **optellen** :

⇨ Wanneer we **2 negatieve** getallen van **n bits** in 2-complement optellen, is het correcte resultaat de **n minst beduidende bits** van het bekomen resultaat. Dit komt neer op het laten vallen van de meest beduidende carrey-over bit. Zorg er dus altijd voor dat je altijd met een **vast aantal bits** werkt. Bv.

```
  0010  +2        0010  +2        1110  -2
  0100  +4        1100  -4        1100  -4
+ 00000         + 00000         + 11000
  0110  +6        1110  -2        1010  -6
```

⇨ Wanneer we **2 getallen** van **n bits** met **zelfde teken** optellen kan de som **buiten het voorstelbare bereik van n bits** vallen (zowel te groot als te klein). We krijgen dan een verkeerd resultaat bij het optellen : **overflow**. We kunnen overflow **detecteren** door het teken van beide op te tellen getallen (hebben hetzelfde teken) te vergelijken met dat van het resultaat. Wanneer ze verschillend zijn hebben we overflow. Bv.

```
  0111  +7        1001  −7
  0110  +6        1010  −6
+ 01100       +  10000
  1101  −3        0011  +3
```

Voor het **aftrekken** van getallen nemen we gewoon het 2-complement van wat er afgetrokken moet worden en tellen we dat op.

```
  0010  +2        0010  +2        1110  −2
  1011  (+4)'     0011  (−4)'     0011  (−4)'
+ 00111       +  00111       +  11111
  1110  −2        0110  +6        0010  +2
```

Modulus : **A mod B** is de rest van A bij deling door B.
A rem B = a – ([het geheel aantal keer dat b in a past] x b)

a	*b*	*a* mod *b*
> 0	> 0	*a* rem *b*
> 0	< 0	(*a* rem *b*) + *b*
< 0	> 0	(*a* rem *b*) + *b*
< 0	< 0	*a* rem *b*

a	*b*	rem	mod
5	3	2	2
5	−3	2	−1
−5	3	−2	1
−5	−3	−2	−2

Wanneer we van een **binair getal** modulus 2^n willen, komt dit neer op de n laatste bits te laten vallen

4) Niet gehele getallen : floating poits en bewegende komma

⇨ **Vaste komma** : **i gehele** getallen voor de komma, **f fractionele** getallen na de komma.
Voorgesteld als **fix<i,f>**
- **optelling** : fix<i_1,f_1> + fix<i_2,f_2> geeft getal met **f=max(f_1,f_2)** en **i=max(i_1,i_2)+1**
waarbij de +1 volgt uit de eventuele overdracht van de binaire som

Tellen we n keer hetzelfde getal op dan wordt i=max(i_1,i_2)+log$_2$(n) daar we telkens overdracht krijgen van 1 bit

- **product** : fix<i_1,f_1> x fix<i_2,f_2> geeft een getal met **f=f_1+f_2** en **i=i_1+i_2**

De **essentie** is hier dat we bij het **uitvoeren van bewerkingen** telkens **getallen bijkrijgen**. Wanneer we in een een schakeling een **vast aantal bits** voorzien voor een getal is dit **geen handige manier** om te werken. Daarom zullen we het systeem van de **vlottende komma** invoeren om met een vast aantal bits per getal te rekenen.

⇨ **Vlottende komma** : In plaats van de komma eender waar in het getal te laten staan, zullen we hem nu telkens tot net achter de meest beduidende bit schuiven. We moeten dan onthouden hoeveel plaatsen hij verschoven is : dit doen we door het getal te vermenigvuldigen met de juiste macht van de radix (door de macht te onthouden kunnen we de komma altijd terug op zijn plaats zetten).

⇨ Een getal in bewegende komma bestaat uit : [signbit][exponent][fractie].

- ***signbit S*** : 0 voor + en 1 voor – zodat we het volledige getal kunnen laten beginnen met $(-1)^s$
- ***exponent E*** : de macht van de radix R waarmee we de mantisse moeten vermenigvuldigen om de komma op de juiste plaats te zetten. Is dus ook het aantal plaatsen dat de komma verschoven is.

  We willen nu eigenlijk liefst niet met negatieve exponenten werken : we zullen hem coderen in excess-code genaamd characteristic. Dit houdt is dat we bij de exponent een **bias** optellen : **bias B =2^{e-1}-1** met e het aantal bits voor de fractie. De bias heeft nu de grootte van de meest negatieve exponent die we kunnen voorstellen : door er de bias bij op te tellen worden alle exponenten zijn nu positieve getallen. We herstellen de exponent door er gewoon de bias terug af te trekken wanneer we ermee willen werken.
- ***fractie F*** : via de exponent wordt de **komma tot net achter de meest beduidende bit** geschoven. Elk getal ziet er dus uit als 1,abc… met 1 het eerste meest beduidende cijfer. Dit noemen we de **mantisse** en die moeten we ook nog onthouden. We weten echter dat het meest beduidende cijfer 1 is in een binair systeem, dus moeten we dit niet onthouden. We onthouden enkel een vast aantal getallen na die 1. Dat noemen we de **fractie**.

Een getal wordt nu dus als volgt voorgesteld :

	$E = 0$	$0 < E < 2^e - 1$	$E = 2^e - 1$
$F = 0$	0	$(-1)^S \cdot 1,F \cdot 2^{E-B}$	$(-1)^S \cdot \infty$
$F \neq 0$	$(-1)^S \cdot 0,F \cdot 2^{-B+1}$	$(-1)^S \cdot 1,F \cdot 2^{E-B}$	NaN

niet-genormaliseerde F → verborgen bit

⇨ Elk deel heeft een **vast aantal bits : float<f,e>** betekent 1 signbit, e getallen voor de exponent en f getallen voor de fractie. Standaart waarden voor deze getallen : IEEE-formaat :

- enkelvoudige precisie : e=8 en f=23 ⇨ B = 127
- dubbele precisie : e=11 en f = 52 ⇨ B = 1023

⇨ We zien nu dat we met bewegende komma een **groter bereik** hebben maar **minder precisie** :

stel dat we een getal voorstellen met 4 bits. (neem voor vlottende komma float<2,2>)

⇨ **vaste** komma : getallen van 0 tot 10^3 voorstelbaar (klein bereik)
altijd een precisie tot op de eenheid (goeie precisie)

⇨ **vlottende** komma : getallen van 0 tot 10^{99} voorstelbaar (veel groter bereik)
precisie tot op macht van de radix (kan heel weinig precies zijn)

We kunnen echter wel met ongeveer dezelfde precisie werken wanneer we dat willen,

⇨ **rekenen met vlottende komma** :

- **optellen** : eerst exponenten gelijk maken of mantissa's verschuiven t.o.v. elkaar om gelijke grootorde te krijgen. Dan mantissa's optellen en normaliseren. Mogelijk verlies van elke precisie t.o.v. het kleinste getal door normaliseren: bv manisses volledig naast elkaar geschoven
- **product** : mantissa's vermenigvuldigen en exponenten optellen. Daarna normaliseren. We verliezen daarbij hooguit zoveel cijfers als 1 getal heeft, maar het verlies aan precisie is gelijk voor grootste en kleinste getal
- **Floating-point-overflow** : wanneer de exponent te groot wordt kunnen we het getal niet meer benaderend voorstellen en wordt het oneindig genoemd

5) Andere codes: BCD, ASCII, ECC, …

⇨ **BCD code : Binairy Coded Decimal :** Code voor decimale getallen voor te stellen
We kennen aan elk cijfer tussen 1 en 9 een 4bit binaire code toe die overeenstemt met het cijfer : 0=0000, 1=0001, … 9=1001. De andere binaire getallen 1010 tot 1111 worden niet gebruikt.
- **negatieve getallen** worden voorgesteld met hun 10-complement (zowiezo extra bit nodig)
- **optellen** gebeurt zoals altijd, alleen moeten we oppassen met overdracht omdat niet elk binair cijfer gebruikt is : voor getallen tussen 10 en 19 moeten we de overdracht corrigeren met 10

⇨ **ASCII : American Standard Code for Information Interchange** : Karaktercode die elke alfabetletter en een hoop toetsenbord-tekens codeert met 7 bits. Kan makkelijk opgeslagen worden in 1 byte (8 bits) waarbij de 8ste bit een foutcorrectiebit is.

Booleaanse algebra

1) Axiomatische definitie van booleaanse algebra

Booleaanse algebra is een set van elementen B {0,1}, een set van operatoren [+, ·, '] en **axiomas** :

⇨ *Axioma 1* : B is **gesloten** voor + en ·
het resultaat van optelling of product is opnieuw elemente van B :
$x + y \in B$ en $x \cdot y \ (= xy) \in B$

⇨ *Axioma 2* : B heeft een **eenheidselement** 0 voor + en een eenheidselement 1 voor ·
resultaat van bewerking van getal en eenheidselement is terug dat getal :
$x + 0 = x$ en $x \cdot 1 = x$

⇨ *Axioma 3* : B is **commutatief** voor + en ·
plaats van elementen in de bewerking is niet belangrijk :
$x + y = y + x$ en $x \cdot y = y \cdot x$

⇨ *Axioma 4* : B is **distributiviteit** voor + en ·
volgorde der bewerkingen is niet belangrijk : haakjes uitwerken :
$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$ en $x + (y \cdot z) = (x + y) \cdot (x + z)$

⇨ *Axioma 5* : B heeft voor + en · voor elk getal een **complementair element**
elk element heeft een tegengestelde : elke bewerking van een element met zijn tegengestelde geeft het eenheidselement voor die bewerking :
$\forall x \in B, \exists x' \in B : x + x' = 1$ en $x \cdot x' = 0$ (0+1=1 en 0·1=0)

⇨ *Axioma 6* : B heeft **minstens 2 elementen** (cardinality bound)

⇨**Verschillen** met de **gewone algebra :**

⇨ In booleaanse algebra bestaat geen inverse bewerking voor de optelling (OR) of de vermenigvuldiging (AND) : **aftrekking of deling bestaan niet** (invers element wel)

⇨ In gewone algebra is **+ niet distributief t.o.v. ×** : $5 + (2 \times 4) \neq (5 + 2) \times (5 + 4)$
$1 + (1 \cdot 0) = (1+1) \cdot (1+0)$

⇨ In gewone algebra **geldt niet dat $x + x' = 1$ en $x \times x' = 0$**

⇨We definiëren nu **+ als OR**, **· als AND** en **' als inverse** :

NOT-operator

x	x'
0	1
1	0

AND-operator

x	y	x • y
0	0	0
0	1	0
1	0	0
1	1	1

OR-operator

x	y	x + y
0	0	0
0	1	1
1	0	1
1	1	1

2) Theorema's

Naast axiomas die we aannemen (moeten niet bewezen worden) om booleaanse algebra op te bouwen, kunnen we daar nu een aantal theoremas van afleiden. Dit zijn handige uitdrukkingen die helpen bij het vereenvoudigen van booleaanse uitdrukkingen. We kunnen booleaanse uitdrukkingen en theoremas **bewijzen vanuit de axiomas** of door een **waarheidstabel op te stellen**.
Elk theorema voor + heeft een **duaal theorema** dat geldt voor • . Dit heet dualiteit en kunnen we makkelijk bewijzen: Wanneer we in een uitdrukking alle OR door AND vervangen en elke 0 door 1 en omgekeerd, dan blijft de uitdrukking correct.

y	x	y • x	y • x + x
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	1

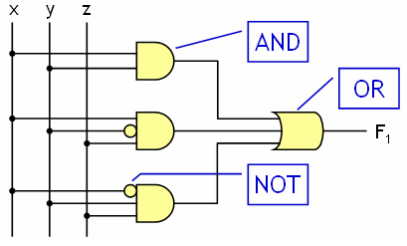
- ⇨ *Theorema 1* : idempotency
  x + x = x en Duaal: **x • x = x**
- ⇨ *Theorema 2* :
  x + 1 = 1 en Duaal: **x • 0 = 0**
- ⇨ *Theorema 3* : absorptie
  (y • x) + x = x en Duaal: **(y + x) • x = x**
- ⇨ *Theorema 4* : involutie
  (x')' = x
- ⇨ *Theorema 5* : associativiteit
  (x + y) + z = x + (y + z) en Duaal: **(x • y) • z = x • (y • z)**
- ⇨ *Theorema 6* : ***wetten van Morgan***
  (x + y)' = x' • y' en Duaal: **(x • y)' = x' + y'**
  Dit is een belangrijk theorema : de inverse van een som van termen is het product van de inversen van die termen

3) Booleaanse functies

- ⇨ Een booleaanse functie is een **uitdrukking** van **binaire variabelen** en de **bewerkingen AND, OR en NOT**. De functie heeft een **binaire waarde** voor **elke set waarden** van de binaire **variabelen**.
- ⇨ De functie wordt opgelost met **prioriteit van bewerkingen** : Haakjes ⇨ Not ⇨ And ⇨ Or
- ⇨ Elke booleaanse functie kan **rechtstreeks gerealiseerd** worden met **poorten**: De binaire variabelen zijn dan de ingangen, het resultaat van de functie is de uitgang. Meerdere uitgangen betekend dus meerdere functies (1 functie per uitgang)
- ⇨ Van elke functie kan een **waarheidstabel opgesteld** worden : een kolom voor elke in- en uitgang, een rij voor elke mogelijke combinatie van ingangen. Dus n ingangen = 2^n rijen.
  Er zijn verschillende volgordes mogelijk voor de rijen :
  - **Standaart-code** : in volgorde van **stijgende binaire waarde** van de ingangen samen
  - **Gray-code** : schikking zodanig dat in **2 opeenvolgende rijen** maar in **1 ingangsvariabele verschilt**

Bijv. $F_1 = xy + xy'z + x'yz$

- F_1 = 1 als x = 1 en y = 1 of als x = 1, y = 0 en z = 1 of als x = 0, y = 1 en z = 1; anders F_1 = 0
- F_1 bestaat uit 3 AND-termen en 1 OR-term



Rij	x	y	z	F_1
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

Standaart-code

x	y	z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Gray-code

x	y	z	F
0	0	0	0
0	0	1	0
0	1	1	1
0	1	0	0
1	1	0	1
1	1	1	1
1	0	1	1
1	0	0	0

⇨ We kunnen elke functie inverteren : via de wetten van de Morgan kunnen we zo wisselen tussen 2 manieren van implementatie :

⇨ **Product van sommen** (Or-And) : (a+b+c)(d+e+f)(g+h+i)

elke combinatie waarvoor de uitgang 0 moet zijn is een term van het product

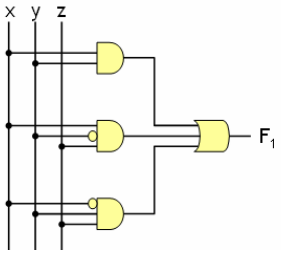
⇨ **Som van producten** (And-Or) : = (abc + def + ghi)

elke combinatie waarvoor de uitgang 1 moet zijn is een term van de som

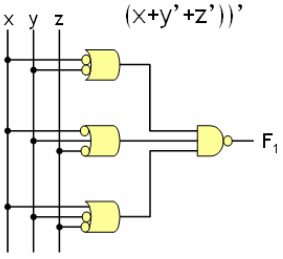
⇨ **De morgan** : (a'+b'+c')(d'+e'+f')(g'+h'+i') = (abc + def + ghi)'

F_1' = (xy + xy'z + x'yz)'
= (xy)'(xy'z)'(x'yz)' (De Morgan)
= (x' + y')(x' + y + z')(x + y' + z') (De Morgan)

AND-OR realisatie
F_1 = xy + xy'z + x'yz



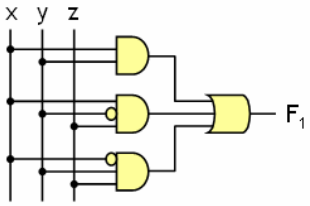
OR-AND realisatie
F_1 = ((x'+y') (x'+y+z') (x+y'+z'))'



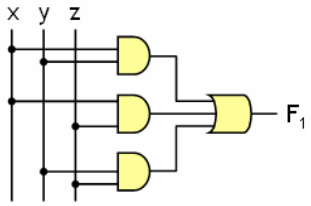
⇨ Ons doel is nu elke booleaanse functie **om te vormen naar zijn korste en eenvoudigste vorm** ; de vorm die het minste kost aan poorten om te realiseren. We kunnen die op de tast doen door de **theoremas toe te passen** :

F_1 = xy + xy'z + x'yz
= xy + xyz + xy'z + x'yz (absorptie)
= xy + x(y+y')z + x'yz (distributiviteit)
= xy + x1z + x'yz (complement)
= xy + xz + x'yz (identiteit)
= xy + xyz + xz + x'yz (absorptie)
= xy + xz + (x+x')yz (distributiviteit)
= xy + xz + 1yz (complement)
= xy + xz + yz (identiteit)

F_1 = xy + xy'z + x'yz



F_1 = xy + xz + yz



4) Canonische & standaard vorm

Opstellen van booleaanse functies vanuit de **waarheidstabel** : Elke booleaanse functie kan beschreven als een **som van zijn 1-mintermen** of een **product van zijn 0-maxtermen**

⇨ **minterm** : booleaanse functie die waar is voor **slechts één enkele rij** van de waarheidstabel :

- 0-minterm als de functie er nul is, 1-minterm als de functie er 1 is.
- een minterm is altijd een **product** van de n ingangsvariabelen al dan niet geïnverteerd

⇨ **maxterm** : booleaanse functie die waar is voor **alle rijen van de waarheidstabel behalve één rij**.

- 0-maxterm als de functie er nul is, 1-maxterm als de functie er 1 is.
- een maxterm is altijd een **som** van de n ingangsvariabelen al dan niet geïnverteerd

Rij	x	y	z	minterm	notatie
0	0	0	0	x'y'z'	m_0
1	0	0	1	x'y'z	m_1
2	0	1	0	x'yz'	m_2
3	0	1	1	x'yz	m_3
4	1	0	0	xy'z'	m_4
5	1	0	1	xy'z	m_5
6	1	1	0	xyz'	m_6
7	1	1	1	xyz	m_7

Rij	x	y	z	F_1	1-minterm
0	0	0	0	0	–
1	0	0	1	0	–
2	0	1	0	0	–
3	0	1	1	1	m_3 = x'yz
4	1	0	0	0	–
5	1	0	1	1	m_5 = xy'z
6	1	1	0	1	m_6 = xyz'
7	1	1	1	1	m_7 = xyz

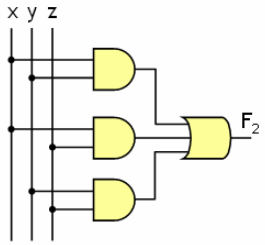
Rij	x	y	z	maxterm	notatie
0	0	0	0	x+y+z	M_0
1	0	0	1	x+y+z'	M_1
2	0	1	0	x+y'+z	M_2
3	0	1	1	x+y'+z'	M_3
4	1	0	0	x'+y+z	M_4
5	1	0	1	x'+y+z'	M_5
6	1	1	0	x'+y'+z	M_6
7	1	1	1	x'+y'+z'	M_7

Rij	x	y	z	F_1	0-maxterm
0	0	0	0	0	$M_0 = x+y+z$
1	0	0	1	0	$M_1 = x+y+z'$
2	0	1	0	0	$M_2 = x+y'+z$
3	0	1	1	1	–
4	1	0	0	0	$M_4 = x'+y+z$
5	1	0	1	1	–
6	1	1	0	1	–
7	1	1	1	1	–

⇨ ***Canonische vorm*** : som van 1-mintermen of product van 0-maxtermen : elke min- of maxterm bevat **alle ingangsvariabelen** hetgeen een dure implementatie oplevert. (= maximale implementatie)
We moeten de canonische vorm dus omvormen tot een eenvoudigere vorm. Bemerk dat elke maxterm het complement is van de overeenkomstige minterm en omgekeerd.
We gebruiken de **canonische vorm als startuitdrukking** omdat we hem makkelijk uit de waarheidstabel kunnen opstellen. Dan gaan we door **vereenvoudiging** over naar **standaartvorm** :
⇨ ***Standaart vorm*** : ***som van producttermen*** of een ***product van somtermen*** met het **kleinst mogelijke aantal variabelen**. Dit is dus de canonische vorm in eenvoudigere versie, omdat niet elke productterm of somterm alle variabelen moet bevatten. Dit is de goedkoopste impementatie in 2 lagen (een And- en een Or-laag). Soms kunnen we met meer lagen nog goedkoper uitkomen. Hoe we de standaartvorm vinden komt straks.

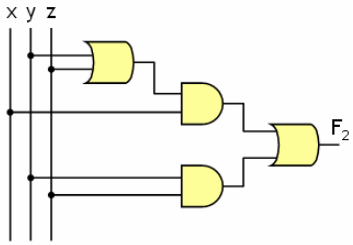
Standaartvorm :

$F_2 = xy + xz + yz$



Meer-lagen optie :

$F_2 = x(y + z) + yz$



5) 16 functies van 2 variabelen

Met **2 binaire variabelen** kunnen we **2^2=4 mogelijke ingangscombinaties** krijgen. Voor een willekeurige functie kan elke ingangscombinatie een **0 of 1 als functiewaarde** geven. We kunnen dus **4^2=16 verschillende functies** definiëren op 2 ingangsvariabelen :

x	y	F_0	F_1	F_2	...	F_{15}
0	0	0	0	0		1
0	1	0	0	0		1
1	0	0	0	1		1
1	1	0	1	0		1

Naam	Symbool	Functiewaarde voor x,y				Uitdrukking
		00	01	10	11	
Zero	–	0	0	0	0	$F_0 = 0$
AND	x·y	0	0	0	1	$F_1 = xy$
Inhibition	x/y	0	0	1	0	$F_2 = xy'$
Transfer	–	0	0	1	1	$F_3 = x$
Inhibition	y/x	0	1	0	0	$F_4 = x'y$
Transfer	–	0	1	0	1	$F_5 = y$
XOR	x⊕y	0	1	1	0	$F_6 = xy'+x'y$
OR	x+y	0	1	1	1	$F_7 = x+y$
NOR	x↓y	1	0	0	0	$F_8 = (x+y)'$
XNOR	–	1	0	0	1	$F_9 = xy+x'y'$
Complement	y'	1	0	1	0	$F_{10} = y'$
Implication	–	1	0	1	1	$F_{11} = x+y'$
Complement	x'	1	1	0	0	$F_{12} = x'$
Implication	–	1	1	0	1	$F_{13} = x'+y$
NAND	x↑y	1	1	1	0	$F_{14} = (xy)'$
One	–	1	1	1	1	$F_{15} = 1$

Logische poorten

Wanneer we **binair willen werken** in een chip moeten we **op een bepaalde manier een 1 en een 0** kunnen voorstellen : We gebruiken daarvoor **2 verschillende spanningen** : **V_L en V_H**, resp. lage spanning en hoge spanning. We zouden ook 2 verschillende stromen kunnen gebruiken, optische reflecties (galsvezelprocessoren), verschillende drukken, enz. (maar spanning is meest handig)

⇨ **positieve logica** : $V_L = 0$ en $V_H = 1$: meest gebruikt

⇨ **negatieve logica** : $V_L = 1$ en $V_H = 0$

⇨ **Een actief laag signaal** is een signaal dat **actief is als het 0** is. Dit is dus de **negatieve logica**. Dit houdt in dat we een binaire 0 weergeven met een bepaald actief signaal (spanning bv), en een binaire 1 dus met een passief signaal (geen spanning bv). **Actief zijn is dus een interpretatie van een signaal, niet van een niveau!** Meestal duiden we het actief lage signaal aan met X, X* of X'.

⇨ Waar en waarom gebruikt :

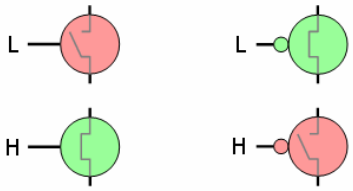
- Bv. **de reset-ingang** van een toestel is meestal een actief laag signaal : we resetten het toestel als het actieve signaal (0) passief wordt (1 = dus geen signaal meer) reset. RST*=0
- we kunnen door te switchen tussen actieve en passieve signalen 1 implementatie voor 2 verschillende poorten gebruiken : bv. **Actief lage OR is actief hoge AND**.
- Actief lage signalen worden gebruikt in **afgesloten verbindingen** : inactieve signalen moeten daar meestal 1 en actieve signalen 0
- omwille van **wired-or functionaliteiten** bij een **'open-drain'** implementatie

1) Functionele eigenschappen

Basispoorten

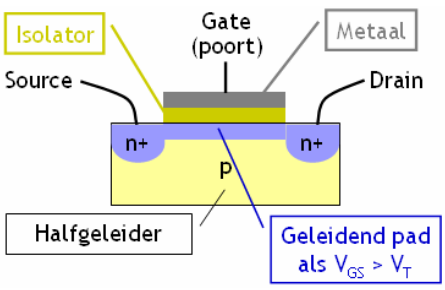
Een **logische poort** is meestal **opgebouwd uit schakelaars** met 2 standen (aan- of uitgeschakeld) die geschakeld (gestuurd) worden met een **stuursignaal** (transistor). Voor een elektrische chip is dit weerstand 0 of oneindig, voor een druksysteem is dat een open of een gesloten klep, voor lichtprocessor het wel of niet doorlaten van licht.

⇨ De schakelaar is **onderbroken** bij een **laag** signaal, **gesloten** voor een **hoog** signaal. Met invertoren kunnen we de werking aanpassen



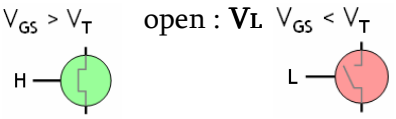
⇨ De schakelaar is meestal een **MOS transistor** (Metal-Oxide-Semiconductor) : Dit is een **spanningsgestuurde schakelaar** die open of gesloten is afhankelijk van de spanning op zijn sturing. De spanning op de sturing noemen we V_{GS}, de grens tussen V_L en V_H noemen we V_T (dus $V_L = V_{GS} < V_T$ en $V_H = V_{GS} > V_T$). Al deze voltages zijn negatief voor de MOS-transistor.

De transistor is **opgebouwd** uit een **plaatje metaal** en een **plaatje halfgeleider** met een **isolator** tussen. Op het metalen plaatje komt de spanning van de **Gate (basis)** : dit is de **sturing** en die bepaald of de halfgeleider tussen **source (collector)** en **drain (emittor)** al dan niet geleidend is.

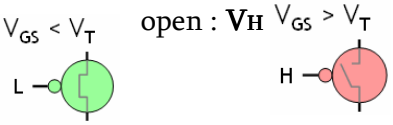


In **MOS-technologie** kunnen we **2 soorten** transistors maken :

⇨ **NMOS** transistor (**P-transistor** in boek) : **open** (niet-geleidend) bij **VL** op de sturing, **gesloten** (geleidend) bij **VH** op de sturing gesloten : **VH** $V_{GS} > V_T$ open : **VL** $V_{GS} < V_T$



⇨ **PMOS** transistor (**N-transistor** in boek) : **open** (niet-geleidend) bij **VH** op de sturing, **gesloten** (geleidend) bij **VL** op de sturing gesloten : **VL** $V_{GS} < V_T$ open : **VH** $V_{GS} > V_T$



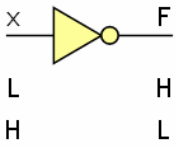
De **werking** van de N en P zijn **invers**, hetgeen handig is voor de implementatie van poorten. We kunnen nu **verschillende poorten** maken met behulp van de 2 MOS-transistoren (afgekort 'tors').

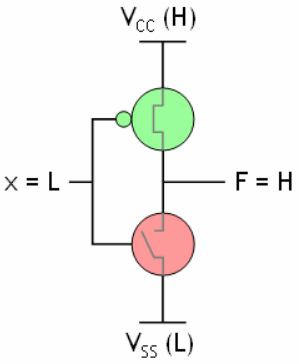
De **basiswerking** van de schakelingen voor poorten : we **schakelen** telkens de tranisitors met de **ingangssignalen van de poort** (komen dus op de sturing) en op de **collectors** van de transistors zetten we **VL of VH van een externe bron**. Externe bron duiden we aan met **Vcc(H)** voor hoog signaal en **Vss(L)** voor laag signaal.

Implementatie van de 3 basis-**poorten** in CMOS-technologie :

⇨ ***Invertor*** : F = x'

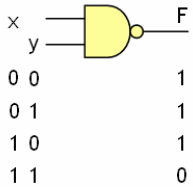
- 2 **transistoren** : 1 N & 1 P
- **relatieve vertraagtijd** = 1
  (invertor is de referentie voor vertraagtijd)
- wanneer x = VL wordt de ene tranistor gesloten en krijgen we VH op de uitgang, wanneer x = VH, wordt de andere gesloten en krijgen we VL op de uitgang.

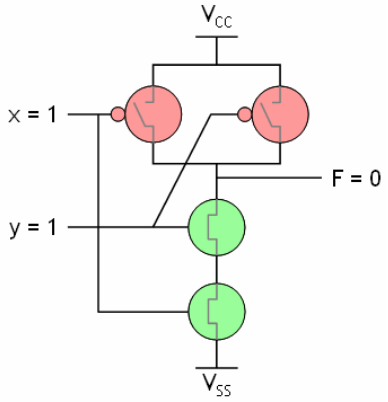




⇨ ***NAND-poort*** : F = (xy)'

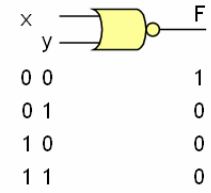
- 4 **transistoren** : 2 N & 2 P
- **relatieve vertraagtijd** : 1,4
- Als minstens één van beide ingangen 0 is, dan is VL daardoor afgesloten en het bovenste systeem aangesloten en krijgen we VH op de uitgang, wanneer beide 1 zijn is VL aangesloten.

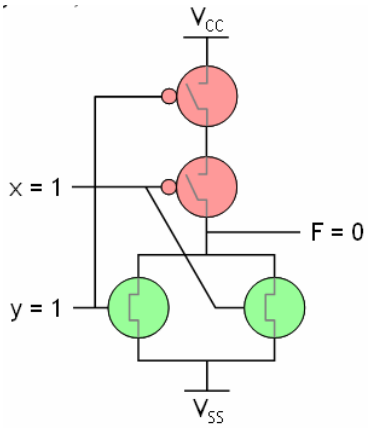




⇨ ***NOR-poort*** : F = (x+y)'

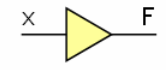
- 4 **transistoren** : 2 N & 2 P
- **relatieve vertraagtijd** : 1,4
- Voor VH op de uitgang moeten beide ingangen 0 zijn. Vanaf dat er één ingang 1 is, is VH geblokkeerd wordt VL naar de uitgang gestuurd.





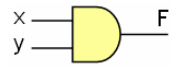
De **invertor**, **NAND** en **NOR** zijn de 3 basispoorten : Alle andere poorten worden eruit gebouwd :

⇨ **Buffer of 'driver'** : de functie is gelijk aan het ingangssignaal : gebruikt om meer vermogen te kunnen gebruiken (signaal wordt weer versterkt tot op originele niveau)

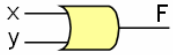


F = x = (x')' 4 tor vertraging = 2

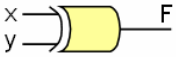
⇨ **AND-poort** : F = xy = ((xy)')' 6 tor vertraging = 2,4



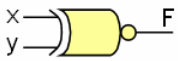
⇨ **OR-poort** : F = x + y = ((x + y)')' 6 tor vertraging = 2,4



⇨ **XOR-poort** : F = x ⊕ y = xy' + x'y = ((x + y') (x' + y))' 12 tor (cfr. OAI) vertraging = 3,2

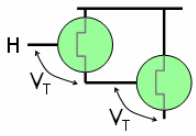


⇨ **XNOR-poort** :F = (x ⊕ y)' = xy + x'y' = ((x + y)(x' + y'))' 12 tor (cfr. OAI) vertraging = 3,2



We maken **enkel inverterende poorten** (invertor, NAND en NOR) met MOS transistoren omwille van de werking van de transistoren :

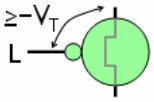
⇨ **NMOS transistor** is slecht **pull-up** : wanneer de NMOS gesloten is, wordt enkel V_L makkelijk doorgegeven, en niet V_H omdat V_H met V_T verminderd wordt.



Daarom zullen we de NMOS alleen gebruiken om de V_L aan en uit te schakelen en niet V_H.

⇨ dus **op NMOS collector alleen V_L**

⇨ **PMOS transistor** is een slechte **pull-down** : wanneer de PMOS gesloten is, wordt enkel V_H makkelijk doorgegeven, en niet V_L.



Daarom zullen we PMOS alleen gebruiken om V_H door te geven.

⇨ dus **op PMOS collector alleen V_H**

We kunnen dus enkel V_L doorgeven met een ingang V_H en V_H doorgeven met ingang V_L : dit resulteert zowiezo in inverterende poorten

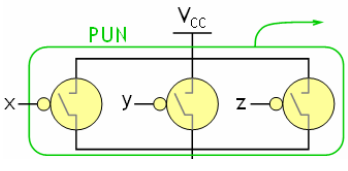
Meerdere ingangen (fan-in)

Wanneer een poort **meerdere ingangen** heeft, heeft hij ook **meer transistors** nodig voor zijn werking en zal hij dus **duurder en trager** zijn. We vergelijken een 3-input NAND met een 2-input NAND :

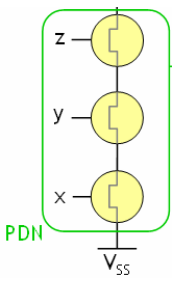
⇨ Geeft V_H als minstens één van de 3 ingagnen 0 is : parallelschakeling van 3 PMOS,
Geeft V_L als alle 3 de ingangen 1 is : serieschakeling van 3 NMOS

We zien dat we van 4 naar 6 transistors gaan, en **per extra ingang nog 2 extra transistors** nodig hebben. Analoog voor NOR. De vertragingstijd is hier 1,8 (voor 2 ingangen is dat 1,4)

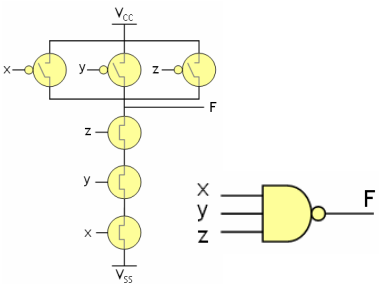
V_H als :



V_L als :



samen :



De **kostprijs en de vertraging** van een poort is **afhankelijk van de Fan-in** (= het aantal ingangen) en worden **relatief uitgedrukt** t.o.v. de waarden voor de **Invertor** : De invertor kost zoveel als 2 transistoren en heeft als vertraging de tijd die een transistor nodig heeft om te schakelen.

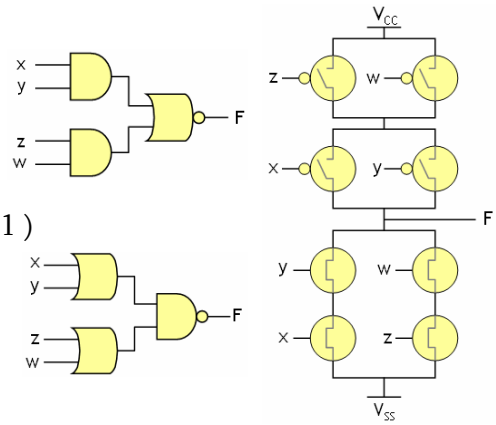
Elke **niet-inverterende** poort (AND, OR, …) heeft de **waarde van de inverterende +1** omdat voor de niet-inverterende een **extra invertor** nodig is (en die heeft alle waarden 1)
(Inverterende poorten zijn invertor, NAND en NOR)

- ⇨ **Kostprijs** : - voor inverterende poort : **Fan-in**
  - voor niet-inverterende poorten : **Fan-in + 1**
- ⇨ **Vertraging** : - voor inverterende poort : **0.6 + Fan-in*0.4**
  - voor niet-inverterende poorten : **1.6 + Fan-in*0.4**

Meerdere operatoren : complexere poorten

Wanneer we **complexe schakelingen** krijgen, is het soms voordeliger om een bepaalde combinatie van poorten **rechtstreeks te implementeren** i.p.v via standaart-poorten te werken :
Bv. de 2-wide 2-input AND-OR-Invertor (=AOI) :

- ⇨ **F = (xy+zw)'** : F = 0 als (x = 1 en y = 1) of (z = 1 en w = 1)
  We kunnen deze schakeling implementeren met
  8 transistors en een vertraging van 2.5,
  in plaats van met 2 ANDs en een NOR
- ⇨ **F = ((x+y)(z+w))'** : F = 0 als (x = 1 of y = 1) en (z = 1 of w = 1)
  We kunnen deze schakeling implementeren met
  8 transistors en een vertraging van 2.2,
  in plaats van met 2 ORs en een NAND



2) Niet-functionele eigenschappen

Spanningsniveau's

Logische variabelen 0 en 1 worden voorgesteld door een **bepaald spanningsgebieden**. **V_{SS} en V_{CC}** zijn de **aangelegde spanningen** aan de transistors, **V_{IL} en V_{IH}** zijn de **absolute grenzen** tot waar we de logische variabelen definiëren. Alles wat **er tussen** valt is **onbepaald**.

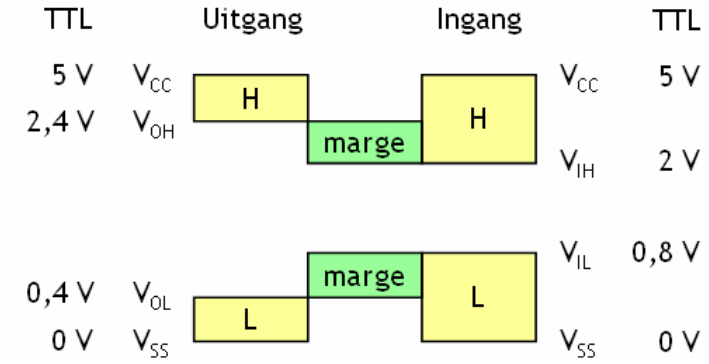
Logisch niveau	Spanningsgebied		TTL	
	van	tot	van	tot
L	V_{SS}	V_{IL}	0 V	0,8 V
H	V_{IH}	V_{CC}	2 V	5 V

Digitaal werken heeft hierdoor **voordelen** tegenover analoge electronica. Digitaal kunnen we ons veel **makkelijker spanningsvariaties permitteren** waardoor het systeem robuuste is :
- minder gevoelig voor **procesvariaties**, die de karakteristieken van de componenten kunnen veranderen
- minder gevoelig voor **omgevingsvariaties** zoals temperatuur, voedingsspanning, enz.

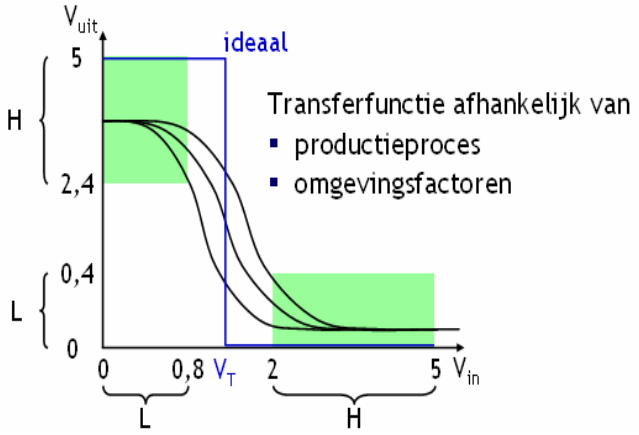
Ruismarges : We ontwerpen nu een chip om op een bepaald vast voltage te werken. We zien echter dat tijdens de werking van de chip de **spanning willekeurig zal variëren** rond dit vooropgestelde voltage. Deze afwijking is het gevolg van variaties op de voedingsspanning, verandering van de temperatuur in de chip, inductie van andere draden, enz. Dit noemen we **ruis**, en we moeten hiermee rekening houden bij de **interpretatie** van de voltages als binaire variabelen :

⇨ We kennen aan een **bepaalde binaire variabele** een **spanningsinterval** toe. Dit is het interval waarin de spanning moet liggen wanneer we de binaire variabele uitzenden (**uitgang** v/d poort)

⇨ Wanneer het signaal nu in een andere poort komt (**ingang**), moet het daar geïnterpreteerd kunnen worden als binaire variabele. Er zit nu echter **ruis** op (het voltage is veranderd) en het is dus mogelijk dat het **voltage buiten het vooropgestelde interval** valt. Toch zouden we graag het signaal nog herkennen en het niet onbepaald classificeren : We moeten bij de interpretatie van een **ingangssignaal dus een groter interval voltages** toelaten. Deze extra toegelaten voltages zijn de **ruismarges**, één voor lage signalen en één voor hoge signalen : Voor CMOS zijn deze 0.4 volt :

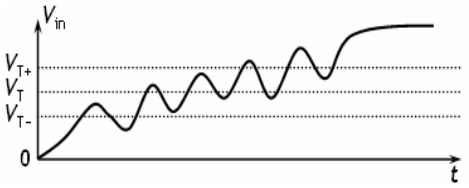


Onlogische spanningen : Het overgaan van het ene logische spanningsniveau naar het andere gebeurt niet ogenblikkelijk. Hoe snel het ook gebeurt, er is altijd een kleine periode waar het signaal **tussen de 2 logische** spanningsgebieden zit en dus **onbepaald** is. De uitgang veranderd tijdens diezelfde periode ook mee en is dan dus ook binair onbepaald. De **transferfunctie** is de functie die het **verband** weergeeft tussen **ingangsspanning en uitgangsspanning**: wanneer de ingang binair bepaald is, is de uitgang dat ook, maar alles wat **ertussen ligt is onbepaald**. De transferfunctie is afhankelijk van productieporces en omgevingsfactoren



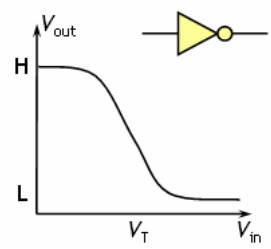
Bv. bij de invertor : wanneer we overgaan van V_H naar V_L op de ingang, zal de uitgangsspanning van V_L naar V_H moeten evolueren. Dit zorgt voor een bepaalde transferfunctie :

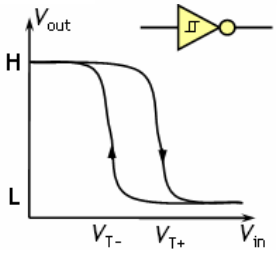
Schmitt-trigger-ingangen : ingangen voorzien van een **invertor met hysteresis**.



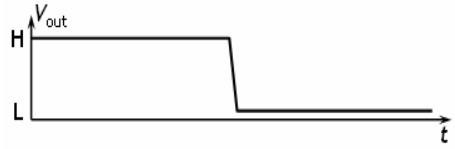
Stel dat je een ingangssignaal krijgt dat wat schommelt rond V_T (de het transitievoltage dat V_L en V_H scheidt) in het gebied van voltages waarvan de binaire waarde onbepaald is. Dit kan voorkomen bij het overschakelen van de ene binaire waarde naar de andere.

⇨ Een **gewone invertor** met een gewone transferfunctie zou dan eveneens een **schommelende uitgang** geven die vooral in het onbepaalde gebied zit





⇨ Een **Schmitt-trigger-ingang** zal nu werken met 2 transferfuncties en 2 schakelspanningen $V_{T+} > V_T$ en $V_{T-} < V_T$ met daartussen een hysteresis :



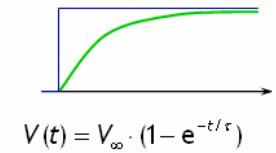
- wanneer we op V_L zitten zal de uitgang pas V_H worden wanneer Vin > V_{T+}
- wanneer we op V_L zitten zal de uitgang pas V_L worden wanneer Vin<V_{T-}

We zullen nu de uitgang pas overschakelen naar een ander niveau als de ingangsspanning voldoende sterk veranderd om zo de overgangsschommelingen weg te werken

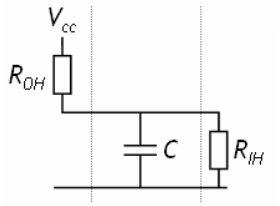
Tijdsgedrag

Wanneer draden naast elkaar liggen, ontstaat er een **kleine capaciteit** tussen hen. Daardoor zal bij **switch van voltage** de draad **even tijd nodig hebben** om deze verandering over heel de draad door te voeren. We zien dat er bij de verbinding van 2 poorten een capaciteit ontstaat tussen de verbinding en de neutrale draad. Hierdoor **duurt het altijd heel** even voor de **binaire verandering op de uitgang** van de eerste poort de **ingang van de 2de poort** heeft bereikt. Deze verandering is exponentieel en heeft een bepaalde tijdsconstante. We zien een **verschil** tussen de overgangen **V_L -> V_H** en **V_H->V_L**.

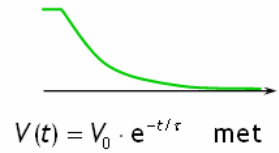
V_L -> V_H : **de stijgtijd**



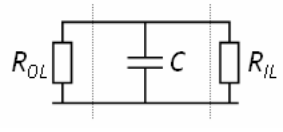
$V(t) = V_\infty \cdot (1 - e^{-t/\tau})$ met $\begin{cases} V_\infty = \dfrac{R_{IH}}{R_{IH} + R_{OH}} V_{CC} \\ \tau = \dfrac{R_{IH} R_{OH}}{R_{IH} + R_{OH}} C \end{cases}$



V_H -> V_L : **de daaltijd**



$V(t) = V_0 \cdot e^{-t/\tau}$ met $\tau = \dfrac{R_{IL} R_{OL}}{R_{IL} + R_{OL}} C$



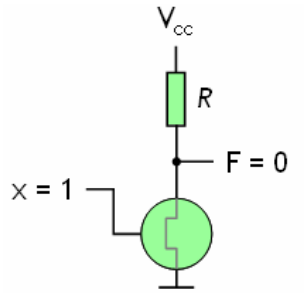
We willen uiteraard **zo snel mogelijke reactie** van het systeem. Daarom :

⇨ We willen bij een verandering in spanning deze zo snel mogelijk aan de 2de poort hebben :

- **uitgangsimpedantie *R*o zo klein mogelijk** (poort 1) omdat dan grote stromen kunnen vloeien naar de 2de poort (weinig spanning verloren gaat in de uitgang)
  ⇨ door die grote stromen ook groot vermogenverbruik (=nadeel van snelheid)
- **ingangsimpedantie *R*i zo groot mogelijk** (poort 2) om hier zo veel mogelijk spanningsval te krijgen (de stroom wordt hier zoveel mogelijk gebruikt)

⇨ ***C* (van draad & ingang) zo klein mogelijk** omdat dan de tijdsconstante kleiner is :
⇨ vermijdt lange verbindingen (hoe langer hoe groter C)

⇨ Daarom is **RTL-technologie** niet populair omdat hier **Ro groot** is : Bij RTL wordt in een poort de bovenste reeks transistoren PMOS vervangen door een weerstand R. Neem de invertor: Bij ingang 1 wordt V_{CC} via de weerstand kortgesloten op de neutraal en is de uitgang 0, bij ingang 0 wordt doorgang naar neutraal afgesloten en wordt V_{CC} door de weerstand geleid naar de uitgang die dan 1 wordt.



⇨ Dit geeft een **groot vermogenverbruik** daar er altijd stroom door de eerstand stroom en in geval van ingang 1 gewoon kortstluiting is
⇨ Hierdoor krijgt met ook **grote stijgtijden** (zie formules tijdsconstanten)

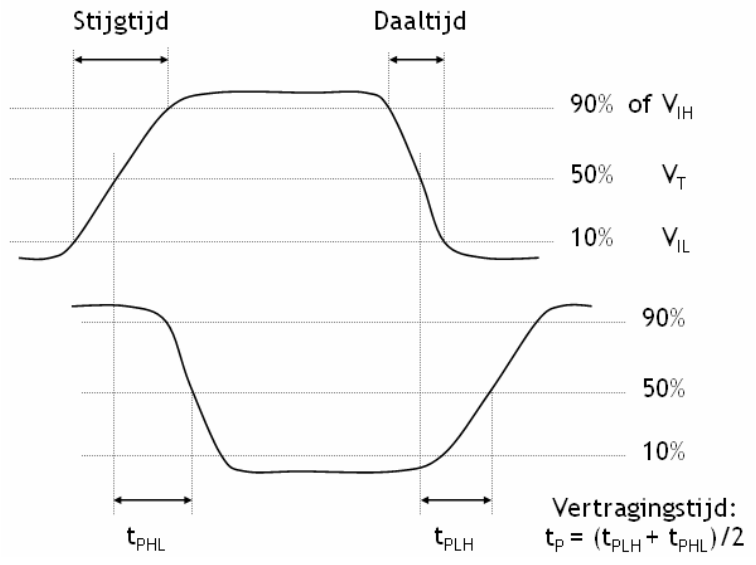
Ook de **poorten zelf** zorgen voor **vertraging**. Bij een verandering van de ingang van een poort zien we pas een tijdje later een verandering op de uitgang. Stel onze ingang van veranderd van V_L naar V_H en een tijdje later van V_H terug naar V_L.

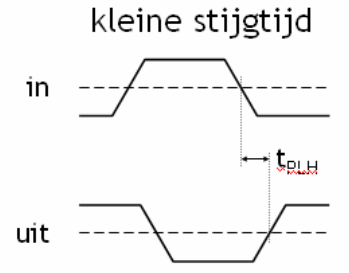
⇨ Uit het vorige weten we dat het **veranderen van de ingang geleidelijk gebeurt** :
de **stijgtijd** en de **daaltijd** = de tijd om te wisselen tussen 10% (V_{IL}) en 90% (V_{IH}) voltage.

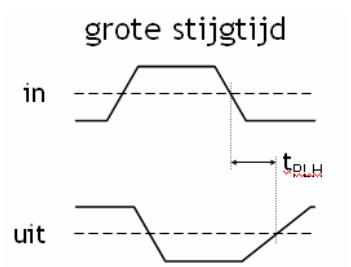
⇨ We zien dat pas een **vaste tijd later** de **uitgang veranderd**. Deze **vertragingstijd** tussen in- en uitgang is gedefiniëerd als de tijd tussen 50% verandering op ingang en 50% verandering op de uitgang. De vertragingstijden voor stijgen en dalen zijn **verschillend** :
⇨ vertragingstijd **t_{PHL} : V_H -> V_L**
⇨ vertragingstijd **t_{PLH} : V_L -> V_H**

De **algemene vertragingstijd** van een poort is dan het gemiddelde van deze 2 : **$t_P = (t_{PLH} + t_{PHL})/2$**

⇨ De **vertragingstijden** zijn **afhankelijk** van de **stijg-en daaltijden**. Hoe langer het duurt voor een signaal veranderd op de ingang, hoe langer het ook duurt voor deze verandering is doorgetrokken door de poort







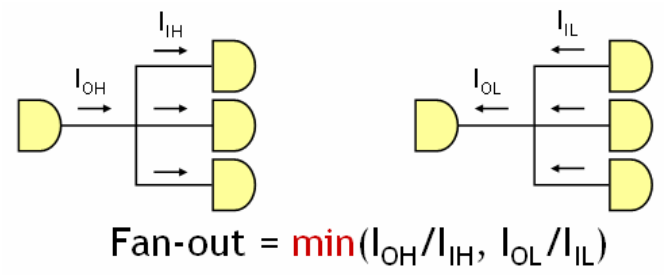
Fan-out

De **Fan-out** van een poort is het **max aantal ingangen** dat aan zijn **uitgang** kunnen gekoppeld worden.
⇨ de **stroomgedreven** technologieën (TTL, ECL, …) werken op basis van **stroom** die constant tussen de poorten vloeit en door weerstanden geconsumeerd wordt.

⇨ Wanneer de **uitgang V_H** is, vloeit er stroom **van de uitgang naar alle verschillende ingangen**, en we moeten er dus voor zorgen dat **elke ingang voldoende stroom** I_{IH} krijgt om een voltage te hebben dat in de juiste range ligt. De uitgang van de poort levert echter maar een maximale hoeveelheid stroom I_{OH}. De **maximale I_{OH}** en **minimale I_{IH}** zijn dus bepalend voor de fan-out

⇨ Wanneer de **uitgang V_L** is, vloeit er stroom **van de verschillende ingangen naar de uitgang**, hier moeten we zien dat de uitgang al deze stroom I_{OL} kan verwerken en dat elke ingang een minimale hoeveelheid stroom I_{IL} kan leveren. De **maximale I_{OL}** en **minimale I_{IL}** zijn hier dus bepalend voor de fan-out.

Alles tesamen is de **Fan-out** dus het **minimum** van **I_{OH}/I_{IH}** en **I_{OL}/I_{IL}**



⇨ de **ladingsgedreven** technologieën (**CMOS**) werken op basis van **spanning** :

⇨ In **statische toestand** (wanneer er niet geschakeld wordt) vloeit er **ongeveer geen stroom** $I_I \approx 0$.

⇨ Bij het **overschakelen** tussen de binaire niveau's V_L en V_H **vloeit er wel stroom** I_O doordat **capaciteiten** moeten op- of ontladen worden :

- **capaciteit** van alle **verbindingen** tussen uitgang en ingangen
- **ingangscapaciteit** van alle **aangestuurde poorten** (=C_{in} x fan-out)

De fan-out is hier gekoppeld aan de capaciteit van het geheel en dus aan de **maximale schakelfrequentie** : $I = C \cdot dV/dt = C \cdot f \cdot \Delta V \Rightarrow f = I/(C \cdot \Delta V)$

⇨ hoe kleiner C en ΔV, hoe groter de schakelfrequentie

bijv. voor Xilinx Virtex:

- 10 pF ingangscapaciteit, I_{Omax} = 20 mA, 0,8 pF/cm PCB, V_{cc} = 3,3 V
- Voor fan-out = 3 & 10 cm PCB-draden:
  C = 3 · 10 + 0,8 · 10 = 38 pF
  schakelfrequentie = I/(C · ΔV)
  = 20 mA/(38 pF · 3,3 V) = 160 MHz

Vermogenverbruik

Het **vermogenverbruik** van een chip is afhankelijk van de gebruikte **technologie** :

⇨ **TTL** : **stroom** vloeit **wanneer er niet geschakeld** wordt, dus ongeveer **altijd**. **P = Vcc*Icc** want ongeveer 10mW per poort is. Dit is redelijk veel als je bedenkt dat 1 miljoen poorten dan 10 kW verbruikt. TTL wordt dus alleen gebruikt bij hoge spanningen en stromen (voor bussen bv.)

⇨ **CMOS** : **stroom** vloeit **enkel als er geschakeld** wordt (overgaan van één toestand naar de andere) en is dus zo **afhankelijk van de schakelfrequentie** : **$P = VI = CfV^2$.** Nu is de periode dat er geschakeld wordt veel korter dan die waarin er niet geschakeld wordt, en dus is CMOS economischer.

bv. Virtex: P = 38 pF · 160 MHz · (3,3 V)2 = 66 mW/pin;
als de helft van de 200 pinnen gelijktijdig schakelen is er 6,6 W nodig

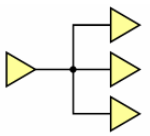
Het **vermogen** moet **zo beperkt mogelijk** gehouden worden omdat :

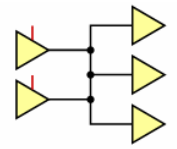
⇨ het vermogen **geleverd moet worden** door bv. een **battery** (hoe meer vermogen, hoe minder lang de batterij meegaat)

⇨ het volledige vermogen wordt **gedissipeerd als warmte** (bij te hoog vermogen zal de chip oververhitten en vereist dus koeling). We zien dat bij CMOS het **vermogen stijgt met de frequentie**. Wanneer we dus in frequentie (snelheid) willen toenemen, moeten we dat compenseren door V te verlagen (nauwkeuriger werken, C is moeilijk te verlagen)

3) Verbindingen

Busverbindingen





⇨ Traditioneel verbind men de **uitgang van één poort** met **verschillende ingangen** van anderen.

⇨ We kunnen ook **meerdere uitgangen bundelen in 1 bus**. Een bus is een verbinding waar uitgangen van meerdere poorten op dezelfde verbinding zijn gezet. We mogen echter **maar één uitgangssignaal (aansturing) tegelijkertijd op de verbinding** zetten en daarom moeten we elke uitgang **afblokken** met een **3-state-buffer**. Hiermee schakelen we telkens alle uitgangen van de bus af, op één uitgang na die dan de bus stuurt. We kunnen zo met de 3-state-buffers **kiezen welke uitgang** wanneer de bus stuurt : De bus heeft dus **telkens de waarde van slechts één van de uitgangen** !

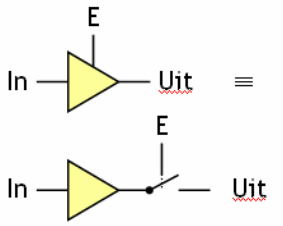
De tri-state-buffer is een buffer met :

⇨ **3** mogelijke **uitgangen** : **0, 1** en **Z** met Z de '**hoog-independante**' of '**zwevende**' uitgang waarbij de buffer eigenlijk **virtueel is afgekoppeld** van de bus.

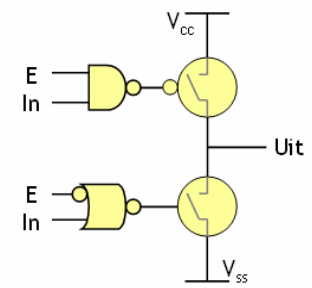
⇨ **2 ingangen** : de **enable ingang** en de **uitgang** van de poort die hij op de bus zet.

Wanneer **enable 0** is, is de uitgang van de buffer **Z** en is de uitgang dus **ontkoppeld** van de bus.

Wanneer **enable 1** is, wordt de uitgang van de buffer **gelijk aan de uitgang** van de poort en wordt die dus op de bus gezet. Alle andere buffers hebben op dat ogenblik enable 0, zodat er maar één uitgang tegelijk op de bus kan staan



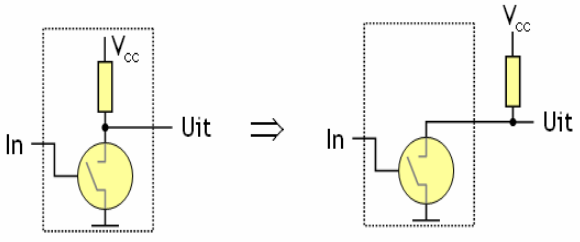
E	Uit
0	Z
1	In



Verbinden als poort (wired-logic)

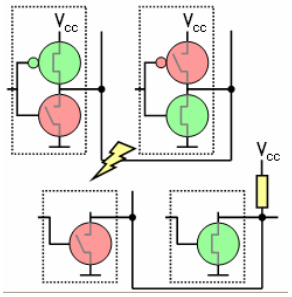
Open-drain- of **open-collector**-uitgangen is een manier om **complexe poorten** (met meerdere inputsignalen) te maken **zonder daarvoor standaard-poorten** te gebruiken. Dit is **TTL**-technologie, niet CMOS !

⇨ Het **principe** is dat we de **pull-up transistor** in de TTL of de R in RTL-technologie **weglaten** en vervangen door een **externe weerstand** die gebruikt wordt door alle open-drain-uitgangen. We kunnen nu complexe poorten maken waarbij elke ingang slechts 1 transistor aanstuurt i.p.v. 2 of waar er maar 1 pull-up weerstand is in plaats van i.p.v elke ingang zijn eigen pull-up weerstand.



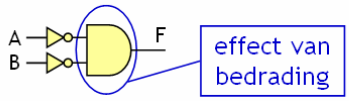
⇨ **nadelen** : de **externe weerstand** zorgt voor een **grote uitgangsimpedantie** hetgeen slecht is voor de stijgtijd (**grote stijgtijden**)
& Alle andere **nadelen van TTL** en **RTL**

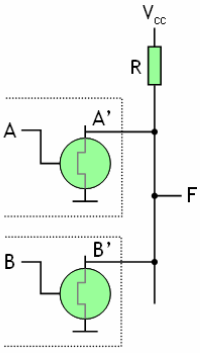
⇨ **voordelen** : - **uitgangen verbinden** geeft **geen kortsluiting**, hetgeen handig is wanneer we uitgangen samen op een bus willen zetten
- **wired logic**



Wired-logic : poorten gemaakt met open-drain-uitgangen. We kunnen nu een NAND-poort maken uit 2 transistors en een weerstand, waarbij we de uitgangen van de transistors gewoon samenbrengen als uitgang. Dit is een **Wired-AND**. De **transistors** fungeren als **invertoren**, en de **AND-poort** is simpelweg het **samenbrengen van de uitgangen** (draden samenbrengen als poort : vandaar de benaming wired-logic). Bemerk dat deze NAND een NOR is voor actief lage signalen en we dus op die manier een **wired-OR** kunnen maken.

A	B	A'	B'	F
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0





4) Implementatietechnologieën

Integratieniveau's

SSI: Small Scale Integration

- ⇨ < 10 poorten per verpakking
- ⇨ poorten direct verbonden aan pinnen
- ⇨ ontwerp op transistorniveau
- ⇨ gebruikt in ontwerpen op poortniveau

MSI: Medium Scale Integration

- ⇨ 10 – 100 poorten per verpakking
- ⇨ registers, optellers, pariteitsgeneratoren, …
- ⇨ ontwerp op poortniveau
- ⇨ gebruikt in ontwerpen op 'Register Transfer Level' (RTL)

LSI: Large Scale Integration

- ⇨ 100 – 10K poorten per verpakking
- ⇨ controllers, datapaden
- ⇨ ontwerp op RTL-niveau
- ⇨ gebruikt in ontwerpen op gedragsniveau

VLSI: Very Large Scale Integration

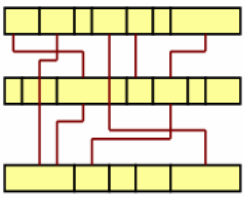
- ⇨ 10K – 1 Miljoen poorten per verpakking
- ⇨ geheugen, microprocessor, microcontroller, FFT
- ⇨ ontwerp op gedragsniveau : elke chip is gemaakt om 1 specifieke taak uit te voeren en voert die taak dan ook beter en efficiënter uit dan een programmeerbare chip met programmatuur
- ⇨ gebruikt in ontwerpen op systeemniveau

ULSI: Ultra Large Scale Integration ?

- ⇨ > 1 Miljoen poorten per verpakking
- ⇨ 2 microcontrollers, 20 DSP-processoren,
  16 Mbyte geheugen, 10 hardware-versnellers,
  1 Mgate FPGA, analoge interface, RF, …
- ⇨ ontwerp op systeemniveau : slechts 1 chip voor volledige toepassingen

We kunnen op 3 verschillende niveaus of manieren chips ontwerpen :

1) **Maatwerk & standaart cellen** : we ontwerpen een chip heel specifiek voor een bepaald doel en maken deze **zo efficiënt mogelijk** (qua tijd, responsie, vermogen, …). Dit is de **duurste optie**, omdat we heel het ontwerp doen voor 1 chip, die alleen gebruikt kan worden voor wat hij ontwerpen is. Heeft zeer grote dichtheid van poorten en is **zeer efficiënt** gemaakt.

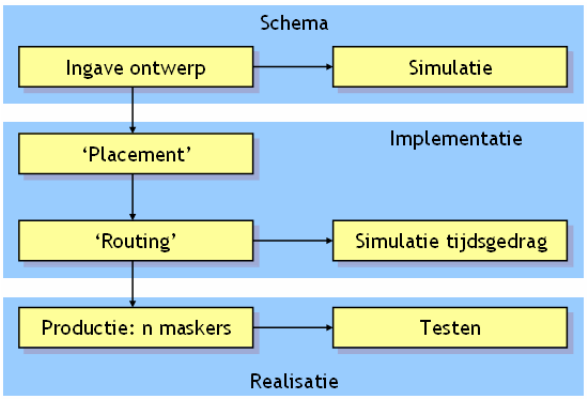


⇨ **Ontwerpproces** : De chip wordt **volledig ontworpen** op transistor- en bedradingsniveau : De **poorten** worden geïmplementeerd (met transistors) **op rechthoeken** (**celen**) naast elkaar in **rijen** van variabele lengte en standaart hoogt. Bovenaan de rij zitten de ingangen en onderaan de uitgangen. We hebben zo een **aantal rijen** achtereen, met daartussen **plaats voor de bedrading** (routing channels)

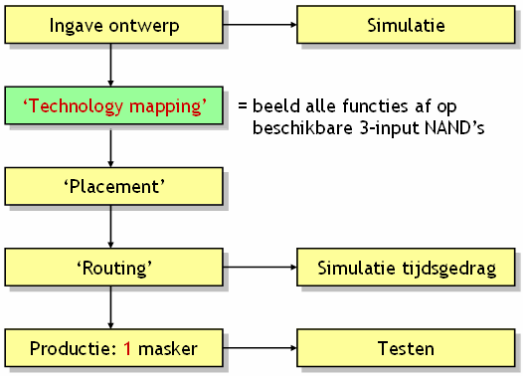
⇨ ideaal voor het werken met **bibliotheken** van **componenten** voor herbruik van ontwerpen : zo kunnen we **sneller complexe bouwblokken ontwerpen**. Elke **chipfabricant** heeft zijn eigen bibliotheek van componenten, aangepast aan zijn procestechnologie. Wanneer je dus een chip ontwerpt werk je met één bepaalde fabrikant en gebruik je die zijn bibliotheek. Het ontwerp van de chip zelf beperkt zich zo tot de
- allocatie van de **componenten** (placement)
- allocatie van de **bedrading** (routing)

⇨ chip moet wel **volledig herontworpen** worden wanneer de **technologie wijzigt**

Ontwerpschema :



Voor gate-arry's :



2) **Gate-Array's** : we maken een **standaart chip** met **heel veel dezelfde poorten** allemaal in dezelfde richting georiënteerd met **heel veel interconnecties** tussen die poorten.

(sea of gates, bv. allemaal NAND-poorten met 4 ingangen ofzo).

⇨ We moeten dan ons ontwerp nu omzetten naar een ontwerp met alleen maar NAND-pooten met 4 ingangen. Dit is **minder efficiënt** dan het vorige, maar het voordeel is dat we de standaard chips **goedkoper** kunnen krijgen (massa-productie). We moeten er alleen nog zelf ons ontwerp in implementeren met een laatste metallisatielaag voor de juiste verbindingen.

⇨ Het **ontwerpschema** is ongeveer **hetzelfde** als dat voor maatwerk, alleen komt er nu voor de placement een **technologie-mapping**, we we ons **ontwerp omzetten** naar een ontwerp met alleen het beschikbare soort poort. We moeten voor de productie ook maar 1 masker maken. (de verbindingen)

3) <u>**Field-programmable design**</u> : ook deze chip is een **standaard chip** die we met massaproductie **goedkoop** kunnen kopen. Deze chip heeft een standaard placement van verschillende soorten poorten en registers en standaard routing, waarbij we ons eigen ontwerp moeten programmeren.
Verschillende vormen :

⇨ ***<u>Gebruik makend van zekeringen</u>*** (PLA, PLD) : Alle mogelijke **verbindingen zijn gemaakt** en we moeten bepaalde **verbindingen verbreken** om ons eigen ontwerp te realiseren. Verbreken van verbindingen door met hoge stromen **zekeringen** (dunnere stukken draad) door te branden.

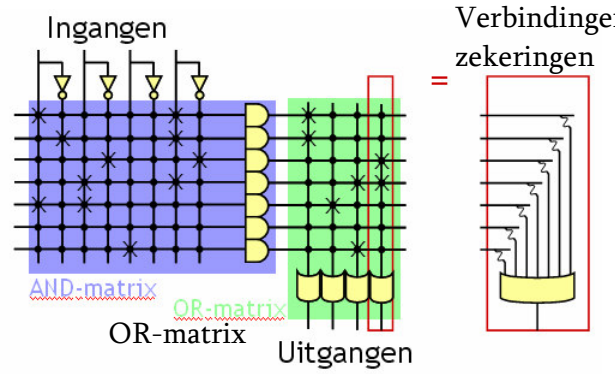
Dit is **irreversibel** : slechts bijprogrammeren mogelijk via extra zekeringen door te branden.

Voorbeelden :

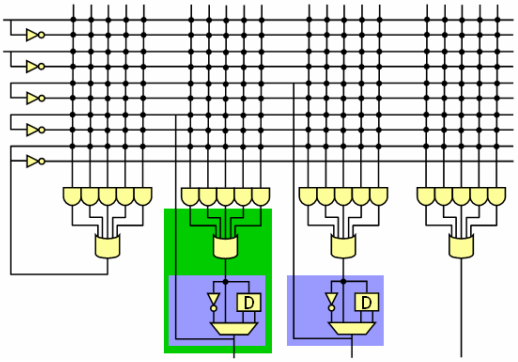
⇨ **PLA** : Programmable Logic Array : een reeks ingangen en hun inversen worden via een rooster naar AND-poorten gebracht, waarvan de uitgangen dan gecombineerd worden in een aantal OR-poorten tot uitgangen.

⇨ **PAL** : zelfde systeem, maar met vaste OR-matrix

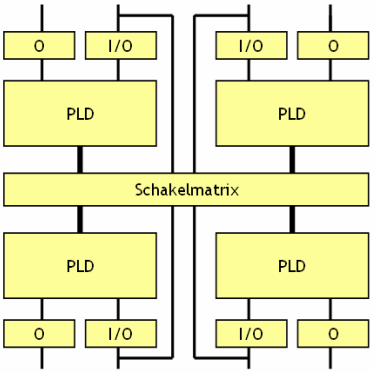
⇨ **PROM** : zelfde systeem, maar met vaste And-matrix (voor bv. adresdecoder)



⇨ **PLD** : Uitgebreiding van de PLA met macro-cellen : we zien verschillende groepen OR's die elk naar een AND gaan en terugkoppeling van een AND als een ingang.
Ook extra logica en flip-flops aan de uitgangen.



⇨ ***<u>EE- en flash-programmeerbaar</u>*** (CPLD) : De verbindingen tussen de componenten zijn **transistoren** waarvan het **poortniveau opgeladen** kan worden om de verbindingen te maken of te verbreken. **Herprogrammeren** is mogelijk, maar **traag en slechts een beperkt aantal keren**. Eigenlijk een soort van complexe PLD met extra verbindingen.

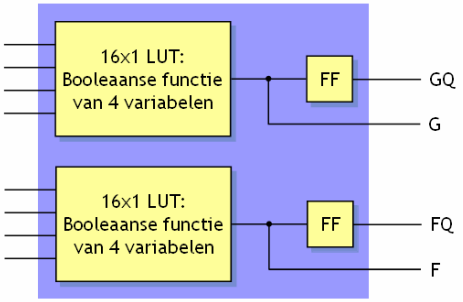


⇨ ***<u>Gebruik makend van SRAM</u>*** (FPGA : Field Programmable Gate-Array) : De verbindingen zijn **transistoren** waarvan het **poortniveau in een RAM-geheugen** opgeslagen wordt. Ook de poorten zijn minder standaard : booleaanse functies inladen. De chip is **volledig en vlot herprogrammeerbaar** : (statische of dynamische) herconfiguratie mogelijk. Telkens we de voedingsspanning aanleggen wordt heel de chip opnieuw geladen.

⇨ **Toepassingen** :
Voor prototypes & medium volumes (<100K stukken/jaar)
10 k logische cellen (> 5 M poorten) @ 400 MHz (in 2004)

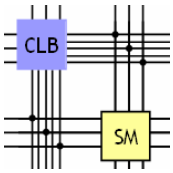
⇨ **FPGA** : volledige programmeerbare chip :

⇨ bestaat uit een **rooster** van **programmeerbare logische blokken** ; de ***CLB's***. (Configureable Logic Block)

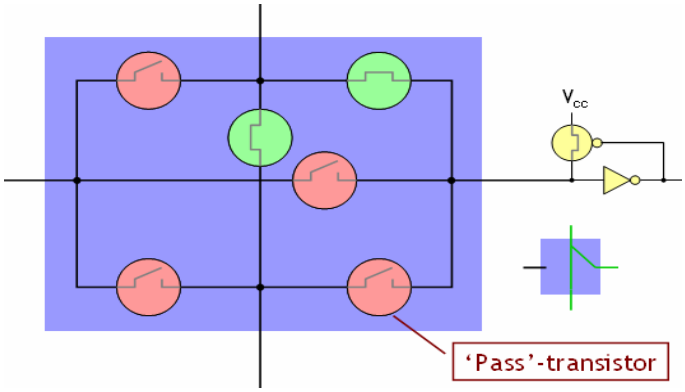


- **2 keer 4 ingangen** die elk in een **Logische eenheid** worden verwerkt. (LUT) In deze logische eenheid is hij het programmeren van de chip een **booleaanse functie** ingeladen die nu als poort toegepast wordt op de ingangen. De **LUT** (Look-Up Table) is een 32-Bit geheugen waarin de functie wordt gestoken en die daaruit wordt uitgelezen om toe te passen. De 32-Bit geheugen is zo bruikbaar als 32-bit SRAM geheugen
- De **uitgang** van elke logische blok verschijnt **rechtstreeks** als uitgang (G, F) uit de CLB en wordt ook in een **flipflop** (FF) **opgeslagen** (GQ, FQ)

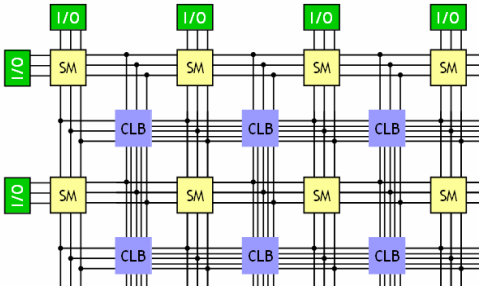
⇨ De CLB's zijn **verbonden** met elkaar via **lijnen** waarop ***SchakelMatrices (SM)*** staan om te bepalen welke CLB ins en outs met elkaar verbonden worden.



- Elk **interconnectiepunt** van 2 draden in de SM bestaat uit **6 transistors** die **geprogrammeerd** moeten worden tot het juiste ontwerp. **Elke mogelijke verbinding** van de 4 stukken draad is een combinatie van open en gesloten **transistors**.

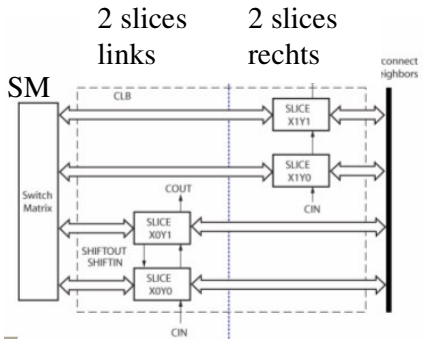


⇨ ***Directe verbindingen*** tussen naburige blokken
⇨ ***Lange lijnen*** om verafgelegen CLB's te verbinden
⇨ ***Globale kloklijnen*** zodat de klok overal even nauwkeurig en gelijk is
⇨ ***I/O blokken*** op de rand voor input en output naar buiten (gegevensinput, dataoutput, sturing)



⇨ specifieke **soorten** FPGA :

⇨ **Spartan-3** (gebruikt tijdens de labo's) : complexere FPGA met meer functies : het patroon is gelijkaardig aan de FPGA, maar er zijn verschillen :



⇨ De **logische eenheid** van de FPGA is de CLB. In de spartan-3 bestaat een **CLB uit 4 slices**, waarbij elke slices even groot is als een FPGA-CLB. (spartan CLB dus **4 keer groter** dan normaal) 2 slices staan **links** in de CLB en kunnen voor logica gebruikt worden, maar ook als RAM of als schuifregister. De 2 **rechtse** enkel voor logica.

⇨ Met de **2 rechtse slices** maken we de **Standaard logica** : 2 keer booleaanse functie van 4 variabelen met eventueel 2 flipflops en een functie van 5 met ev. 1 flipflop. (via F5MUX)

⇨ De **2 linkse slices** kunnen we gebruiken voor **extra logica** :

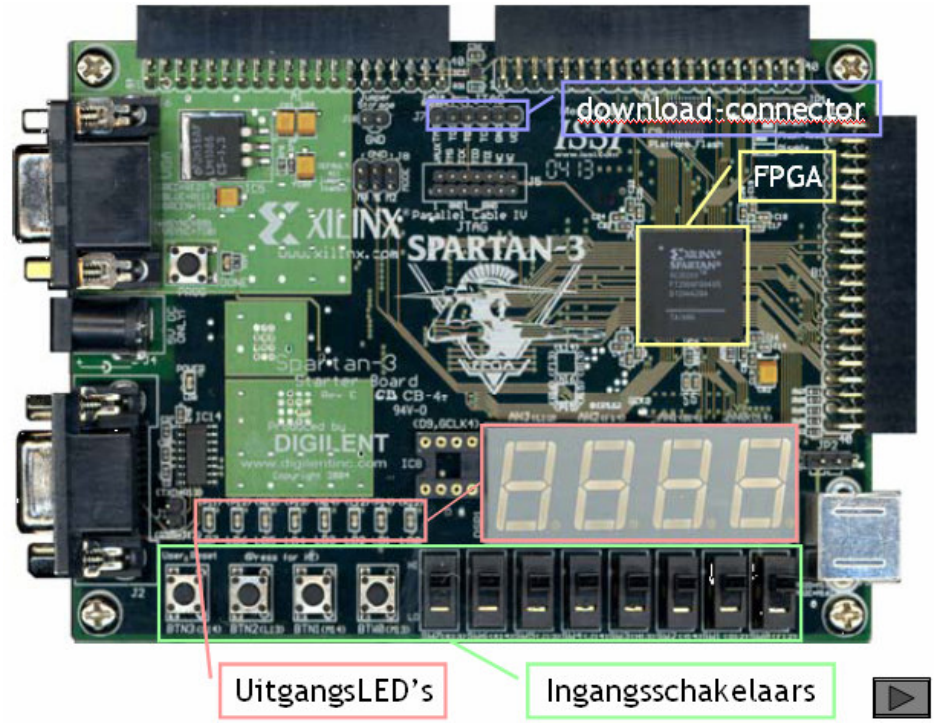
- carry logica (incl. AND en XOR)
- multiplexers
- 2 × 16 bit gedistribueerde RAM
- 2 × 16 bit schuifregister

⇨ de spartan-3 heeft **Klokgeneratie**, geheugen (**RAM**) en **vermenigvuldigers** zodat die niet meer met veel CLB's gemaakt moeten worden maar reeds efficient en snel ter beschikking zijn

⇨ **Andere** moderne FPGA's :

- specifieke transceivers of voor hoge snelheid (bv. Ethernet, 10 Gbit/s + CRC)
- DSP-functies (bijv. MAC)
- microprocessoren : (bv. 32 bit PowerPC met specifieke interface voor hardware-versnellers)

We gebruiken in de labo's **Xilinx ISE** omgeving voor het ontwerp van **programmatie** van de FPGA spartan-3's. We geven eerst het **ontwerp** in. Dat testen we dan op de **logische correctheid**. Wanneer de schakeling volledig is wat we willen, zullen we het ontwerp **technology-mappen** (omzetten naar de FPGA-mogelijkheden en de routing voorzien) . Dit ontwerp testen we dan in de tijd (**tijdsimulatie**) om te zien of er door de **delay-tijden** geen **onverwachte dingen** optreden. Dan kunnen we alles op de FPGA **uploaden** en kijken of het werkt.



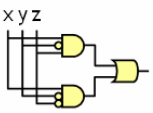
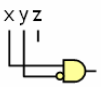
Hfdst 3) Combinatorische schakelingen

Minimalisering van booleaanse functies

1) Karnaugh-kaart

We zullen ons best doen om **booleaanse functies** zo veel mogelijk te **minimaliseren** (zo klein mogelijk te maken) : hoe **minder complex** de functie, hoe **minder poorten** nodig om ze te realiseren. Om :

- ⇨ de **kostprijs** te verlagen : hoe minder poorten hoe minder duur
- ⇨ **sneller** te werken : hoe minder poorten hoe sneller de responsie van het systeem :
  De **responsie** van een poort wordt uitgedrukt als de **tijd nodig** om het kritisch pad van het systeem te doorlopen. **Kritisch pad** = het pad met de **grootste vertraging** van ingang tot uitgang
  bv. F = *xy'z + xy'z'* is te vereenvoudigen tot *xy'(z + z')* en tot *xy'*

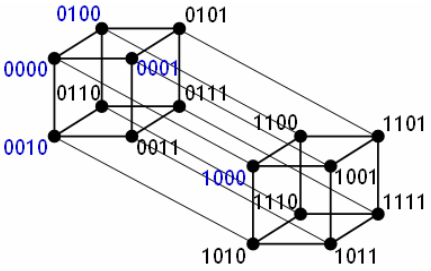
		
Kostprijs	(1+1) + (4+4) + 3 = 13	1 + 3 = 4
Vertraging	1 + 2,8 + 2,4 = 6,2	1 + 2,4 = 3,4

Hoe kunnen we een booleanse functie minimaliseren ?

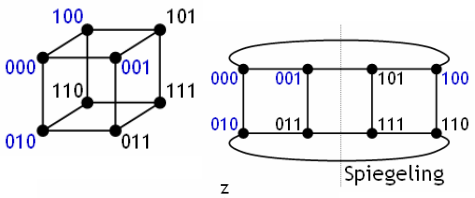
- ⇨ Via **manipulatie** van **Booleaanse uitdrukkingen**. Dit is echter zeer **moeilijk**: er bestaat geen methode om de opeenvolgende theorema's te kiezen die leiden tot de minimale oplossing. We zijn dus **nooit zeker** of het de meest minimale oplossing is, en de stappen die we moeten zetten moeten we zelf zien.
- ⇨ Een **voorstelling** gebruiken waarin **opvalt** welke ingang geen belang heeft :
  - ⇨ **Waarheidstabel** : We kunnen soms zien dat bepaalde uitgangen onafhankelijk zijn van bepaalde ingangen. Maar dit valt pas op wanneer deze gevallen echt onder elkaar staan.
    bv. Duidelijk dat F = 1 als x = 1 en y = 0, dus daar is F onafhankelijk van z. Dit is echter alleen maar duidelijk voor z omdat de lijnen voor z = 0 of 1 onder elkaar liggen

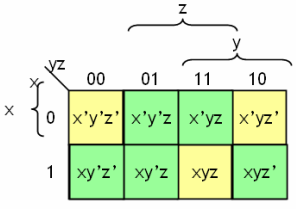
x	y	z	F	
0	0	0	0	-
0	0	1	0	-
0	1	0	0	-
0	1	1	0	-
1	0	0	1	xy'z'
1	0	1	1	xy'z
1	1	0	0	-
1	1	1	0	-

  - ⇨ **N-kubus** : N-kubus is een n-dimensionale kubus waarbij elke dimensie met 1 variabele overeen komt. Je kan hierin alle mogelijke combinaties van de variabelen voorstellen en hun onderlingen verbanden : vanuit elk punt (stelt 1 mogelijke combinatie van de variabelen voor) vertrekken n lijnen naar n punten die slechts 1 bit verschillen met de variabelen van dat punt.



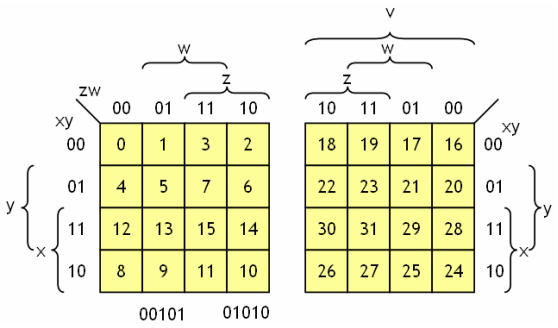
  - ⇨ **Karnaugh-kaart** : Een **2-dimensionele** voorstelling van een **n-kubus** : tot **n = 4 variabelen** is dit redelijk **triviaal** (we hebben dan een vierkant met 16 vakken), voor **n > 4** zullen we voor elke eenheid groter dan 4 deze vakken telkens 2 keer kopiëren zodat we 2^n vakken hebben, geschikt in vierkanten van 16 vakken.
    ⇨ **Elk vak** heeft dan **n buren** (dus ook buren in andere vierkanten van 16) die met dat vierkant slechts in **1 variabele verschillen**.



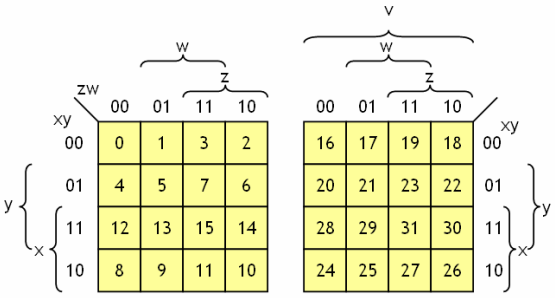


⇨ Wanneer er **meer dan 4 variabelen** zijn, liggen alle **buren niet meer fysisch naast elkaar**. Hierdoor wordt de kaart zeer **onoverzichtelijk** wordt naar mate er meer variabelen komen. We kunnen werken met **spiegeling** van de kaarten of door gewoon de kaarten te **herhalen** en de meest beduidende bit te veranderen (overal 16 optellen).

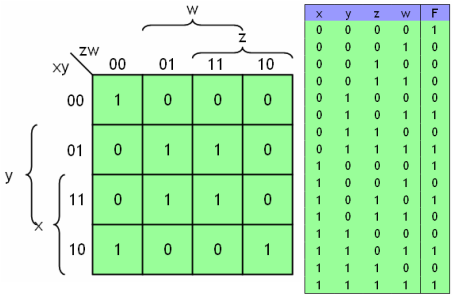
Gespiegelde kaarten



Herhaalde kaarten



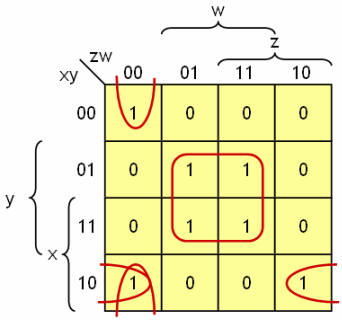
⇨ Het **invullen** van de Karnaugkaart gebeurt met de **waarheidstabel** : elk **vakje** komt overeen met een **rij** in de waarheidtabel. We vullen gewoon de uitkomst van de functie in een bepaalde rij in in het overeenkomstige vakje.



⇨ **Minimale oplossingen** voor de booleaanse functie vinden we dan door zo **groot mogelijke aaneensluitende gebieden** met oppervlakte machten van 2 met dezelfde waarde te selecteren en de overeenkomstige variabelen te combineren. Hiervoor zijn verschillende mogelijkheden.

Bv. F = x'y'z'w' + x'yz'w + x'yzw + xy'z'w' + xy'zw' + xyz'w + xyzw

Een minimale oplossing daarvoor is yw + xy'w' + y'z'w'



2) Minimalisering met Karnaugh-kaart

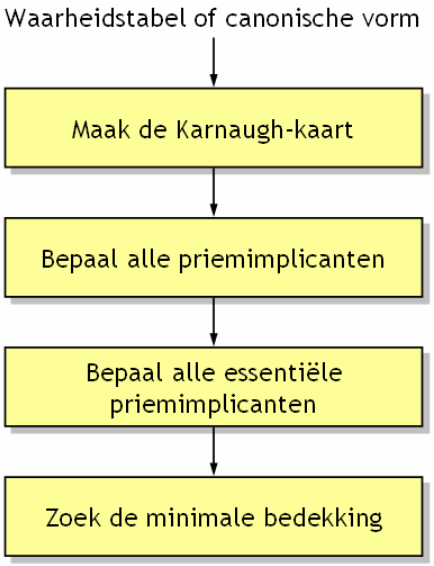
Minimale AND-OR realisatie

Minimalisatie van booleaanse functie tot **AND-OR** implementatie :

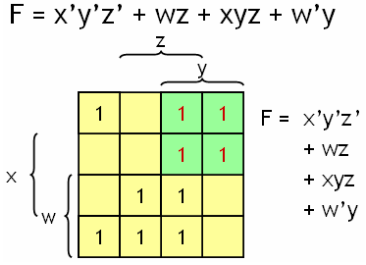
1) Karnaugh-kaart opstellen :

Vanuit de de canonische vorm of de waarheidtabel :

- voor elke **1** in de waarheidstabel zetten we een **1** in de karnaughkaart
- ook elke don't care (wanneer het niet uitmaakt of de uitgang 0 of 1 is omdat de ingangcombinatie toch niet voorkomt) geven we weer met een **kruis**
- de **0**en zijn **niet belangrijk**

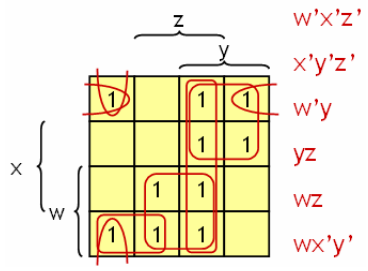


F = x'y'z' + wz + xyz + w'y



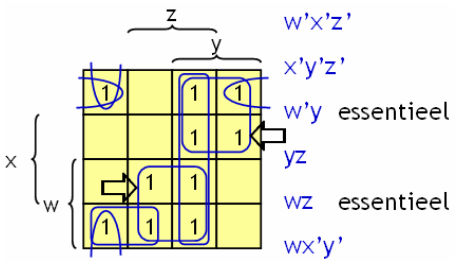
2) Priemimplicanten bepalen :

We bekijken **elke 1** in de kaart en bepalen voor elke 1 de **grootst mogelijke subkubus** die die minterm bevat. (kunnen meerdere kubussen zijn als ze even groot zijn). We **noteren elke priemimplicant.** De subkubussen kunnen ook **rechthoeken** zijn, kunnen **over de randen** gaan naar alle **buren** van het vakje, en het aantal vakjes in de subkubus is altijd een **macht van 2**.



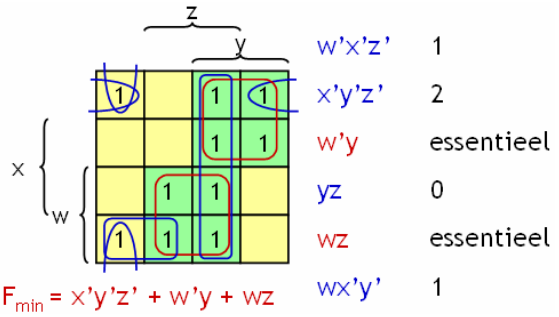
3) Essentiële priemimplicanten bepalen :

We zoeken nu alle 1'tjes (**1-mintermen**) die **slechts in 1 priemimplicant** voorkomen : dit zijn de **essentiële priemimplicanten** die zeker mee opgenomen moeten worden in de minimale functie.



4) Minimale dekking zoeken :

Zoek de **kleinste set** (zo weinig mogelijk) van **zo groot mogelijke** priemimplicanten die **alle 1-mintermen** omvat : We beginnen deze set als **eerst met de essentiële priemimplicanten** en daarna voegen we telkens een priemimplicant toe die **zoveel mogelijk onbedekte 1-mintermen** bevat tot dat we alle 1-en hebben.



⇨ Dit is een **gulzige** ('greedy') strategie: we kiezen hier telkens de **beste oplossing** zonder rekening te houden met de gevolgen op **toekomstige keuzes**. Daardoor is dit niet noodzakelijk de meest optimale keuze voor het gehele systeem omdat we soms **eerder geïmplementeerde functies kunnen herbruiken**.

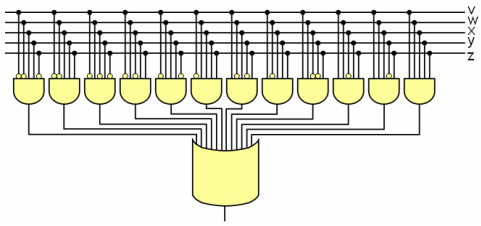
⇨ Soms is het beter om niet met AND-OR te werken : AND-OR is een **2-laags systeem** waarbij de 1ste laag AND is en de 2de laag OR. We kunnen **soms** echter met een **3-laags systeem een goedkopere en/of snellere oplossing** vinden

Nog een voorbeeld :

v	w	x	y	z	F
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	1
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	1	0
0	1	1	1	0	1
0	1	1	1	1	1

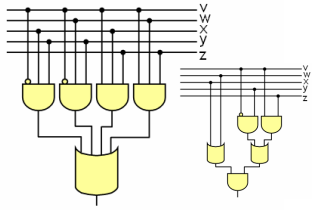
v	w	x	y	z	F
1	0	0	0	0	0
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	0	1	1
1	0	1	1	0	0
1	0	1	1	1	1
1	1	0	0	0	0
1	1	0	0	1	1
1	1	0	1	0	0
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	0	1	1
1	1	1	1	0	0
1	1	1	1	1	1

$F = \Sigma(6,7,10,11,14,15,21,23,25,27,29,31)$

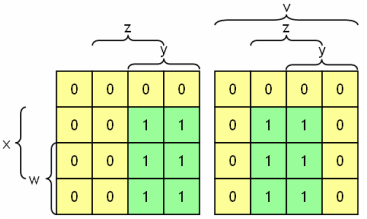


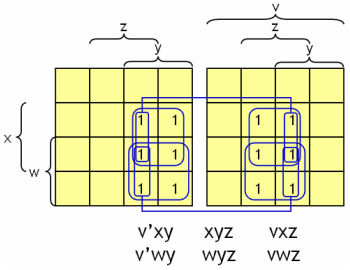
Kostprijs = 5×1 + 12×6 + 1×13 = 90
Vertraging = 1 + 3,6 + 6,4 = 11

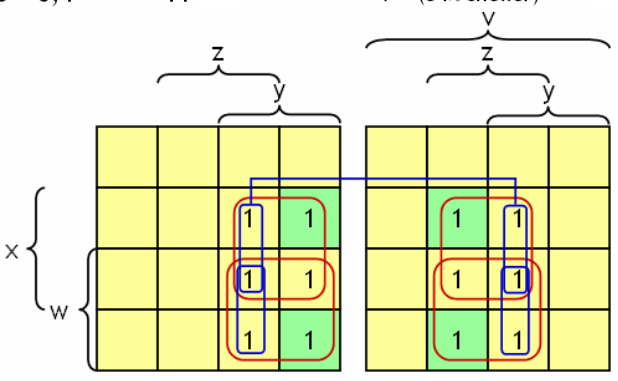
$F_{1min2} = v'xy + v'wy + vxz + vwz$



Kostprijs = 1×1 + 4×4 + 1×5
= 22 (76% goedkoper)
Vertraging = 1 + 2,8 + 3,2
= 7 (34% sneller)



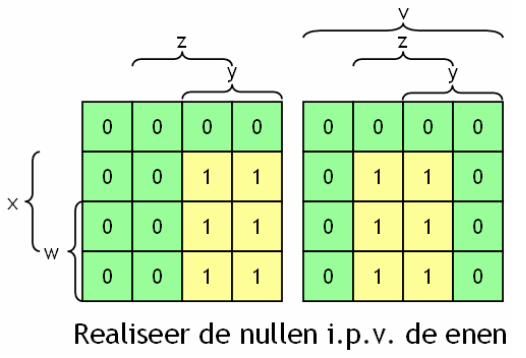


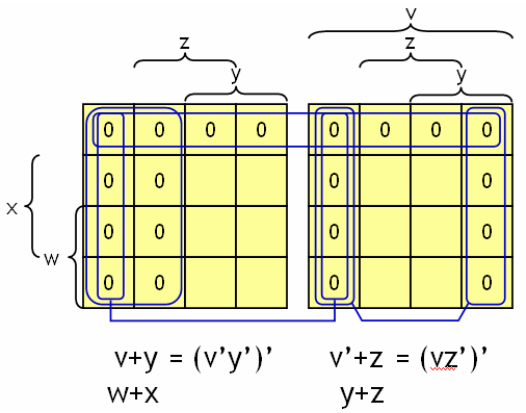


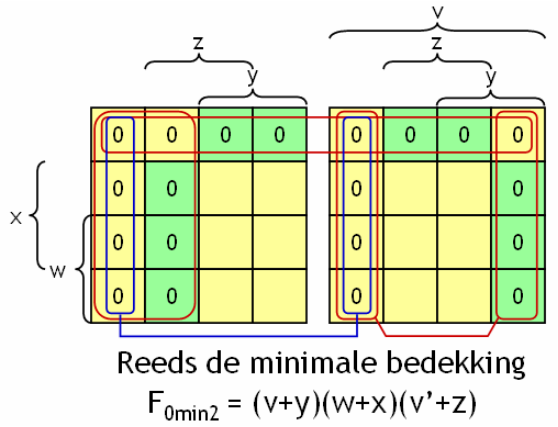
Reeds de minimale bedekking
$F_{1min2} = v'xy + v'wy + vxz + vwz$

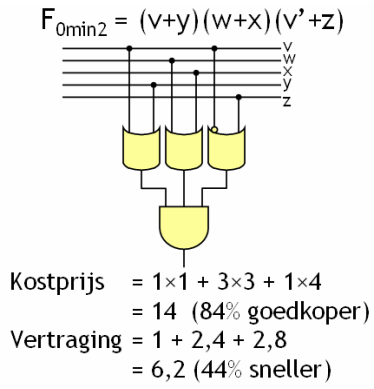
Minimale OR-AND realisatie

Deze methode is **ongeveer hetzelfde**, alleen zullen we hier de **0-mintermen** realiseren in een functie in plaats van de 1-mintermen. We beginnen dus met alle **0en en de don't cares** te zoeken, daarvan de priemimplecanten en essentiële priemimplicanten te zoeken, We realiseren hier een functie als Product van sommen. Bij de AND-OR implementatie was dit een som van producten.









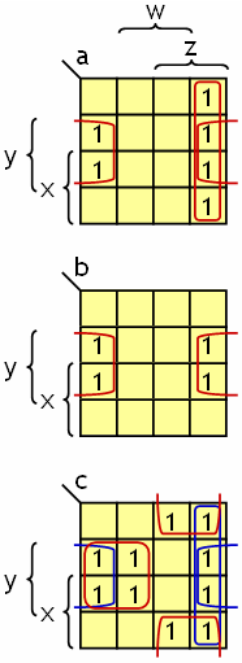
We kunnen dus op **verschillende manieren** proberen om de functie te minimaliseren. Elke methode levert **een voor zijn methode minimale functie**, en we kunnen dus elke methode overlopen en zien welke de best is. Meestal komt **snelheid** ook **ten koste van prijs** en omgekeerd.

Realisatie	Kost	Rel. kost	Vertraging	Rel. vertraging
Som van 1-mintermen	90	100%	11	100%
Minimale AND-OR	22	24%	7	64%
3-lagen	16	18%	8,2	75%
Minimale OR-AND	14	16%	6,2	56%

compromis grootte ↔ snelheid

3) Meerdere uitgangen

Wanneer we een **ontwerp** met **meerdere uitgangen** moeten realiseren, zullen we **meerdere booleaanse functies** hebben van de ingangsvariabelen die **elk één van de uitgangen** afbeelden. We moeten dan voor **elke uitgang een verschillende Karnaugh-kaart** maken.



⇨ We zullen bij de **minimalisatie** zoals gewoonlijk alle **essentiële priemimplicanten** bepalen. Daaruit selecteren we nu de priemimplicanten die we zullen gebruiken voor de **minimale functie**. We zullen bij deze selectie **zoveel mogelijk gebruik** maken van die priemimplicanten die **al essentieel zijn voor een andere functies**. Deze zijn **al gerealiseerd** (voor die andere functies) en kunnen dus **gratis** herbruikt worden.

Bv. bij het maken van b zien we dat yw' reeds essentiëel is voor a : herbruiken

⇨ Wanneer we de **keuze** hebben tussen **verschillende reeds gerealiseerde priemimplicanten**, zullen we die pakken die **in het kleinst aantal fucnties voorkomt**. Dit doen we om de **fan-out** van elke poort zo **laag** mogelijk te houden.

Bv. bij het maken van c zien we dat yw' reeds geïmplementeerd is voor a en b en zw' reeds voor a. We hebben hier keuze tussen het hergebruiken van yw' en zw'. We zullen kiezen voor zw' omdat yw' reeds door meer functies wordt gebruikt. (fans-out van yw' zou 3 worden, waar ze nu 2 is)

⇨ Soms kan het ook **voordelig** zijn om **niet-priemimplicanten** te gebruiken wanneer we dan **meer implementaties kunnen herbruiken** (dit is wel een trail-and-error-methode). De mogelijkheden stijgen wanneer er **meer lagen poorten** toegelaten zijn.

$$\begin{cases} a = yw' + zw' \\ b = yw' \\ c = yz' + y'z + zw' \end{cases}$$

4) 'Don't care' condities

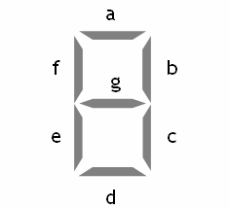
Soms is een booleaanse functie voor **bepaalde ingangscombinaites niet gedefinieerd**. Dit komt voor wanneer die **ingangscombinaties nooit voorkomen** tijdens de werking van het systeem. Het maakt dan niet uit welke waarde de functie heeft en we kunnen daar handig gebruik van maken om die **waarde dan zelf te kiezen**. We kunnen dit gebruiken om **grotere priemimplicanten** te maken. We stellen de 'don't cares' voor als kruisjes in de karnaughkaarten en kunnen dan grotere gebieden van 1 of 0 maken waardoor de booleaanse functie kleiner wordt.

Bv. we beelden een **BCD-code** af op een **7-segment display**. BCD-code levert **4 ingangsvariabelen** die binair de waarde van de cijfers 0 tot 9 voorstellen. We zoeken nu de **7 booleaanse implementaties** om de 7 segmenten aan te sturen met de 4 BCD-ingangsvariabelen. We kunnen echter met die 4 BCD-bits ook de getallen 10 tot 15 vormen, maar die zullen nooit als ingang verschijnen omdat we die toch niet kunnen afbeelden op de 7-segment display. Voor die ingangscombinaties maakt het dus niet uit wat de uitgangen zijn, aangezien de ingangen toch nooit in die combinatie zullen voorkomen.

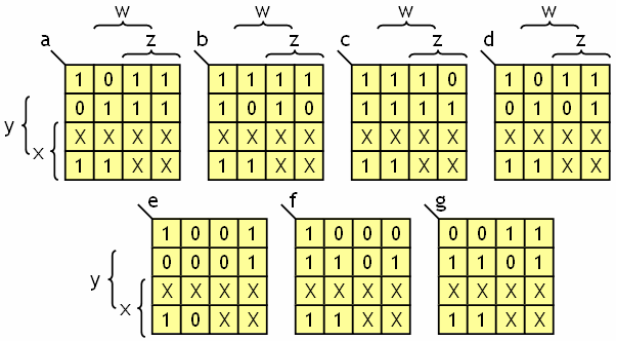
7-segment display

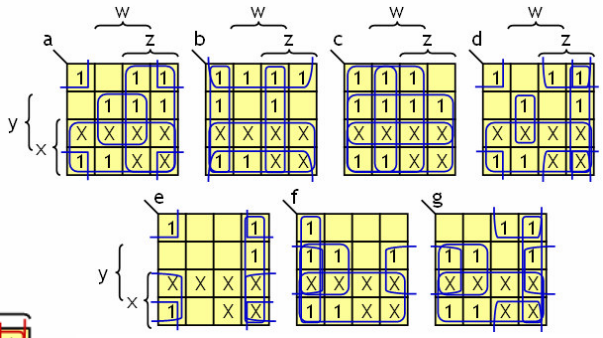
Uitwerking van de 7 implementaties van de segmenten in functie van de 4 ingangen.

BCD → 7-segment



x	y	z	w	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1
1	0	1	0	X	X	X	X	X	X	X
1	0	1	1	X	X	X	X	X	X	X
1	1	0	0	X	X	X	X	X	X	X
1	1	0	1	X	X	X	X	X	X	X
1	1	1	0	X	X	X	X	X	X	X
1	1	1	1	X	X	X	X	X	X	X

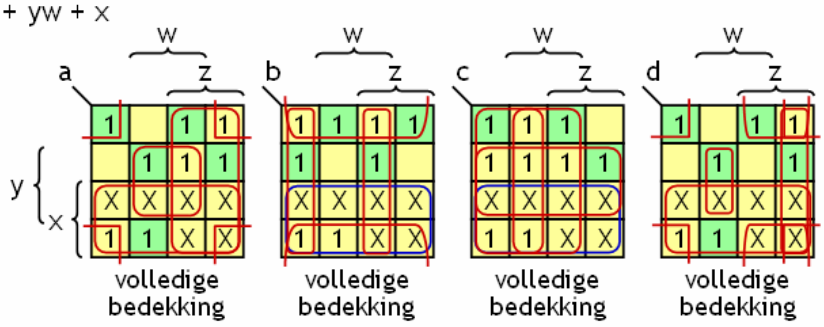




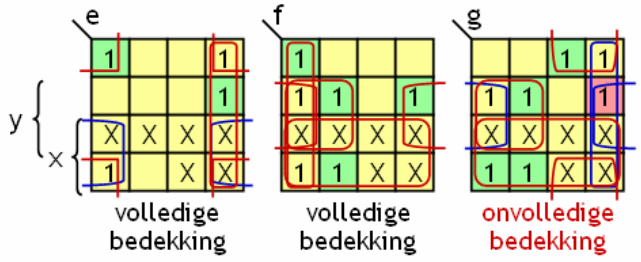
a = y'w' + z + yw + x

b = y' + z'w' + zw

c = z' + w + y

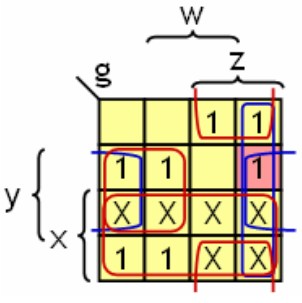


d = y'w' + y'z + yz'w + zw' + x



e = y'w' + zw'

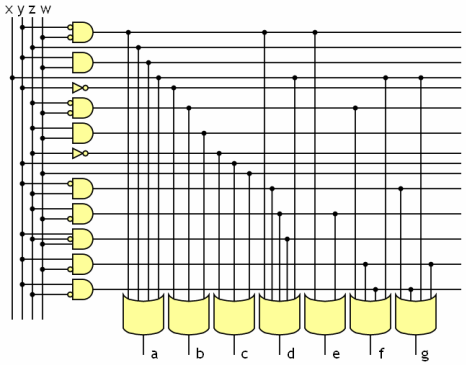
f = z'w' + yz' + yw' + x



g = y'z + yz' + yw' + x

Keuze van de priemimplicant die de overblijvende minterm realiseert:

- Selecteer alle priemimplicanten die de minterm realiseren en die al essentieel zijn voor een andere functie
  - zw' reeds essentieel voor d of e
  - yw' reeds essentieel voor f
- Kies de implicant die in het kleinst aantal functies voorkomt
  - zw' reeds in d en e
  - yw' reeds in f



5) Quine-McCluskey

Deze methode om minimale booleaanse functies te zoeken maakt **gebruik van tabellen** om een optimale oplossing te vinden. Het is een **vaste manier** van werken die **makkelijk te implementeren is voor CAD-software**. Echter **moeilijk met de hand** te doen (veel rekenwerk).

⇨ De basis is hetzelfde : we zoeken **priemimplicanten** : eerst gewoon alle 1'tjes, dan alle mogleijke groepjes van 2, dan van 4, enz. tot we alle priemimplicanten hebben. Daarna steken we die in een tabel en kijken we welke 1'tjes **maar door 1 priemimplicant** wordt bedekt. Dit zijn de **essentiële**. Daarna gaan we gewoon **alle mogelijke combinaties** af en kijken wel welke de meest optimale is.

Impact van technologie

1) Gate-Array (NAND, NOR)

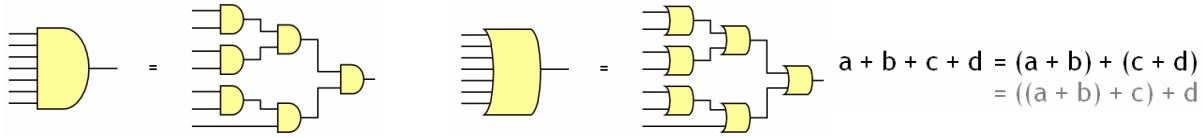
Wanneer we een ontwerp willen realiseren op een **chip met gate-arrays**, moeten we dus ons ontwerp **mappen op de technologie** waarmee de **chip** werkt. In een gate-array-chip hebben we een heleboel **poorten** van **éénzelfde type** met **éénzelfde aantal ingangen** : **NAND-poorten** of **NOR-poorten** met **m ingangen** (meestal 3 of 4 ingangen). We moeten ons ontwerp omzetten naar een ontwerp met enkel die soort poort en dat aantal ingangen.

⇨ voor **m-NAND-poorten** zullen we ons ontwerp baseren op **INV-AND-OR** ontwerp, geminimaliseerd met **1-mintermen**

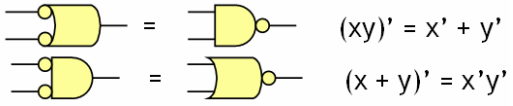
⇨ voor **m-NOR-poorten** zullen we ons ontwerp baseren op **INV-OR-AND** ontwerp, geminimaliseerd met **0-mintermen**

Het **aanpassen** van het ontwerp aan de technologie gaat als vorlgt :

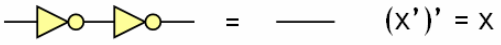
1) ***Decompositie*** : vervang de n-input AND-poorten door m-input AND-poorten (herleid het aantal ingangen van elke poort tot m). Dit is meestal het opslitsen in meerdere poorten :



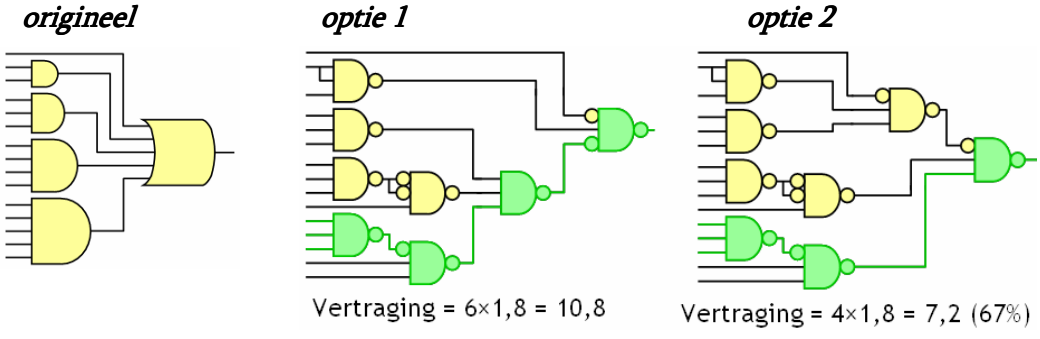
2) ***Conversie*** : vervang AND door NAND (of door OR door NOR) : wetten van Morgan



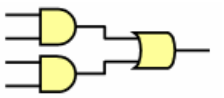
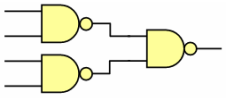
3) ***Optimalisatie*** : eliminatie van dubbele negaties

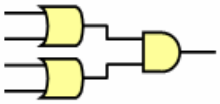
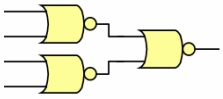


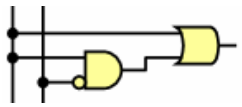
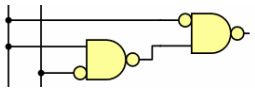
4) ***Optimalisering vertraging*** : we trachten alle vertragingen van ingang naar uitgang gelijk te maken. Zo is het kritische pad zo kort mogelijk. Bv. onderstaand ontwerp implementeren met 3-input-NAND-poorten : de aangeduide weg is het kritische pad :

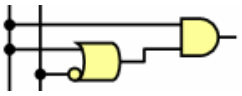
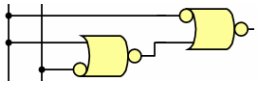


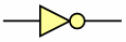
⇨ Enkele **voorbeelden** :

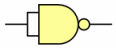
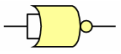
1)  wordt 

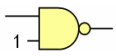
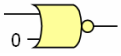
2)  wordt 

3)  wordt 

4)  wordt 

⇨ Voor de realisatie van een **invertor**  zijn er 2 opties :

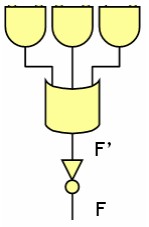
⇨  en  : nadeel is dat hier de sturende poort 2 keer belast wordt in plaats van 1

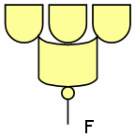
⇨  en  : nadeel is dat we hier langere bedrading nodig hebben om 1 en 0 te leveren

2) Componentenbibliotheek (AOi, OAI, …)

Wanneer we nu zelf een chip ontwerpen op een **zo efficiënt mogelijke manier**, (dus op poortniveau zelf chip maken) dan beschikken we over **alle soorten poorten** die de **chipfabrikant** in zijn technologie aanbiedt. We kunnen hier gebruik maken van alle poorten die zich in de **bibliotheek** van componenten van de producent bevindt. Bv. voor ASIC's hebben we nu naast NAND's en NOR's ook **OAI's** en **AOI's** ter beschikking. Deze zijn klein en snel.

⇨ **AOI's en OAI's** zijn **speciale specifieke poorten** die ondersteund worden door sommige bibliotheken (ASIC) en die gebruikt kunnen worden bij het specifieke ontwerp van chips. We impementeren hier eigenlijk **2 lagen poorten en een invertor in één poort**. Het voordeel is dat de AIO-poort veel goedkoper en sneller is dan het 3-lagen-systeem.





Dit is dus symbool voor één poort !!!

⇨ bv. een **3-breed, 2-input-AOI** is één poort met 6 ingangen die dezelfde booleaanse uitdrukking heeft als een systeem van 3 lagen met eerst 3 AND-poorten met 2 ingangen, dan 1 OR-poort met 3 ingangen en dan een invertor.

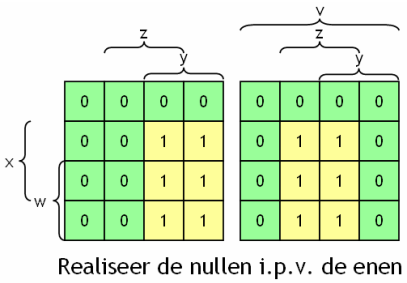
- De AOI kost **6** (6 ingangen) terwijl het 3-lagen-systeem **10** kost (= 3x2 + 3 + 1)
- De AOI heeft vertraging **3** (=0.6 + 6x0.4), het 3-lagen-systeem vertraging **7** (=3x1.4 + 1.8 + 1)

⇨ **Technology-mapping** komt nu neer op het **groeperen van poorten** in ons ontwerp zodat **elke groep vervangen wordt door 1 specifieke component** in de bibliotheek. Dit is complexer dan bij gate-arrays. We gebruiken **verschillende methodes** voor kleine en grote functies :

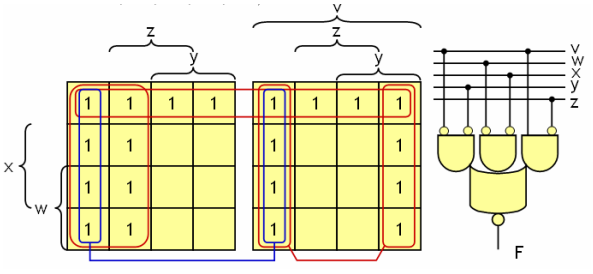
⇨ ***Kleine functies*** : we realiseren de **inverse** functie met AND-OR of OR-AND en we zetten daar een **invertor** achter, zodat we dit **3-lagensysteem** kunnen stoppen in **1 AOI of IAO**. Deze ene poort plus invertor zal **goedkoper** uitkomen dan het 2-lagensysteem AND-OR of OR-AND. We **vervangen gewoon elke 0 door 1** en omgekeerd en **optimaliseren** met karnaugh. We krijgen dan als uitgang F', en dus zetten we op het einde gewoon een invertor om terug F te krijgen.

bv. bij het voorbeeld van optimalisatie kregen we na minimaliseren een OR-AND implementatie met kost 14 (16%) en vertraging 6,2 (56%).

Wanneer we nu echter de **inverse van de functie** minimaliseren kunnen we het resultaat in één grote AOI met 6 ingangen stoppen. Resultaat : kostprijs van 11 en een vertraging van 4.



Realiseer de nullen i.p.v. de enen

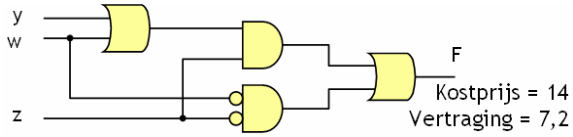


Kostprijs = 5×1 + 6 = 11 (12%)
Vertraging = 1 + 3 = 4 (36%)

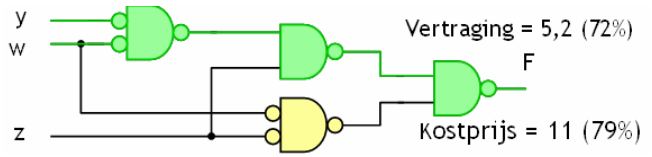
Σ(6,7,10,11,14,15,21,23,25,27,29,31) ⇒
Σ(0,1,2,3,4,5,8,9,12,13,16,17,18,19,20, 22,24,26,28,30)

⇨ *Grote functies* : we volgen een vast systeem :

1) Realiseer de functie als **AND-OR** of als **OR-AND** schema.

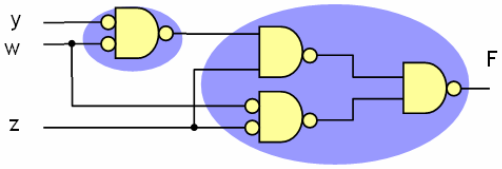


2) Transformeer het schema naar **NAND** of **NOR**, omdat dit de snelste poorten zijn. Zo hebben we een ontwerp dat gegarandeerd zo snel mogelijk is.

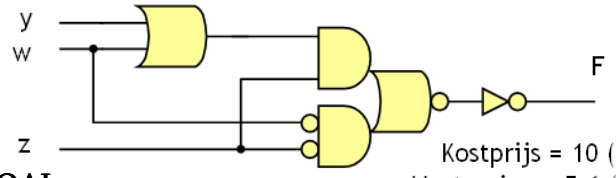


3) Bepaal het **kritische pad** (het pad met de langste delay tussen input en output). Door de realisatie met **NAND en NOR** is dit pad het **kortst mogelijke pad** en we willen dus dit pad **behouden**. (Wat we verder veranderen mag niet langer worden dan dit pad)

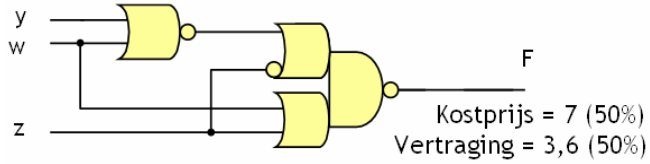
4) Vervang zoveel mogelijk **groepen van 2 lagen poorten** op het **kritische pad** door **AOI of OAI**. Hierdoor verlaagt de kost zonder dat het kritische pad langer wordt.



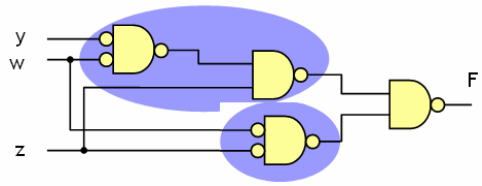
Wordt met <u>AOI</u>



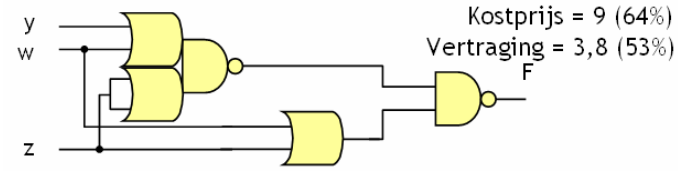
Wordt met <u>OAI</u>



We kunnen ook andere groepen van poorten op het kritische pad vervangen :



Wordt met <u>OAI</u>



Bij deze laatste optie is na het vervangen van de bovenste groep poorten een **nieuw kritisch pad ontstaan** !!! Hierop vervangen we dan de 2-de (groep) poorten.

5) Vervang zoveel mogelijk **groepen van 2 lagen poorten buiten het kritische pad** door **een component uit de bibliotheek.** Dit geeft een lagere kost.

3) FPGA

Wanneer we het vorige toepassen op een FPGA, zien we dat wel **alles moeten opsplitsen** in **subschakelingen van 4 of 5 variabelen**. (De CLB's hebben alleen functies van 4 of 5 variabelen). We hebben hier geen AOI's en OAI's.

⇨ Voor **prototypes** bij de FPGA of ASIC gebeurt deze aanpassing **automatisch**, voor het **eindproduct** kan **handmatig** optimaliseren voordelig zijn.

Tijdsgedrag

1) Hazerds vermijden

Soms zien we dat een bepaald **signaal plots onverwacht veranderd voor een korte tijd** (naar een fout niveau) en **dan terug juist** wordt. Zo een bijkomende niveauverstoring (‘**glitch’**) noemen we een hazerd.

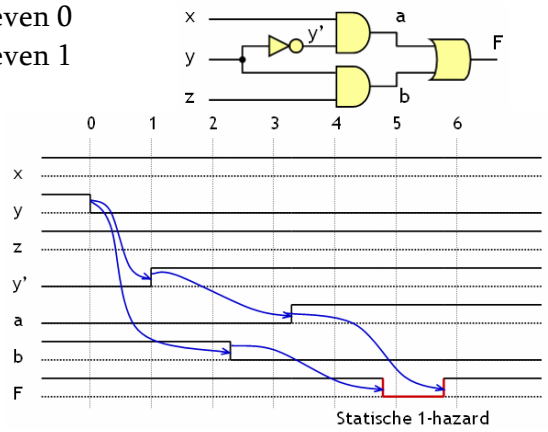
⇨ **Hazerds** zijn te wijten aan het het feit dat **verschillende poorten verschillende vertragingen** met zich mee brengen. (het tijdsgedrag van de schakeling dus) Wanneer een signaal door verschillende poorten langs **verschillende paden** moet gaan en **daarna weer samengevoegd** wordt, is er een probleem. We zien dan dat de **verandering op het ene pad** soms **voor zit** op de andere (minder poorten en/of snellere poorten tegengekomen waardoor het **sneller vooruit** gaat). Op de plaats waar de signalen dan **weer samenkomen** hebben we op een bepaald moment **2 verschillende waarden voor hetzelfde signaal**, en zal de **uitgang dus verkeert** zijn. Het probleem lost zich vanzelf op wanneer de 2de verandering de 1ste heeft ingehaald. Aangezien bij deze hazerds de uitgang 2 keer nodeloos omgeschakelt wordt, is dit verspilling van vermogen.

Bv. invertor : stel we takken variabele a voor de invertor af. Wanneer we a veranderen naar a’, zal de invertor relatieve tijd 1 nodig hebben voor de uitgang veranderd van a’ naar a. We zien dus dat tijdens die relatieve tijd 1 de uitgang nog a’ is en de ingang ook a’, waardoor zowel de ingang als de uitgang van de invertor gedurende periode 1 dezelfde waarde hebben.

⇨ ***Statische hazard*** : constant niveau vertoont gedurende korte periode verstoring :

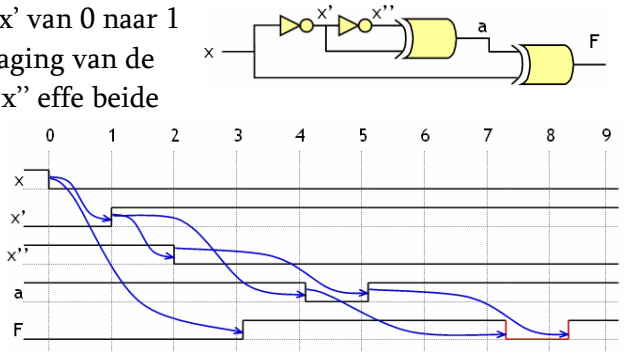
- statische 1-hazard: uitgang moet 1 blijven maar wordt even 0
- statische 0-hazard: uitgang moet 0 blijven maar wordt even 1

bv. alle ingangen zijn 1, en we veranderen y van 1 naar 0. Hierdoor zal eerst b veranderen na 2.2 (wachttijd van AND). Dan verandert a na 1+2.2 (wachttijd van invertor+AND). Door de invertor zit er nu een tijdsverschil van 1 tussen veranderingen in a en b. We zullen nu zien dat F 2.2 na de verandering van b 0 wordt, omdat dan a en b 0 zijn. 2.2 na de verandering van a en dus 1 na het 0 worden van de uitgang veranderd die weer in 1 omdat dat a 1 is.



⇨ ***Dynamische hazard*** : uitgang schakelt niet eenmaal meer wisselt eerst effe tussen 0 en 1 (overgangsverschijnselen) bv. i.pv. direct 1 → 0 blijft het signaal eerst even op en neer bewegen)

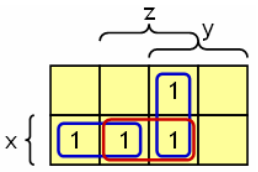
bv. x verandertvan 1 naar 0. hierdoor verandert1 later x’ van 0 naar 1 en nog 1 later verandert x’’ van 1 naar 0. Door de vertraging van de 2de invertor zal de XOR effe 0 worden omdat dan x’ en x’’ effe beide 1 zijn. F wordt door de verandering van x eerst 1 zou normaal 1 moeten blijven omdat a 1 is en x 0. Maar door de glitch van a die effe 0 wordt zal F ook effe 0 zijn.

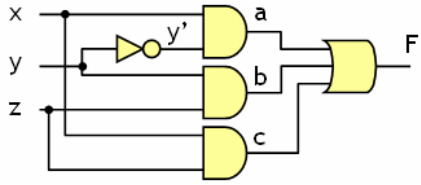


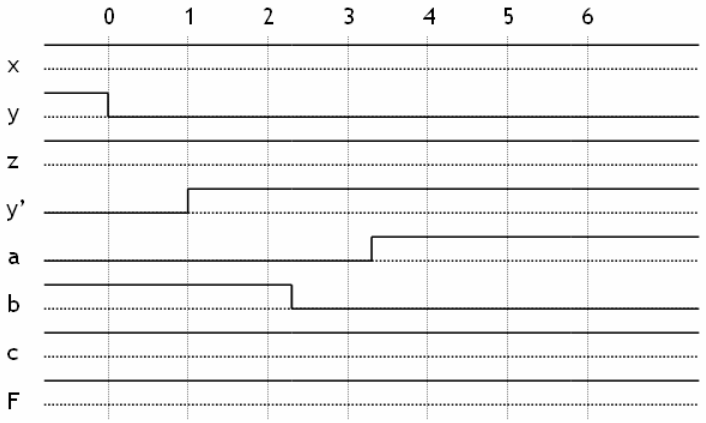
⇨ Het **verschil** tussen de statische hazerd en de dynamische is dat bij de dynamische de uitgang F effectief moet veranderen van 0 naar 1 en daarna een glitch maakt. Dit is op zich geen echt groot probleem daar de schakeling toch al aan het schakelen was. Maar bij statische hazerds zou F normaal constant zou moeten blijven, in plaats van effe te veranderen. Daarom is een statische hazerd een groter probleem dan een dynamische (denk ik), daar de rest van de schakeling door de hazerd hier ook allemaal schakelt waar dat niet de bedoeling is.

We willen nu **liefst een ontwerp zonder hazerds**. Oplossingen :
⇨ We voegen **extra poorten** toe om de **disjuncte gebieden** in de karnaugkaart te bedekken. (hazerds komen namelijk voor waar 2 groepen in een karnaugkaart onafhankelijk schakelen bij dezelfde ingangen) .We zullen nu naast de 2 verschillende paden waarover de veroorzakende term propageer,t een **ander (3de) pad** voorzien waar de veroorzakende term niet in zit. Deze **extra poorten** zijn **redundant** : ze zijn totaal overbodig voor de logische werking, ze werken alleen de hazerd weg.

- voor de **1-hazerds** zullen we een **extra minterm** toevoegen voor disjuncte **1-gebieden**
- voor de **0-hazerds** zullen we een **extra maxterm** toevoegen voor disjuncte **0-gebieden**







bv. in het eerste voorbeeld hadden we disjuncte gebieden op de karnaugh-kaart. We zullen die nu verbinden met een extra gebied dat de y niet bevat. We brengen een extra poort c in die altijd 1 blijft omdat x en z altijd 1 blijven en zo zal ook F altijd 1 blijven, ook al worden a en b effe beide 0.

⇨ Hazards zijn **moeilijk handmatig te detecteren**. We zullen ons ontwerp moeten **simuleren** in de **tijd** en het tijdsgedrag analyseren om hazerds te ontdekken.

⇨ De **kans** op hazards **vergroot** naarmate t_{PLH} en t_{PHL} meer verschillen
(verschil in overgangstijden van laag naar hoog en omgekeerd)

⇨ Hazerds zijn een **probleem** als :

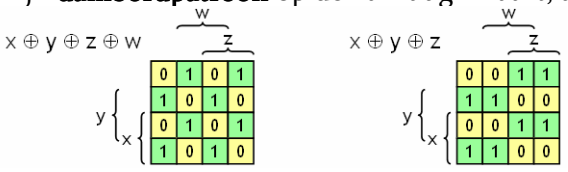
- Als **signaalovergangen** gebruikt worden om te schakelen (bijv. klok) dan kan een hazerd ongewild aanzien worden als een signaalovergang.
- Als **signaalniveaus** gebruikt worden en de hazard komt op een slecht ogenblik (cfr. sequentiële schakelingen). Slechte ogenblikken voor een synchrone schakeling zijn hazerds die samenkomen met de klok. Wanneer de hazerd net valt op de klok zal de foute waarde van het signaal ingelezen worden, ook al staat deze foute waarde maar heel even op de draad. Voor een asynchrone schakeling is de hazerd altijd slecht.

Basisbouwblokken op RTL-niveau (ADD, ALU, MUX, …)

1) Optellen & aftrekken

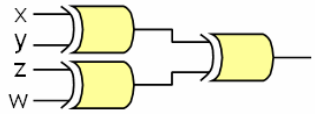
⇨ Interludium : de **XOR-poort**. Deze is **alleen 1** als **slechts één** van zijn ingangen 1 is.

⇨ **herkenbaar** aan zijn **dambordpatroon** op de karnaugh-kaart, afhankelijk van de schikking van de variabelen.



⇨ voor **meerdere ingangen** kunnen we hem **opsplitsen in een boom** van XOR's met minder ingangen

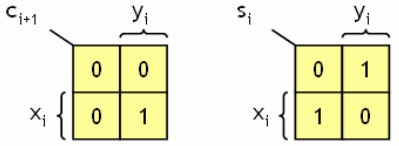
x ⊕ y ⊕ z ⊕ w
= (x ⊕ y) ⊕ (z ⊕ w)

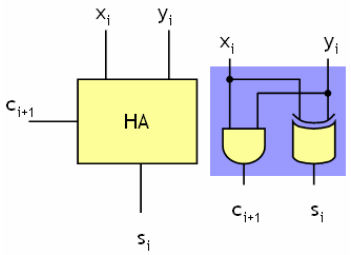


⇨ de XOR is **handig bruikbaar** als **programmeerbare invertor** (we kunnen sturen of het signaal al dan niet geïnverteerd wordt)

⇨ **Halve opteller** (Half adder) : **2 input-bits** die **opgeteld** worden. Gewone werking van de **OR-poort**, buiten het feit dat wanneer **beide ingangen 1** zijn, we de **uitkomst 0** stellen en een **carry-bit 1** stellen. Deze carry-bit is 1 maar stelt dan de waarde 2 voor.

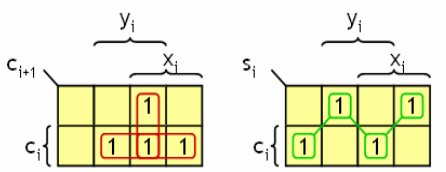
x_i	y_i	c_{i+1}	s_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

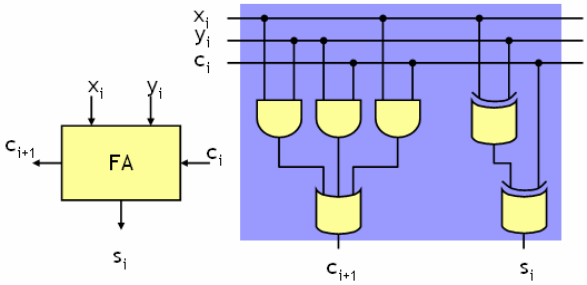


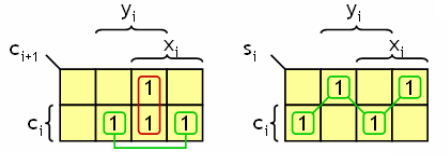


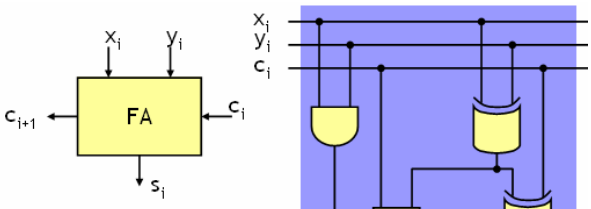
⇨ **Volledige opteller** (Full Adder) **FA** : Houdt ook rekening met een **eventuele carry-bit als ingang** van de vorige optelling (3 ingangen nu dus). De werking is analoog met de halve opteller. 0 wanneer alles 0 is, uitgang 1 wanneer er 1 ingang 1 is, carry als er 2 ingangen 1 zijn en carry en uitgang 1 wanneer alledrie 1. Er zijn verschillende mogelijke implementaties :

x_i	y_i	c_i	c_{i+1}	s_i
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	0	1	1	0
1	1	1	1	1





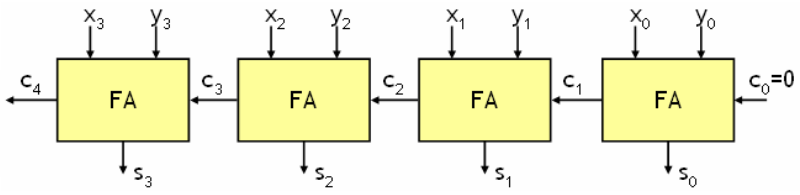




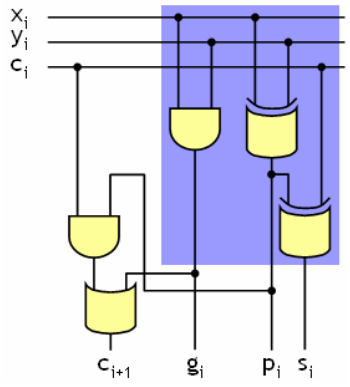
1 poort minder,
grotere vertraging van x_i & y_i naar c_{i+1},
dezelfde vertraging van c_i naar c_{i+1}

⇨ **Ripple-carry opteller** : Opteller van **2 n-bits getallen**. We hebben dus **2 keer n ingangen**, n uitgangen en een carry-uitgang. Realisatie door het **aaneenschakelen van n volledige optellers** waarbij de carry **doorgegeven** wordt.

4-bit ripple-carry opteller
Kritisch pad: x_0 of $y_0 \rightarrow c_n$: 1 XOR + *n* AND + *n* OR
Vertraging: 3,2 + *n* × 2,4 + *n* × 2,4



⇨ **Carry-look-ahead opteller** : de ripple-carry heeft een **redelijk lang kritisch pad** van x_0 naar c_{n+1}. We kunnen dit **versnellen** door **c_{n+1} appart te berekenen** uit c_0, $x_0 \ldots x_n$ en $y_0 \ldots y_n$. Dit noemt met carry-look-ahead.



Opbouw : de rechter XOR's dienen voor de **algemene som s_i** te geven en **c_{j+1}** is de **algemene overdracht**. De p_i is de som van beide ingangen en g_i is de carry, beide zonder de input-carry te kijken. De **p_i** (**carry-propagate** = $x_i y_i' + x_i' y_i$) en de **g_i** (**carry generate** = $x_i y_i$) zijn bedoeld voor het cascaderen van de optellers tot n-bit optellers. De **carry $c_{j+1} = g_i + c_i p_i$** is 1 wanneer er carry is ten gevolge van x_i en y_i of wanneer er één van beide 1 is samen met de carry-in.

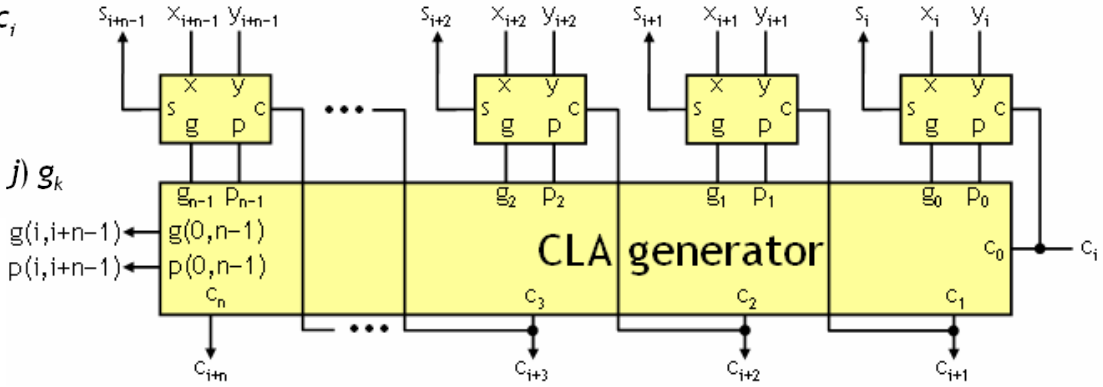
⇨ Wanneer we **n optellers** in **cascade** schakelen om een **n-bit opteller** te krijgen, zullen we nu de **carry's appart berekenen** vanuit de p_i's en de g_i's in een **apparte logische blok : de CLA** generator (Carry-Look-Ahead). Elke carry c_{j+1} wordt daar berekend op basis van factoren g(i,j) en p(i,j) en de vorige carry ci. p(i,j) en g(i,j) zijn functies van de p_i's en g_i's tot j. We kunnen zo een **efficiënte implementatie** van de CLA maken.

CLA generatorfuncties:

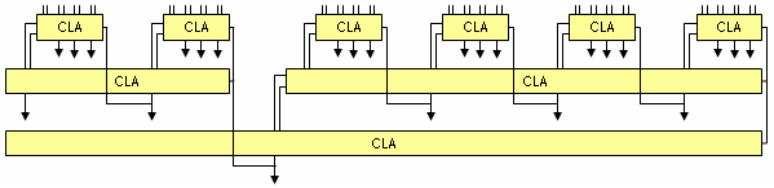
$$c_{j+1} = g(i,j) + p(i,j)\,c_i$$

$$p(i,j) = \prod_{k=i}^{j} p_k$$

$$g(i,j) = g_j + \sum_{k=i}^{j-1} p(k+1,j)\,g_k$$

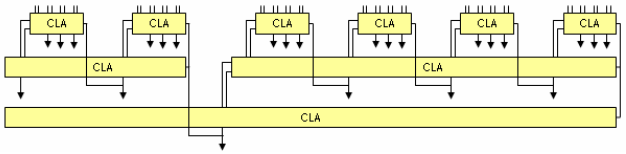


⇨ De CLA moet volgens de formules nu dus voor een **n-bit opteller** een som en product kunnen maken van **n binaire getallen**. Dit impliceert een **AND en een OR poort** met **n ingangen**. Wanneer n groot wordt, **overstijgen we de fan-in**. We moeten onze CLA's dus **beperken in aantal optellers** die eraan hangen; meestal **4 optellers per CLA**. Wanneer we nu optellers met meer dan 4 bits willen maken, moeten we de CLA's combineren in verschillende lagen : cascade. Elke CLB krijgt dus uitgangen g_{j+1} en p_{j+i} om over te dragen op de CLA die een trapje hoger staat en die 4 CLA's uit het trapje onder hem combineert.



⇨ het **kritische pad** wordt voor een 2-lagen systeem : x_i of y_i naar c_{i+n} geeft dus 1 XOR, 1 n-input AND en 1 n-input OR. De vertraging wordt dan 6.4 + 0.8n [=3.2 + 2(1.6 + n0.4)].

⇨ **cascade van k-bit CLA-generators in $\log_k(n)$ niveaus** : bv. k = 4 bit met n=24 (geeft 2.29 niveaus = log 4 24). Vertraging is $3{,}2 + (2\lceil \log_k n \rceil - 1)\times(3{,}2 + 0{,}8k)$). Deze vertraging is logaritmisch stijgend.



⇨ <u>Opteller-Aftrekker</u> (adder-substractor) : We maken nu een systeem waarmee we **2 n-bit getallen** kunnen **optellen en aftrekken**. We hebben dan een **extra inputbit <u>S</u>** waarmee we zeggen of er een **optelling volgt of een aftrekking**.

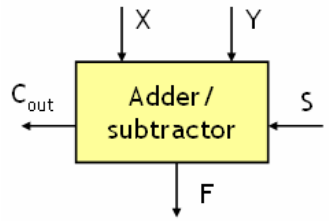
Een aftrekking zullen we doen door het **2-complement** van het getal dat afgetrokken moet worden **op te tellen** bij het andere. Dit 2-complement bekomen we door **elke bit van het 2de getal te inverteren**, en er **1 bij op te tellen**.

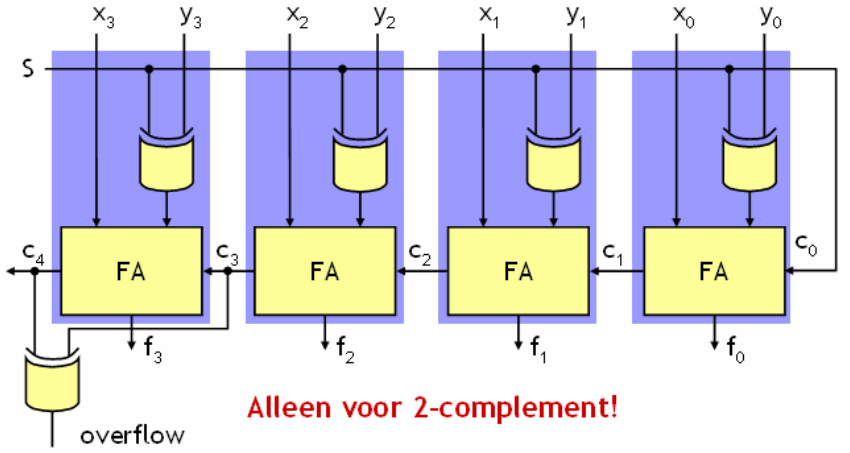
S	Functie	Bewerking
0	X + Y	Optelling
1	X − Y = X + Y* = X + Y' + 1	Aftrekking

⇨ Deze **1 erbij optellen** doen we door de **carry** van de eerst FA (Full Adder) op 1 te zetten. (hierdoor komt er 1 bij de som bij, dat is hetzelfde als 1 bij het af te trekken getal te tellen.

⇨ Om het 2de getal **te inverteren**, zullen we **XOR's gebruiken**. We hebben gezien dat we die als **schakelbare invertoren** hebben gebruiken. We zullen dus elke ingang y_n samen met het S-signaal door een XOR halen.

- Wanneer S 0 is tellen we op en zal y_n gewoon doorgelaten worden.
- Wanneer S 1 is zal yn geïnverteerd worden.





Detectie van overflow : We hebben overflow wanneer de **2 op te tellen getallen** (dus het gewone en het geïnverteerde) **allebei van teken verschillen met het resultaat** (dus beide hetzelfde teken en dat verschilt van het resultaat).

Het **positief of negatief** zijn van een getal zien we aan de **meest beduidende bit** :
0 voor positief en 1 voor negatief.

⇨ Wanneer we **2 positieve getallen optellen**, zal er dus bij **overflow** carry zijn naar de meest beduidende bit c_3, en zal c_4 zowiezo 0 zijn omdat zowel x_3 als y_3 0 zijn voor positieve getallen.

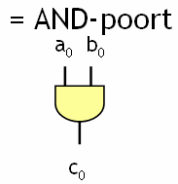
⇨ Wanneer we **2 negatieve getallen** optellen zal er zowiezo carry zijn op c_4, en bij overflow niet op c3 omdat de meer negatieve getallen x_2 en y_2 0 hebben (zo 2 optellen geeft overflow) en de minder negatieve getallen x_2 en y_2 1 hebben (zo 2 optellen geeft geen overflow).

Dit volgt uit de definitie van het 2 complement : bv. 7 is 0111 / 1 is 0001 / -1 is 1111 / -8 is 1000.

Om de **overflow te detecteren** combineren we dus **c_3 en c_4 in een XOR**.

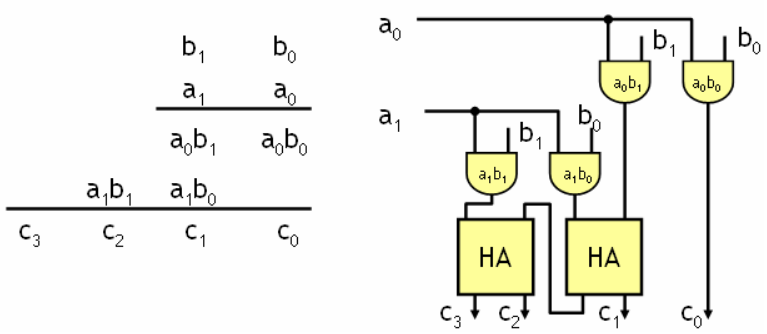
2) Vermenigvuldigen

Parallelle vermenigvuldigers : we vermenigvuldigen **2 n-bits** binaire getallen.

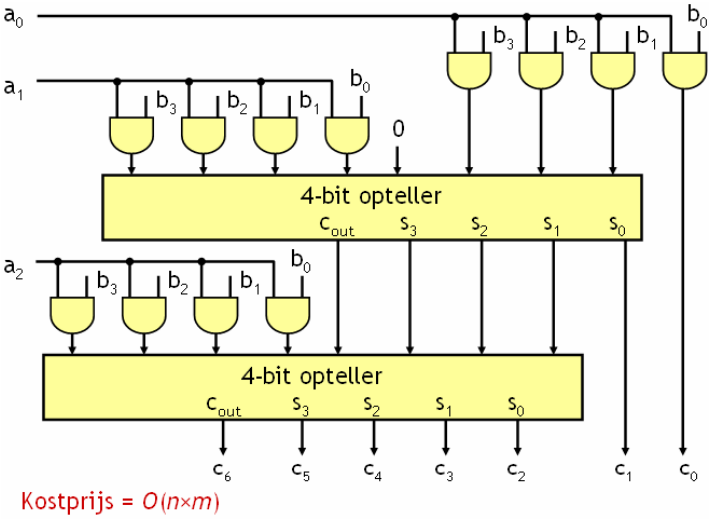


⇨ **1-bit maal 1-bit product** : Dit is simpelweg een AND-poort : wanneer er minstens één van beide 0 is geeft een product 0. Wanneer beide 1 zijn geeft 1

⇨ **2-bit maal 2-bit product** : Dit geeft maximaal een **4-bit getal** dus moeten we 4 bits rekenen als uitgang. Systeem van vermenigvuldigen :
we **herhalen het vermenigvuldigtal** verschillende keren onder elkaar,
telkens **1 plaats opgeschoven** (realiseren met draden),
telkens **vermenigvuldigt met het volgende getal** uit de vermenigvuldiger (met AND- poorten),
En uiteindelijk **tellen we alles op** tot de einduitkomst (met halve optellers).



⇨ 4-**bit maal 3-bit product** : we werken in stapjes en tellen telkens een nieuw, overschoven en vermenigvuldigt vermenigvuldigtal op bij wat we al hadden.



Kostprijs = $O(n \times m)$

⇨ Wanneer we werken met **optellers/aftrekkers** en rekening houden met de **tekenbit**, kunnen we ook **negatieve getallen vermenigvuldigen**. Wanneer we bv. -1 maal -1 in 4-bit-voorstelling vermenigvuldigen krijgen we 1111 maal 1111. dit geeft inderdaad 00000001.

Bijv. $1111_2 \times 1111_2 = 00000001_2$
($1111_2 = -1_{10} = -8 + 4 + 2 + 1$)

1×-1	⇒	11111111
+ 2×-1	⇒	+ 11111110
+ 4×-1	⇒	+ 11111100
+ -8×-1	⇒	+ 00001000
-1×-1	⇒	00000001

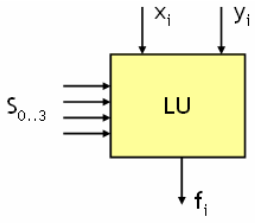
⇨ Overgaan naar **sign-magnitude-voorstelling** geeft meestal **minder hardware** en is dus soms beter.

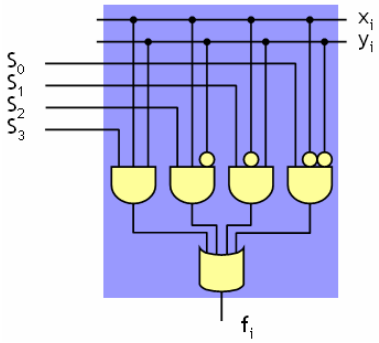
3) (A)LU : (Aritmetische) & logische eenheid

⇨ <u>LU : logische eenheid</u> : component die **alle 16 mogelijke booleaanse functies** van 2 bits kan realiseren. De analoge eenheid voor n bits wordt gemaakt met n eenheden voor elk 1 bit. We **selecteren de functie** die we willen toepassen op de ingangen met een **input van 4 bits $S_3S_2S_1S_0$** , waarmee we de getallen 0 tot 15 kunnen ingeven. Deze komen elk overeen met één van de 16 booleaanse functies : de **codering** van de selectiebits is identiek aan het functienummer in de tabel van de mogelijke Booleaanse functies.

1-mintermen	Functiewaarden voor x,y 00	01	10	11	Uitdrukking
–	0	0	0	0	$F_0 = 0$
m_3	0	0	0	1	$F_1 = xy$
m_2	0	0	1	0	$F_2 = xy'$
$m_2 + m_3$	0	0	1	1	$F_3 = x$
m_1	0	1	0	0	$F_4 = x'y$
$m_1 + m_3$	0	1	0	1	$F_5 = y$
$m_1 + m_2$	0	1	1	0	$F_6 = xy' + x'y$
$m_1 + m_2 + m_3$	0	1	1	1	$F_7 = x + y$
m_0	1	0	0	0	$F_8 = (x + y)'$
$m_0 + m_3$	1	0	0	1	$F_9 = xy + x'y'$
$m_0 + m_2$	1	0	1	0	$F_{10} = y'$
$m_0 + m_2 + m_3$	1	0	1	1	$F_{11} = x + y'$
$m_0 + m_1$	1	1	0	0	$F_{12} = x'$
$m_0 + m_1 + m_3$	1	1	0	1	$F_{13} = x' + y$
$m_0 + m_1 + m_2$	1	1	1	0	$F_{14} = (xy)'$
$m_0 + m_1 + m_2 + m_3$	1	1	1	1	$F_{15} = 1$

We **implementeren** deze logisch eenheid als volgt :





⇨ <u>ALU : Aritmetische logische eenheid</u> : **twee 4-bits ingangen** worden verwerkt tot **één 4-bits uitgang**. Deze uitgang is een bepaalde **functie** van de ingang. Deze **functie kunnen we zelf kiezen**, zoals we in de LU ook de functie van de ingangsvariabelen konden kiezen. Opties :

⇨ we voeren **4 aritmetische bewerkingen** uit op de 4 ingangsbits :

elke a_n en b_n kunnen worden **opgetelt, afgetrokken, 'increment' (+1) en 'decrement' (–1)**

⇨ we voeren **4 logische bewerkingen** op de 4 ingangsbits :

elke a_n en b_n worden gecombineert als **AND, OR, INV of identiteit (gewoon buffer)**.

We **implementeren** deze ALU met :

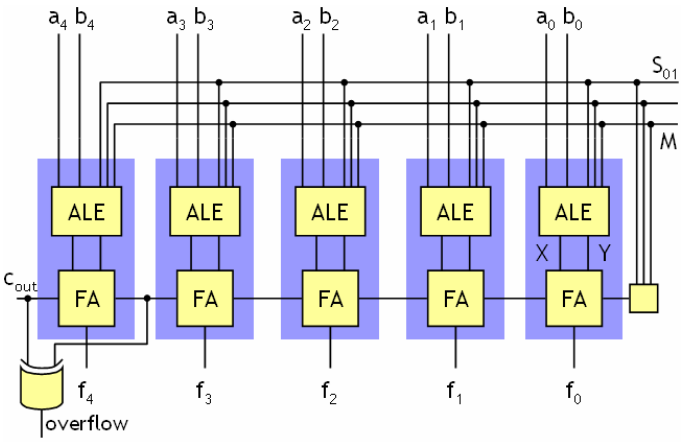
⇨ **Opteller** voor **aritmetische** operaties : de FA (full-adders)

⇨ **Arithmetic-Logic Extender (ALE)** voor de **conditionering van de ingangen** van de opteller (cfr. XOR bij aftrekking) en voor de **logische operaties**. (komt voor de opteller)

We kiezen tussen de **logische of aritmetische** functie via **3 ingangsbits S_0, S_1 en M**. Zij bepalen de carry-in van de eerste FA en komen ook in elke ALE om daar te sturen

⇨ **M** kiest tussen 0 voor **logische** bewerkingen en 1 voor **artimetische**

⇨ **S_0** en **S_1** selecteren **welke bewerking** we doen. (de functie)

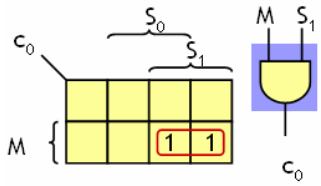


M kiest het type bewerking: 0 = logisch, 1 = aritmetisch
S_0 en S_1 kiezen de bewerking

⇨ **Mogelijke functies** in de ALU : De **logische functies** bepalen we in de **ALE** en worden dan gewoon **opgeteld met 0** in de FA. In de FA's word dan **opgetelt**, en via de **carry in** en de **2de ingang verschillende bewerkingen** op de getal(len) uitgevoerd.

M	S_1	S_0	Functie	F	X	Y	C_0
0	0	0	Complement	A'	A'	0	0
0	0	1	AND	A AND B	A AND B	0	0
0	1	0	Identity	A	A	0	0
0	1	1	OR	A OR B	A OR B	0	0
1	0	0	Decrement	A−1	A	all 1	0
1	0	1	Add	A+B	A	B	0
1	1	0	Subtract	A−B	A	B'	1
1	1	1	Increment	A+1	A	all 0	1

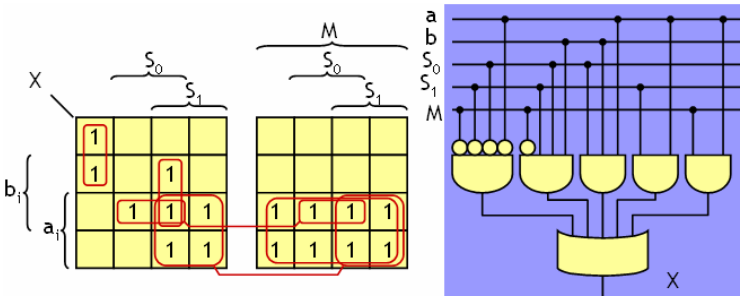
⇨ De **carry-in c0** moet 0 zijn **voor alle logische functies**. Ook voor **decrement** en **optellen** is hij 0. Wordt pas gebruikt bij **increment** wanneer we via de **carry-in 1** kunnen optellen bij het getal, en om hij het **aftrekken** een **2-complement** te maken.

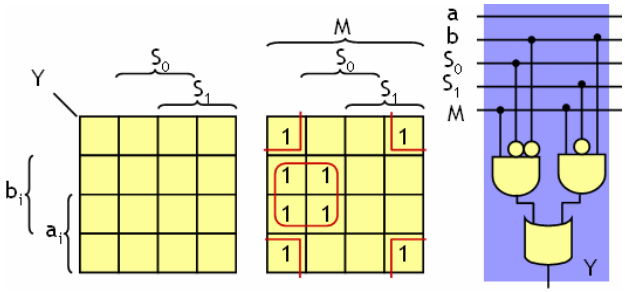


⇨ De **implementatie** van de uitgang X en Y van de **ALE** bepalen we als als volgt :

X-uitgang : - voor **aritmetische** bewerkingen geeft die gewoon het eerste getal telkens door.
- voor **logische** bewerkingen zal hij het resultaat van de bewerking doorgeven. Daar wordt dan gewoon 0 bij optgeteld en zo doorgegeven naar de uitgang.

Y-uitgang : - voor **logische** bewerkingen is die altijd 0 omdat dat 0 bij de andere uitgang wordt opgeteld en die zo als resultaat naar de uitgang gaat.
- voor aritmetische bewerkingen wordt het 2de getal voor de optelling daar geprepareed. Voor optelling gewoon B, aftrekken complement van B, increment 0 en voor decrement allemaal 1tjes (is -1 in het 2-compementair binair)

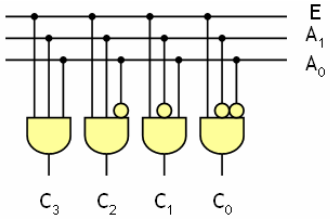


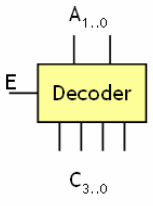


4) (De)multiplexer en Decoders

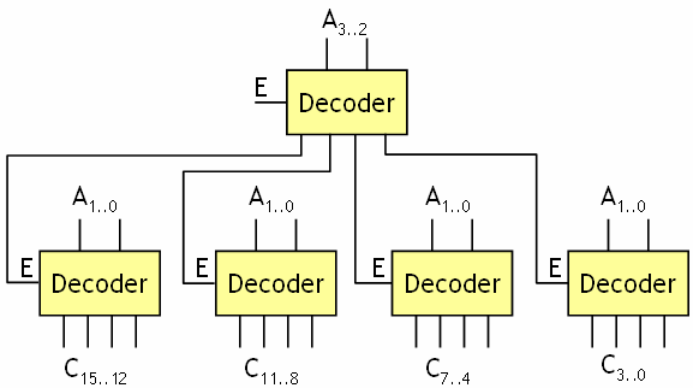
⇨ Een **decorder** of **demultiplexer** (demux) is een schakeling die **bepaald aantal ingangen** omzet naar een **groter aantal uitgangen** volgens een **bepaalde codering** en daarbij dus **redundante info (bits)** toevoegt.

Bv. de 2-naar-4 decoder codeert 2 ingangsbits tot 4 uitgangsbits. Er is ook een enable-ingang waarmee we alle uitgangen 0 kunnen maken.





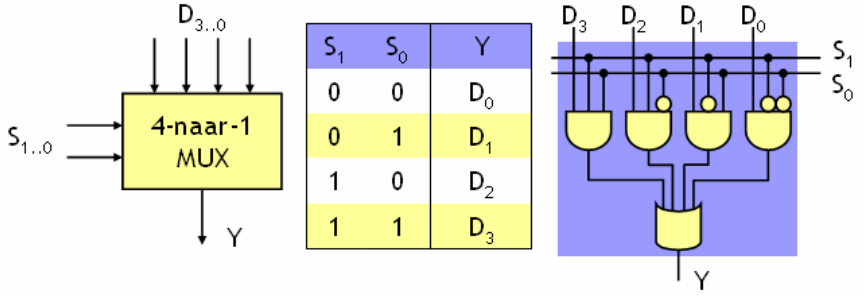
E	A_1	A_0	C_3	C_2	C_1	C_0
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0



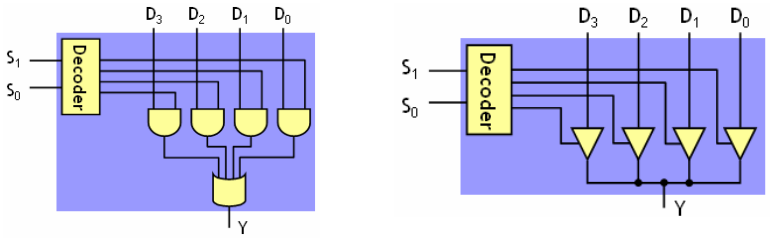
⇨ wanneer we de **decorders in cascade** achtereen zetten, kunnen we grotere decoders maken : een **nm-naar-2^{nm}** decoder maken we door **n niveaus** van gecascadeerde **m-naar-2^{m}** decorders te bouwen. We hebben dan $(2^{nm} - 1)/(2^{m} - 1)$ decoders in totaal nodig.

Bv. 2 niveaus van **2-naar-4 decoders** geeft een **4-naar-16 decoder** : bovenaan komen 2 van de 4 ingangen A_3 en A_2 binnen, die dan gecodeerd naar de 4 enable-ingangen van de 4 decoders op het niveau eronder gaan. In elk van die 4 onderste decoders worden dan de 2 laatste ingangen A_1 en A_0 gecodeerd.

⇨ De <u>selector</u> of <u>multiplexer (MUX)</u> selecteerd één **van de 2^n ingangssignalen als uitgang**. Deze keuze wordt bepaald door de **n sturende inputssignalen** die binair de nummer doorgeven van het te selecteren kanaal. We kunnen een 2-naar-1 mux implementeren op ½ CLB en een 4-naar-1 CLB op 1 CLB.



We kunnen ook een decoder gebruiken om het sturende signaal te ontleden en te verdelen over de 4 ingangen. We kunnen hierbij AND's gebruiken of 3-state-buffers.



Een **bus** is eigenlijk een **soort van gedistribueerde versie van de MUX**. De ingangen zijn over een lange lijn verdeeld, maar het principe is hetzelfde. De sturing wordt gecodeerd en verdeeld over lijnen waarvan er telkens één naar elke aangesloten 3-state-buffer gaat.

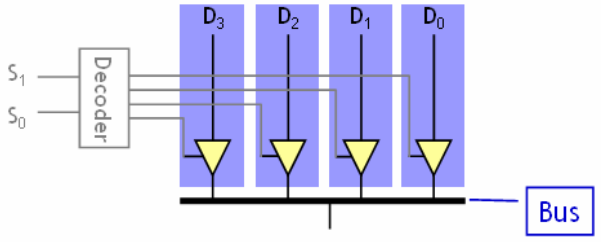
Voordelen van een bus tegenover de MUX :

⇨ bus is **makkelijk uit te bereiden**

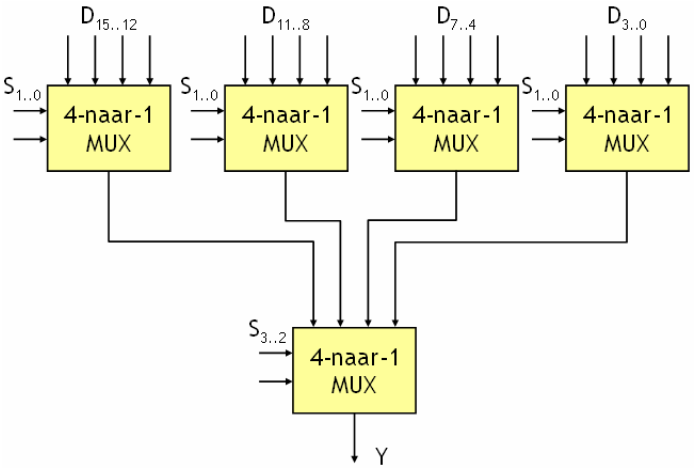
⇨ de **OR-poort** in de MUX heeft een **beperkte fan-in**

⇨ **alle ingangen** moeten naar **1 centrale plaats** bij de MUX waar bij de bus gewoon de bus overal passeert.

⇨ **3-state-buffers** zijn in een FPGA toch **gratis** omdat elke CLB er een heeft die aan een horizontale lange zijn verbonden kan worden



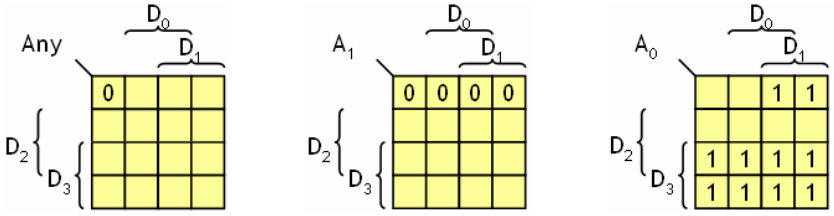
⇨ Ook **multiplexen** kunnen we **<u>cascaderen</u>** om met bv. 5 keer een 4-naar-1-mux een 16-naar-1mux te maken. We hebben dan uiteraard 4 sturingskanalen nodig. Met de eerst 2 sturingssignalen kiezen we uit elke groep van 4 één die doormag, en uit die 4 kiezen we dan met de 2 overige sturingssignalen welke de uiteindelijke uitgang wordt.



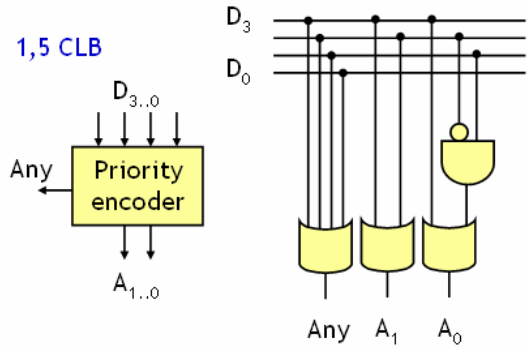
5) Prioriteitsencoder

⇨ De **<u>prioriteitsencoder</u>** is de **inverse van de decoder**. Dat wil zeggen dat een prioriteitsencoder achter een decoder een identiteit geeft. De encoder heeft **n uitgangen** en **2^n ingangen**. De uitgang geeft telkens **binair de nummer van de plaats van de meest beduidende bit** in de ingang. Dit omdat in uitgang van de decoder slechts op één plaats een 1 staat en dit meteen de meest beduidende bit is. De encoder heeft ook een **uitgang Any**, die slechts 0 geeft wanneer alle uitgangen 0 zijn.

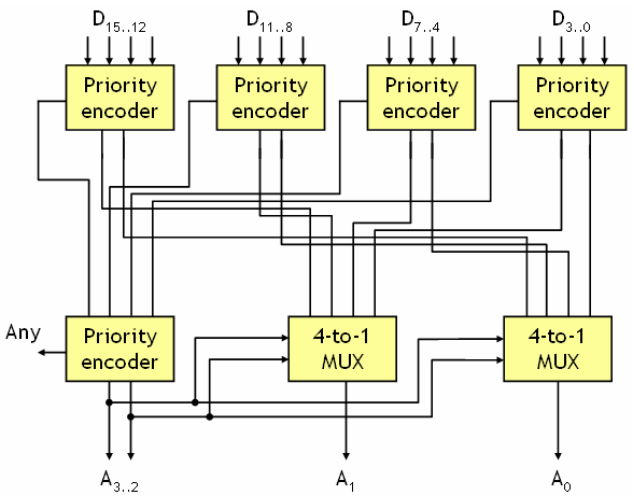
D_3	D_2	D_1	D_0	A_1	A_0	Any
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	1
0	0	1	1	0	1	1
0	1	0	0	1	0	1
0	1	0	1	1	0	1
0	1	1	0	1	0	1
0	1	1	1	1	0	1
1	0	0	0	1	1	1
1	0	0	1	1	1	1
1	0	1	0	1	1	1
1	0	1	1	1	1	1
1	1	0	0	1	1	1
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	1	1	1



⇨ **Implementatie** : Any combineert alles gewoon in een AND, A_1 moet 1 geven wanneer we D_2 en/of D_3 1 hebben en A_0 moet werken wanneer D_1 en niet D_2 of wanneer D_3 1 is.



⇨ **Cascaderen** van prioriteitsencoders om meer complexe encoders te maken : 16-naar-4 encoder gemaakt uit 6 4-naar-2 encoders. Uit elke groep van 4 wordt de nummer van de meest beduidende bit gehaald door 4 encoders. We combineren alle Any's van die 4 encoders in 1 encoder, die daar dan A_3 en A_2 uithaalt. Van de 4 encoders worden telkens alle U_0 en U_1 appart samengebracht in twee multiplexen, met als sturing de uitgangen A_3 en A_2. Met de sturing weten we in welke van de 4 encoders de meest beduidende bit zit, en in de multiplexen selecteren we dan de uitgangen van die encoder als A_0 en A_1.

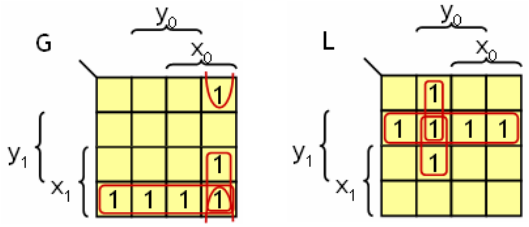


6) Vergelijken (comparator)

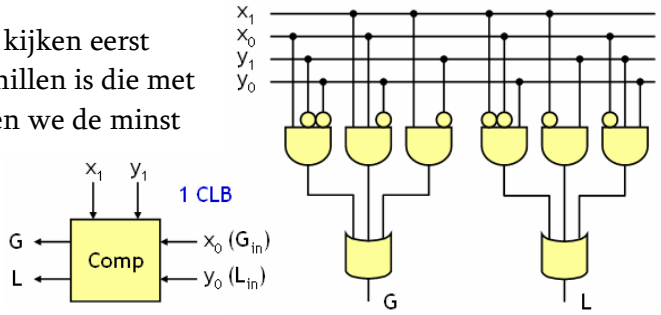
		G_{in}	L_{in}		
x_1	y_1	x_0	y_0	G (X>Y)	L (X<Y)
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	1	0
0	0	1	1	0	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	1	0	1
1	1	1	0	1	0
1	1	1	1	0	0

⇨ De comparator heeft 2 **n-bits ingangen X en Y** en **2 uitgangen G en L**. Hij zal nu de ingangen X en Y **vergelijken** in **numerieke waarden** en op basis daarvan G en L bepalen.
Bepalen van de uitgangen :

- ⇨ **X > Y** dan **G = 1**, dus G = 0 als X ≤ Y
- ⇨ **Y > X** dan **L = 1**, dus L = 0 als Y ≤ X
- ⇨ **X=Y** dan is **G=0** en **L=0**
- ⇨ G en L zijn nooit allebei samen 1

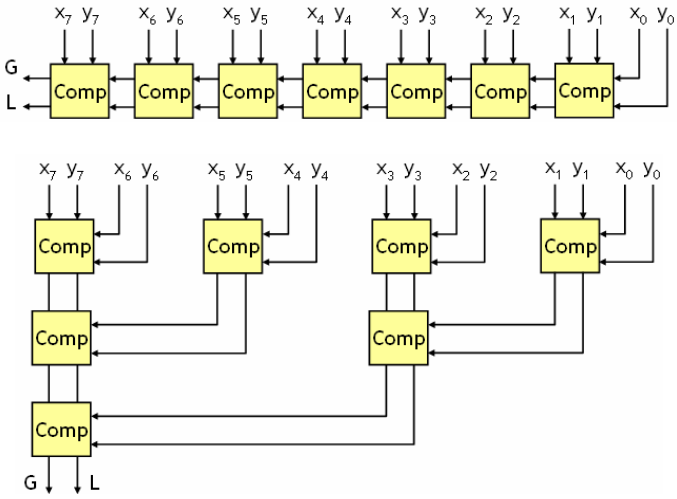


⇨ **Implementatie** van een 2-bits comparator : We kijken eerst naar de **meest beduidende bit** : wanneer die verschillen is die met de 1 de grootste, wanneer die gelijk zijn vergelijken we de minst beduidende bits.



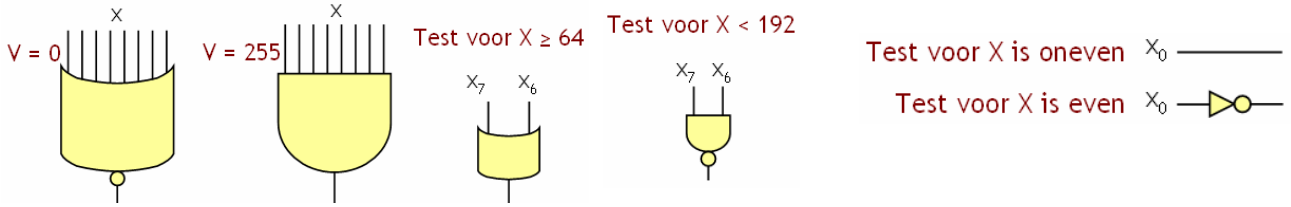
⇨ **Cascade** van **comparatoren** om **grotere getallen** te vergelijken : 2 manieren

⇨ we vergelijken de minst beduidende bits en gebruiken die uitgang G en L telkens als minst beduidende ingang om met het volgende binaire niveau te combineren. G is 1 wanneer van alle niveaus eronder X de grootste is, en anders is L 1. Wanneer beide 0 zijn is wat eronder zit gelijk.



⇨ Het vorige heeft het nadeel nogal lang te zijn en dus minder snel. We kunnen nu in boomvorm werken en telkens per 2 naast elkaarliggende niveaus combineren. We bekomen dan een veel minder lang kritisch pad.

⇨ wanneer we **één n-bits ingang** willen **vergelijken met een vaste waarde** V (dus niet met een 2de variabele ingang), kunnen we dit soms efficienter doen. Stel we vergelijken X (8-bits) met vaste getallen : de schakeling moet 1 geven wanneer x de vaste waarde V heeft.



7) Schuifoperaties

Schuifoperaties : in een reeks van **n bits** (woord, cijfer) verschuiven we **alle bits m posities naar** links '<<' of naar rechts rechts '>>'. Daarbij gaan dus **telkens m bits verloren** en worden er **m nieuwe bits geïntroduceerd**. We kunnen **om 2 redenen verschuiven**, afhankelijk vanwaar de nieuwe bits komen :

⇨ **Logisch schuiven** : de nieuwe bits komen van een **bijkomende ingang**

⇨ **Aritmetisch schuiven** :

- **naar links** schuiven betekend **m nullen rechts erbijvoegen**,
- **naar rechts** schuiven betekend **m bits links erbijvoegen**. Deze toe te voegen bits links hangen af van wat voor getal het is : als we werken met **2-complement** moeten we zorgen dat het teken behouden blijft en moeten we dus **tekenbits** toevoegen (MBS). Wanneer we werken met **sign-magnitude** voegen we **nullen** bij.

Deze verschuivingen zijn (aritmetrisch) **vermenigvuldigingen met 2^m** voor verschuiven naar links en **delen door 2^m** voor verschuiven naar links

Roteren : We roteren een **inputreeks** van **n bits m posities** naar links of rechts : dat betekend dat we links m bits wegnemen en die rechts terug bijvoegen of omgekeerd. Hierbij gaan **geen bits verloren**, de bits die aan de ene kant afvallen worden er aan de andere kant bijgezet.

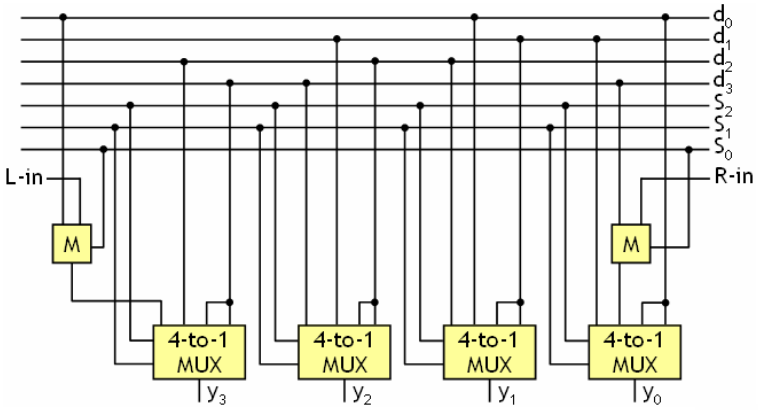
⇨ **Implementatie** : Wanneer we een **4-bit ingangen** hebben, hebben we ook een 4-**bit uitgang**. We hebben ook **3 ingangen om te sturen** :

⇨ S_2 bepaald **of er iets gedaan wordt**.
Wanneer die 0 is ingang gelijk aan uitgang.

⇨ S_1 kiest de **richting** waarin we gaan :
0 voor **links** en 1 voor **rechts**.

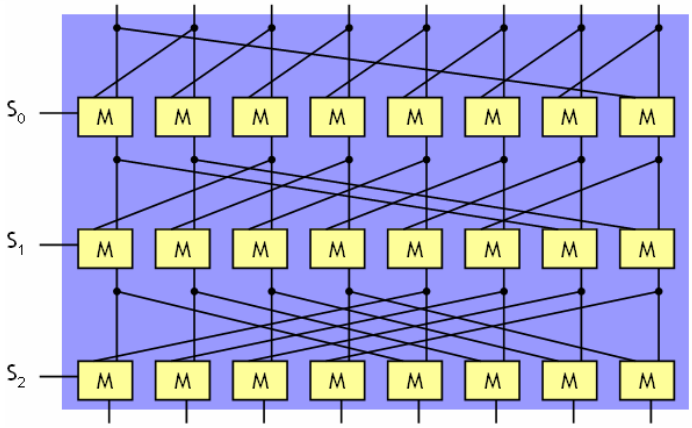
⇨ S_0 bepaald **wat we doen** :
0 voor **verschuiven** en 1 voor **roteren**.

S_2	S_1	S_0	functie
0	X	X	no shift
1	0	0	shift left
1	0	1	rotate left
1	1	0	shift right
1	1	1	rotate right

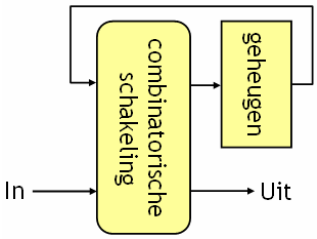


De implementatie gebeurt met **multiplexen** : S_1 en S_2 gaan telkens naar de sturing van de MUXen, en zo wordt bepaald welk van de ingangen geselecteerd wordt als uitgang. In de **blokken M** wordt via S_0 (is 0 wanneer er verschoven wordt) bepaald of we het originele signaal nemen (roteren) of een externe input (verschuiven).

⇨ Een **barrel rotator** is een rotator die een **met ingangen instelbaar aantal keren** naar een **vaste richting** roteert. Bv. de **8-bit barrel left rotator** roteerd de 8 bits van de ingang n keer naar links, waarbij we getal n kunnen instellen met de ingangen S_0, S_1 en S_2, die binair het aantal keer draaien aanduiden.



Hfdst 4) Sequentiële schakelingen



⇨ Bij de **combinatiorische** schakelingen is de **uitgang een directe functie van de ingangen**. Er is daar geen geheugen; in de tijd geeft het systeem altijd gelijke uitgangen voor gelijke ingangen.
⇨ Bij **sequentiële** schakelingen is de uitgang **afhankelijk** van de **ingang** en van de **toestand** waarin het systeem verkeerd. Deze toestand is afhankelijk van de **voorafgaande ingangen** en is opgeslagen in het **geheugen** :

- **statisch** : bv. flip flops : positieve terugkoppeling
- **dynamisch** : dingen waarvan de toestand kan veranderen in de tijd
  bv. capaciteiten die lading opslagen

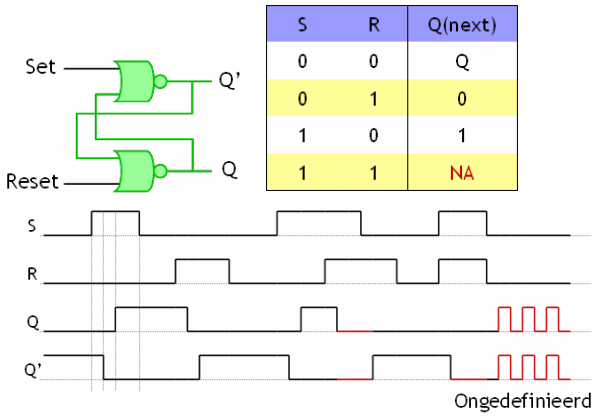
Soorten sequentiële schakelingen :

⇨ **Asynchrone** schakeling: uitgangen & toestand veranderen wanneer een **ingang verandert**
⇨ **Synchrone** schakeling: uitgangen & toestand veranderen alleen wanneer de **klokingang** verandert (op **vaste, geklokte tijdstippen**)
Deze **klok** is onafhankelijk van de schakeling en heeft bepaalde **karakteristieken** :

- **Klokperiode** : tijd tussen 2 opeenvolgende 1 → 0 klokovergangen
  (de tijd van een periode hoog en periode laag samen)
- **Klokfrequentie** : 1 / klokperiode, dus het aantal klokperiodes per seconde
- **Klokbreedte** ('klok width') : lengte van de periode waarop de klok 1 is
- '**Duty cycle**' : verhouding van de klokbreedte en de klokperiode;
  fractie van de klokperiode dat de klok 1 is
- **Stijgende flank** ('rising edge') : 0 → 1 klokovergang :
  kloksignaal is **actief hoog** als de synchrone schakeling schakelt tijdens de stijgende flank
- **Dalende flank** ('falling edge') = 1→0 klokovergang :
  kloksignaal is **actief laag** als de synchrone schakeling schakelt tijdens de dalende flank

De flipflop als bouwblok

1) SR latch



S	R	Q(next)
0	0	Q
0	1	0
1	0	1
1	1	NA

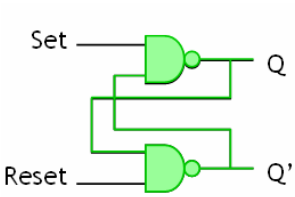
De **SR latch** (set-reset latch) is het **simpelste geheugen-element** : 2 ingangen (**set en reset**), 2 toestanden (set en reset), 2 uitgangen (**Q en Q'**), 2 NOR-poorten die gekruist gekoppelt zijn. Wanneer we de **ingangen veranderen**, zal de **uitgang Q veranderen** afhankelijk van zijn **toestand**.

⇨ Zolang **Set is 0** en **Reset is 0** : de **uitgang Q zal gelijk blijven** (niet veranderen)
⇨ Wanneer **Set 1** en **Reset 0** wordt : de uitgang **Q** komt na 2 poortvertragingen **zowiezo op 1** (of blijft 1 als ie al 1 was). Dit is de Set-toestand. Q' wordt al 0 na 1 klokvertraging.
⇨ Wanneer **Set 0** en **Reset 1** wordt : de uitgang **Q** komt na 2 poortvertragingen **zowiezo op 0** (of blijft 0 als ie al 0 was). Dit is de Reset-toestand. Q' wordt al 1 na 1 klokvertraging.
⇨ Wanneer **tegelijkertijd** Set en Reset **van 1 op 0 vallen** dan is **Q ongedefinieerd** en onvoorspelbaar. Dit komt door de **poortvertragingen** :

- wanneer **beide poortvertragingen gelijk** zijn zullen de 2 poorten telkens tesamen van niveau veranderen en blijven op en neer gaan met als frequentie de poortvertraging.
- wanneer **één van de 2 poorten minder vertraging** heeft zal Q die zijn waarde aannemen.

Het **uitgangssignaal Q** kan nu 5 verschillende waarden aannemen :

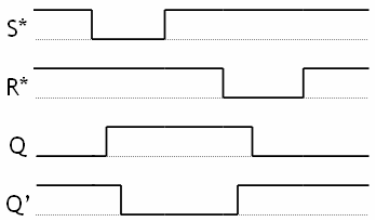
⇨ 0 : het logisch niveau "0"
⇨ 1 : het logisch niveau "1"
⇨ Z : hoog-impedant (zwevend, niet aangesloten)
⇨ X : don't care
⇨ U : ongedefinieerd



S*	R*	Q(next)
1	1	Q
1	0	0
0	1	1
0	0	NA

We kunnen de **SR latch** ook maken met **2 NAND**-poorten.
De werking is dan **invers** en de **ingangssignalen actief laag** :

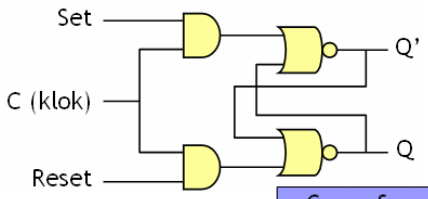
⇨ Wanneer **Set 0** wordt komen we in de **set-toestand** waarbij **Q=1**.
⇨ Wanneer **Reset 0** wordt komen we in de **reset-toestand** waarbij **Q=0**.



2) Geklokte latch

Geklokte SR latch :

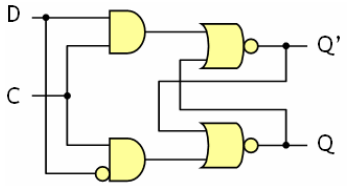
Bij de geklokte SR latch zal de **uitgang Q** pas kunnen **veranderen** wanneer het **kloksignaal 1** is. We sturen de set en reset ingang eerst door een **AND-poort** samen met de klok zodat het 1 worden van set of reset pas effect heeft wanneer de klok 1 is. Wanneer de **klok 0** is blijft de uitgang **Q dus behouden**, ongeacht de ingangen.



C	S	R	Q(next)
0	X	X	Q
1	0	0	Q
1	0	1	0
1	1	0	1
1	1	1	NA

Geklokte D latch :

Het probleem met de SR latch is dat **set en reset ingang nooit samen 1** mogen zijn. Dit wordt verholpen door **slechts 1 ingang D** te nemen die **set door 1** te worden en **reset door 0** te worden. D komt dan rechtstreeks op Set en via een invertor op Reset zodat deze nooit samen 1 zijn. Ook hier betrekken we een **klok C** : de uitgang volgt de ingang wanneer c=1 en de uitgang blijft gelijk wanneer c=0.



klok C	D	Q(next)
0	X	Q
1	0	0
1	1	1

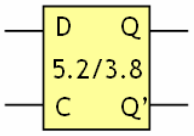
⇨ De **vertragingen** in de **overgangen** :
Wanneer de **klok 1** is heeft een **verandering van D effect op Q en op Q'**:

- vertraging op Q : $t_{HL} = 1 + 2{,}4 + 1{,}4 = 4{,}8$ / $t_{LH} = 2{,}4 + 1{,}4 + 1{,}4 = 5{,}2$
- vertraging op Q' : $t_{HL} = 2{,}4 + 1{,}4 = 3{,}8$

Wanneer de **klok 0** is heeft D geen effect. Wanneer C van 0 naar 1 schakelt zal de uitgang de waarde van de ingang aannemen :

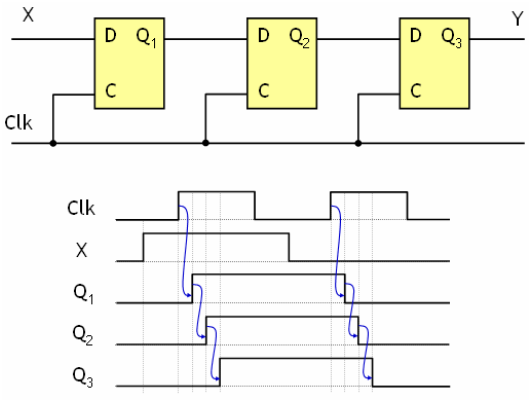
- als D = 1 : $t_{LH} = 2{,}4 + 1{,}4 + 1{,}4 = 5{,}2$
- als D = 0 : $t_{HL} = 2{,}4 + 1{,}4 = 3{,}8$

We zien dat t_{LH} altijd 5.2 is en dat t_{HL} minstens 3.8 is, we noteren dit in het symbool van de D latch :



3) Gevoeligheid

Level-sensitive latch



Voorgaande **geklokte latches** zijn **gevoelig** voor het **klokniveau** : ze onthouden hun waarde wanneer C=0 en zijn transparant wanneer C=1. **Transparant** wil zeggen dat **elke verandering op de ingang effect heeft op de uitgang** (na een kleine vertraging). Dit transparant zijn kan **problemen** geven wanneer we **meerdere latches achter elkaar** willen plaatsen (in bv. schuifregisters). Een verandering in het ingangssignaal kan dan **doorrimpelen door alle latches** tijdens **1 klokperiode**. Dit willen we **vermijden**: we hebben liever dat op elke klokperiode de ingang doorgegeven wordt naar 1 volgende latch en niet naar alle daaropvolgende.

We zouden kunnen **proberen** om de **klokperiodes aan te passen aan de omschakeltijden** van de latch, maar die **verschillen** voor hoog-laag / laag-hoog, waardoor dit **geen goede oplossing** is. We kunnen ook niet de setup-tijd (c=1) en de houd-tijd (c=0) aanpassen aan de schakeltijd, omdat de schakeltijd variabel is.

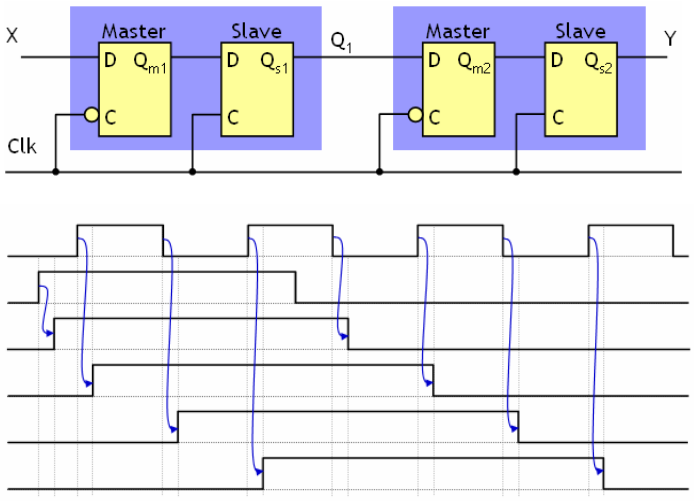
⇨ De oplossing is de **flipflop** die **flankgevoelig** werkt: we zullen de flipflop **enkel omschakelen op een vaste klokflank** waardoor de flipflop **niet meer transparant** is. De uitgang kan slecht veranderen op 1 tijdstip (de flank van de klok) en niet meer heel de periode dat de klok 1 is.

Master-Slave Flip-flop

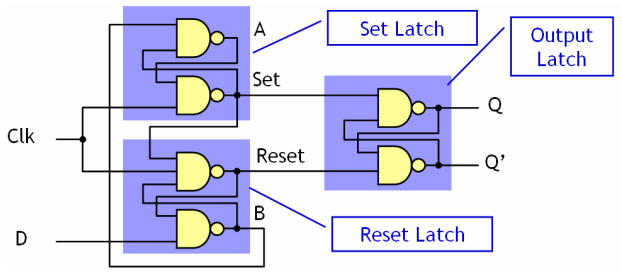
Een **Master-Slave** flipflop is een combinatie van een **master-latch** en een **slave-latch** : de ingang D komt op de **master-ingang Dm**. De master-uitgang Qm wordt dan gebruikt om de slave-ingang Ds aan te sturen. De **uitgang Q** is de uitgang van de **slave Qs**.

⇨ Het **niet-transparant** maken gebeurt door éénzelfde **kloksignaal rechtstreeks aan de slave** te hangen, en **geïnverteerd aan de master**. Hierdoor zijn de **uitgangen** van master en slave slechts **variabel** wanneer de andere **onveranderlijk** is. Hierdoor kan een **verandering op de ingang** nooit direct effect hebben op de **uitgang**, tenzij op een klokflank.

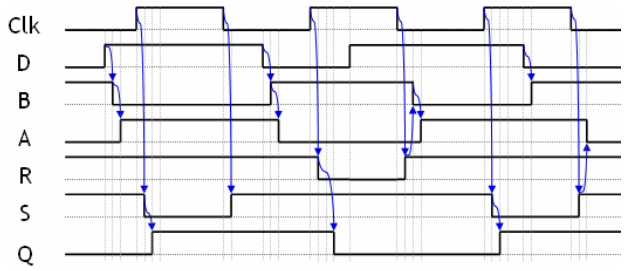
- Bij elke klokovergang **0->1** wordt de **master invariabel** en zal de slave de uitgang van de master doorgeven (die niet verandert).
- Bij elke klokovergang **1->0** wordt de **slave invariabel** en behoud zijn uitgang terwijl de **master een nieuwe waarde** kan aannemen die hij dan zal doorgeven wanneer we terug 0-> gaan. De **uitgang verandert** nu **telkens bij 0->1** en neemt de waarde aan van de **master op de moment van de klokflank.**



Edge-triggered Flip-flop

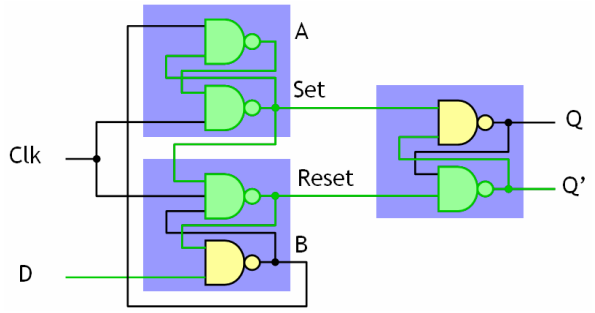


De **edge-triggered flip-flop** bestaat uit **3 SR latches** : de **set**-latch, de **reset**-latch en de **output**-latch. De **ingang D** komt binnen op de **Reset-latch**, en op de andere ingang staan de **klok** en de Set-uitgang van de set-latch. De uitgang Q wordt als reset-uitgang geleverd naar de output, Q' wordt de 2de input van de set-latch. De uitgang Q van de set-latch wordt samen met de reset uitgang in de **output-latch** gestoken die dan de **uitgangen Q** en Q' levert. De SR latches in deze flipflop zijn gebouwd met **NAND-poorten**, waardoor ze invers werken !!!

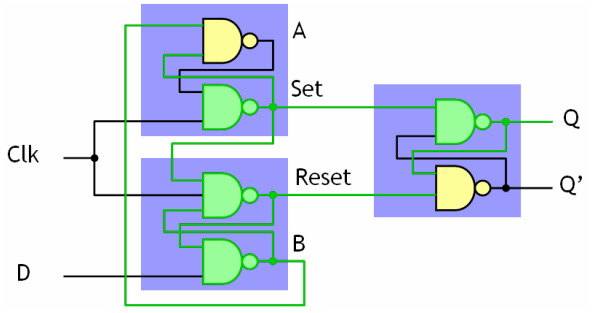


⇨ Wanneer de **klok 0** is wordt een **verandering in D** geregistreerd door de set en reset latches door **A en B te veranderen**. A en B hebben telkens een **tegengestelde waarde** en veranderen **tesamen met D**. Klok 0 betekent ook dat beide **set- en reset-signaalgang** voor de output-latch **1** zijn en dat de **uitgang Q behouden** blijft. (klok zit mee op NAND-poort)

D=1 dan is B=0 en A=1



D=0 dan is B=1 en A=0

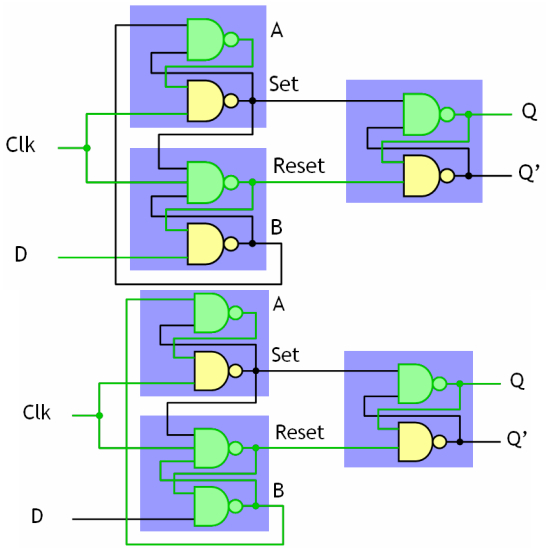


⇨ Wanneer de **klok overgaat 0->1** kan de **combinatie van A en B** op dat moment het **set- of reset-signaal 0** maken, waardoor de **output geset of gereset wordt**. A en B zijn op die moment elkaars tegengestelde. Set en rest kunnen **nooit samen 0** worden door de doorverbinding van set naar de NAND van reset : is set 0 kan is reset zowiezo 1 omdat hij een 0 van set op de NAND krijgt.

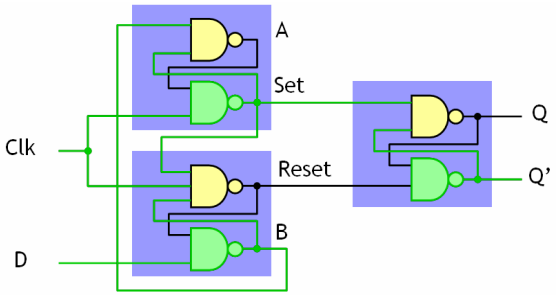
⇨ Wanneer de **klok 1** is heeft een verandering in D geen effect op de uitgang, geen effect op set en reset, maar wel op A en B. Deze veranderen onafhankelijk van set en reset.

A=1 en B=0 op stijgende klokflank zorgt voor **Set**

D heeft geen effect op set/reset



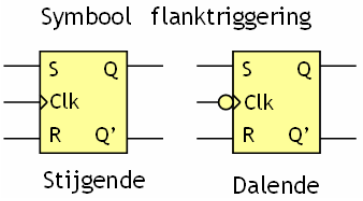
A=0 en B=1 op stijgende klokflank zorgt voor **Reset**



⇨ Wanneer de **klok overgaat 1->0** worden set- en reset-signaal weer beide 1, waardoor de uitgang niet kan veranderen.

Algemeen kunnen we stellen dat de **uitgang enkel gewijzigd** wordt op een **stijgende klokflank** en die neemt dan de **waarde aan van D** op de moment van de stijgende klokflank. Deze werking is volledig analoog aan die van de Master-Slave flip-flop.

4) Types flip-flops



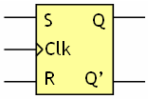
⇨ Elke flip-flop is **flankgetriggerd**. Dit wil zeg dat zijn **uitgang enkel veranderd** op een stijgende of een dalende **klokflank**. Voor flip-flops die schakelen bij stijgende klokflanken wordt de klokinput aangeduid met een kleine driehoek, bij dalende klokflanken komt er een invertor (bolletje) voor de klokingang.

⇨ De **naam** van de flipflop is afgeleid van de namen van zijn ingangen

⇨ Elke flip-flop heeft een **karakteristieke tabel**. Deze is een **korte versie** van de **waarheidstabel** waarbij we **Qnext** bepalen vanuit de ingangen. Deze tabel wordt gebruikt om de ***flip-flop zelf te ontwerpen***. (geeft de werking van de ff zelf aan) We kunnen er de karateristieke vergelijking uit halen, die Qnext geeft als functie van de ingangen en de huidige Q.

⇨ Elke flip-flop heeft **exitatietabel**. Deze geeft de **ingangen** weer die **nodig** zijn om van een gegeven **huidige uitgang Q naar een volgende gewenste Qnext** te gaan bij de volgende klokflank. Deze tabel wordt gebruikt om sequentiële ***ontwerpen te maken met de flip-flop***. (geeft de werking van de ff in een circuit aan).

SR flip-flop

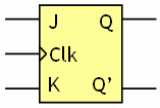


S	R	Q(next)
0	0	Q
0	1	0
1	0	1
1	1	NA

Q	Q(next)	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

Heeft 2 ingangen S set en R reset. Voor S=1 wordt Qnext 1, voor R=1 wordt Qnext 0, voor beide 0 blijft Q behouden. Beide 1 komt nooit voor.

JK flip-flop



Karakteristieke tabel

J	K	Q(next)
0	0	Q
0	1	0
1	0	1
1	1	Q'

Excitatietabel

Q	Q(next)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Heeft 2 ingangen J en K. Zijn werking is **analoog** aan de **SR** ff. Het **verschil** is dat wanneer **beide ingangen 1** zijn, de **uitgang geïnverteerd** wordt. We krijgen dan Q' als Qnext. We kunnen deze ff **maken met een SR** ff door een eenvoudige **terugkoppeling van de uitgangen** Q en Q' en 2 AND-poorten: **S=JQ'** en **R=KQ**.

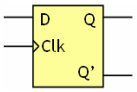
⇨ Wanneer beide **J en K 0** is de werking evident.

⇨ Wanneer **J 1** wordt, zal Q' ofwel 0 zijn en dan is Q reeds 1, ofwel zal Q' 1 zijn en dan wordt de ff geset zodat Q i wordt. Analoog voor reset.

⇨ Wanneer beide **J en K 1** zijn hangt het van de huidige Q af wat er gebeurt, en zal de ff zowiezo van toestand veranderen. (ofwel geset worden als Q'=1 ofwel gereset als Q=1)

JK ff hebben **meer don't cares** in hun exitatietabel en leveren dus **goedkopere** (simpelere) aansturingen op.

D flip-flop



Karakteristieke tabel

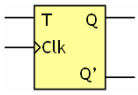
D	Q(next)
0	0
1	1

Excitatietabel

Q	Q(next)	D
0	0	0
0	1	1
1	0	0
1	1	1

De werking van de D flip-flop is **eenvoudig** : Wanneer de **ingang D** veranderd zal de **uitgang Q** mee veranderen.
Deze ff wacht gewoon telkens op de klokflank om de ondertussen eventueel gewijzigde waarde van D door te geven.

T flip-flop



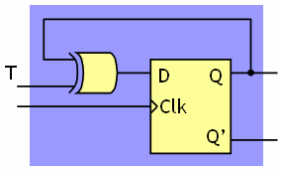
Karakteristieke tabel

T	Q(next)
0	Q
1	Q'

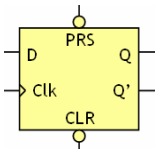
Excitatietabel

Q	Q(next)	T
0	0	0
0	1	1
1	0	1
1	1	0

De **T (toggle) flip-flop** behoudt zijn uitgang wanneer T=0 en inverteerd zijn uitgang wanneer T=1. Bij **elke klokflank** wordt gekeken naar de **waarde van T**, en wordt **Q voor T=0** en **Q' voor T=1** als Qnext genomen.
We kunnen deze T ff maken met een XOR en een D ff.

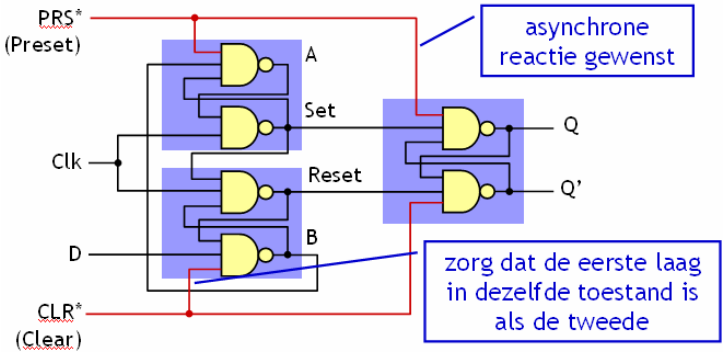


Asynchrone set & reset



Asynchrone set en reset zijn
- een set-ingang (**preset**) en
- een reset-ingang (**clear**) die

⇨ **niet aan de klok** gelinkt zijn (direct de uitgang setten of resetten) en die
⇨ **voorrang krijgen** op alle andere inputsignalen (de anderen worden genegeerd wanneer er een asynchrone set of reset gebeurt).



⇨ Dit wordt gebruikt om de flipflops te **initialiseren** voor gebruik (wanneer de stroom aangezet wordt is de toestand van de ff onbepaald anders) of om de vorige sequentiële ingangen teniet te doen (**herbeginnen**).

⇨ De asynchrone ingangen worden naar de ff **gestuurt als 1** wannneer we willen setten of resetten. Ze worden aan de ingang van de ff echter **geïnverteerd** : het zijn **actief lage signalen** in de ff zelf.

⇨ We **maken** deze door de **asynchrone set PRS** (preset) te koppelen aan :
- aan de **set-ingang** van de output-latch zodat we uitgang setten wanneer PRS 0 wordt
- aan de **NAND** van de **A-uitgang** zodat wanneer PRS 0 wordt we de andere ingangen negeren (beide set en reset worden dan 1)

en door de **asynchrone reset CLR** (clear) analoog te koppelen aan reset en B.

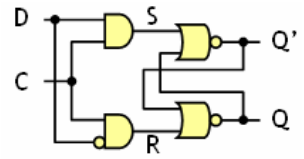
⇨ **Buiten** de flip-flop zijn de **asynchrone signalen actief laag**. Dit doen we omdat deze signalen meestal door **onbekende signalen** of door een **groot aantal signalen** worden **gestuurd**. Het resulterende signaal is dan **actief** wanneer **minstens 1 aansturend signaal actief is**, en dat is makkelijker te realiseren met actief lage signalen in een wired OR.

We zouden dit kunnen **implementeren** met actief hoge signalen in een **grote OR-poort**, maar dat is een slecht idee omdat die **niet flexibel** is : een poort heeft een **beperkt aantal ingangen** (en meestal weten we aanvankelijk niet eens hoeveel ingangen we nodig hebben), en we zullen **veel te lange verbindingen** nodig hebben.

⇨ Oplossing is een ***wired-OR***, en die vereist actief lage signalen.

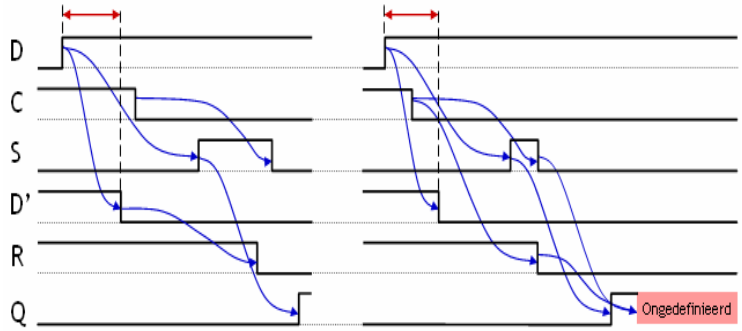
5) Tijdsgedrag

Set-up / houdtijd



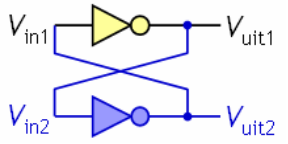
⇨ **Set-up-tijd** : tijd **voor** de **actieve klokflank** waarin de **ingangen niet mogen veranderen**. Dit is de **vertraging** die zit tussen het ingangssignaal en de plaats waar dat samengevoegd wordt met de klok.

Bv. bij de **geklokte D-latch** is dit de vertraging van de invertor, omdat D eerst door een invertor moet en dan bij de klok gevoegd wordt. Stel dat D veranderd vlak voor de klokflank, dan schakelt de latch eerst voor de vorige waarde van D, en na de schakeltijd van de invertor veranderd D' opeens, waardoor de kans bestaat dat we ongedefiniëerde toestanden te krijgen.

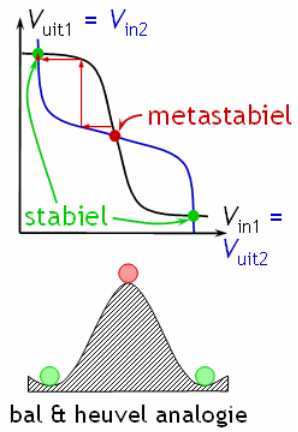


⇨ **Houdtijd** : tijd **na** de **actieve klokflank** waarin de **ingangen niet mogen veranderen**.

Metastabiliteit



Een **bi-stabiel element** is een element met **2 stabiele toestanden**, waarbij de kans bestaat dat er een 3de metastabiele toestand tussen ligt. Een bi-stabiel element in een schakeling krijgen we wanneer we de **uitgang van 2 invertoren** (of poorten met invertor aan de uitgang, bv. NAND's) **gekruist aan elkaars ingang hangen**. (bv. 2 invertors kruiskoppelen) De **uitgang** van de **ene poort hangt dan af** van de uitgang van de **andere** poort en omgekeerd. Voor de 2 invertors zijn er nu 2 stabiele toestanden : de ene uitgang is 1 en de andere 0 of omgekeerd. De metastabiele toestand bestaat erin dat ze beide gelijk zijn en dus ergens tussen 0 en 1 liggen. Op die moment zij ze **ongedefinieerd** (metastabiliteit moet dus vermeden worden). Dit is **geen stabiele toestand** : wanneer één van beide een klein beetje van dit metastabiel punt afwijkt, zullen ze evolueren naar een stabiel punt.



⇨ We kunnen een element **in de metastabiele toestand** brengen door **marginale triggering** (schakelen waar het niet mag) :
- wanneer we de **minimum pulsbreedte schenden**
- wanneer we de ingang veranderen in de **set-up of in de houddtijd**.

⇨ De **kans** dat de **uitgangen in metastabiele toestand zijn** : $p_{\text{nog in meta}}(t) = \exp(-t/\tau)$ met τ een constante die afhangt van :
- van de **hoeveelheid ruis** (meer verstoring -> vlugger van metastabiel evenwicht afwijken)
- van de **steilheid** van de **curve** (makkelijker of moeilijker om uit metastabiel evenwicht te komen)

Ontwerp van synchrone sequentiële schakelingen

1) Finite state machines

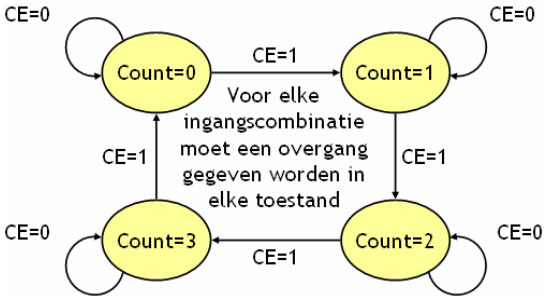
Een **FSM** (Finite State Machine) is een **sequentiële machine** die een **beperkt vast aantal toestanden** heeft. Het systeem bevindt zich **op elk ogenblik** in de tijd in **één van die toestanden** en kan van daaruit **overgaan** naar **verschillende andere toestanden** wanneer de **input-variabelen** veranderen. De input-signalen bepalen **naar welke toestand** de FSM overgaat. De **toestand** waarin het systeem zich bevindt wordt **opgeslagen in flipflops**, en met een bepaalde schakeling kunnen met de input-signalen een nieuwe toestand in de flipflops brengen.

⇨ We zullen nu het **ontwerpen** van een FSM bespreken aan de hand van een voorbeeld :

Ontwerp een modulo 4 teller ('Count uitgang'), die met 1 verhoogt wanneer de ingang CE ('count enable') 1 wordt. Na 3 komt 0 (modulo 4 tellen). We maken deze schakeling synchroon: de teller telt bij elke klokflank 1 bij zolang CE 1 is.

1) We vertalen de **specificatie** (in tekst gegeven) **naar een toestandsdiagram**. We bepalen alle mogelijke **toestanden** en de **verbanden** ertussen. Deze verbanden vinden we door voor **elke toestand alle mogelijkheden** voor de **input-variabelen** af te gaan en te kijken **naar welke andere toestand** we bij die input overgaan. Voor synchrone schakelingen wachten we telkens op de klok om over te gaan naar de volgende toestand.

⇨ We kunnen het toestandsdiagram ook voorstellen als een **tabel**. We zetten dat op **elke rij een toestand**. Voor elke mogelijke **ingangscombinatie** hebben we een **kolom** zodat voor elke toestand en voor elke ingangscombinatie een volgende toestand is bepaald.



Huidige toestand	Volgende toestand	
	CE = 0	CE = 1
S0	S0	S1
S1	S1	S2
S2	S2	S3
S3	S3	S0

2) Minimaliseer het **aantal toestanden**.

In het voorbeeld hebben we reeds een minimum aantal toestanden

3) Codeer de **toestanden** : voor **2^n toestanden** hebben we **n flipflops** nodig. **Elke toestand** krijgt dan een **unieke code** van **n bits** waarmee deze toestand **opgeslagen** wordt in de flip-flops. We geven best toestanden die veel voorkomen codes die niet al te veel bits verschillen zodat de schakeling eenvoudig blijft. (zodat we bij overgang tussen de toestanden niet teveel flipflops moeten veranderen)

Voorbeeld : we zullen hier de 4 verschillende toestanden coderen met elk hun eigen nummer binair voorgesteld. Dit is handig om de uitgang te bepalen en daarmee verschillen de codes bij het overgaan van de ene toestand naar de andere telkens maar 1 bit, zodat we een eenvoudige schakeling krijgen (het zou dom zijn om count 1 te coderen als 11 wanneer we count 0 gecodeert hebben als 00 want dan moeten we bij overgaan telkens de 2 flipflops veranderen)

Toestand	Q_1	Q_0
Count=0	0	0
Count=1	0	1
Count=2	1	0
Count=3	1	1

4) Kies het **type flip-flop** al naar gelang de schakeling.

Voorbeeld : we nemen hier de D flip-flop

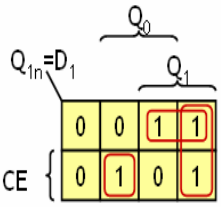
5) We moeten de **schakeling realiseren** met **flipflops** en **combinatorische schakelingen**. We zullen de n bits van de **volgende toestand** (n uitgangen) moeten **bepalen vanuit** de **n huidige bits** en **de i ingangsvariabelen** (n + i ingangen). We hebben dus **n carnaugh-kaarten** die elk n+i ingangen combineren tot een uitgang. Deze uitgangen zijn dan de Qnext van de flipflops.

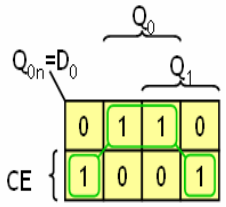
Voorbeeld : we hebben hier 2 D-flipflops die aangestuurd moeten worden vanuit de vorige Q1 en Q2 en de ingang CE (2 carnaugh-kaarten met elk 3 ingangsvariabelen)

Huidige toestand Q_1Q_0	Volgende toestand $Q_{1n}Q_{0n}$ CE = 0	CE = 1
00	00	01
01	01	10
10	10	11
11	11	00

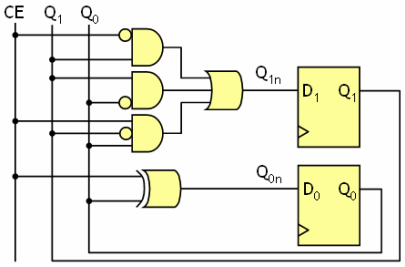
Excitatietabel D-FF

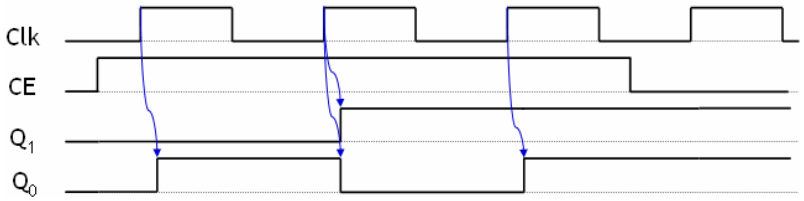
Q	Q(next)	D
0	0	0
0	1	1
1	0	0
1	1	1





6) Implementatie en **simulatie** in het **tijdsdomein** :



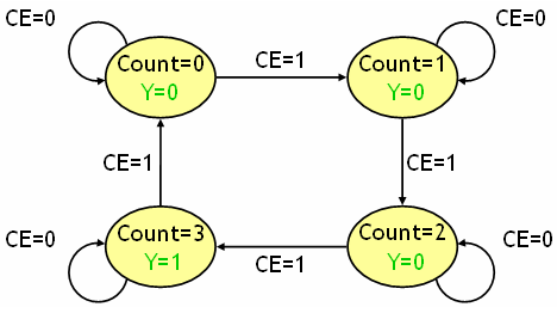


Er bestaan nu **2 soorten FSM** : **Moore**-type waarbij de uitgangen gebaseerd zijn op de toestanden, en **Mealy**-type waar de uitgangen gebaseerd zijn op de huidige toestand **en op de ingangen**.

Moore-type

Een **Moore FSM** is een systeem waarbij de **uitgangen** van de FSM **enkel functie zijn van de huidige toestand**, en niet van de ingangen. Elke toestand heeft zijn **specifieke waarden voor de uitgangen**, die bij genoteerd wordt in de circel van de toestand in het toestandsdiagram.

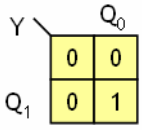
Voorbeeld : stel we willen met de vorige teller een uitgang Y aansturen: We willen dat Y 1 wordt wanneer de teller 3 is, en voor de rest 0 blijft.



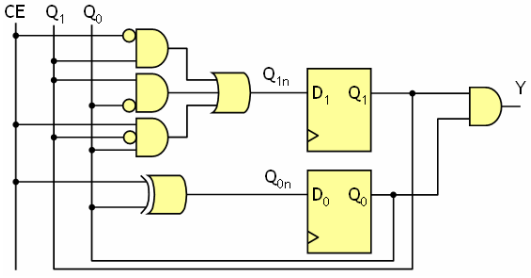
Ontwerp van het voorbeeld : we veronderstellen dezelfde codering van de toestanden. We kunnen dan de tabel van het toestandsdiagram aanvullen met een kolom voor de waarde van Y, die enkel afhangt van de huidige toestand.

Huidige toestand Q_1Q_0	Volgende toestand $Q_{1n}Q_{0n}$ CE = 0	CE = 1	Uitgang Y
00	00	01	0
01	01	10	0
10	10	11	0
11	11	00	1

⇨ We maken ook een extra carnaugh-kaart om de uitgang Y aan te sturen. Daar Y enkel afhankelijk is van de toestand, moet deze kaart enkel de Q's van de ff's als ingangen bevatten, en niet de ingangen van het systeem.



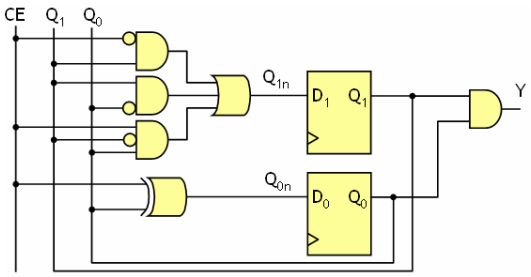
⇨ De uiteindelijke implementatie wordt dan :

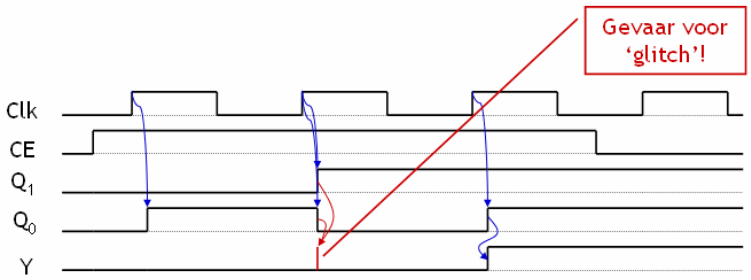


Simulatie in de tijd : We zien dat bij de overgang van Count1 naar Count2 de mogelijkheid voor een **glitch** in de uitgang y bestaat. Q_1 en Q_2 veranderen daar beide tesamen na de klok en met de vertraging van de schakeling. Het kan zijn dat Y

⇨ wanneer de uitgang (met de glitch) **verbonden is aan een kloksignaal** kan dit zorgen voor een **ongewenste actieve klokflank**. Dit is echter moeilijk te ontdekken omdat de glitch kan verdwijnen bij het aanleggen van de meetprobe (hogere capaciteit ⇒ grotere vertraging).

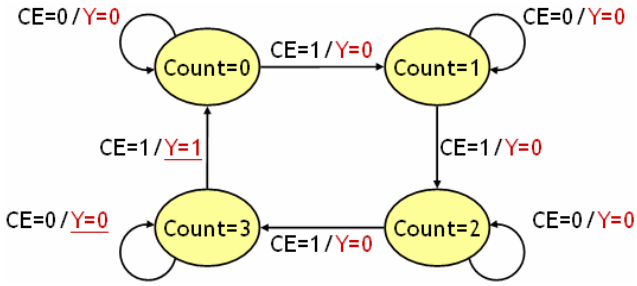
⇨ wanneer de uitgang niet verbonden is met de klok (in synchrone schakelingen) is deze glitch enkel schadelijk als binnen set-up/houdtijd. Hij verbruikt ook wel nodeloos vermogen.





Mealy-type

De **Mealy FSM** is een systeem waarbij de **uitgangen functie** zijn van de **huidige toestand** (zoals Moore) maar **ook van de ingangen**. De uitgang is dus **gespecifiëerd** voor **alle mogelijke toestanden** en **alle mogelijke combinaties van ingangen**. De waarde van de uitgang wordt nu **genoteerd** bij de **overgang** tussen de toestanden, omdat daar de huidige toestand en de ingangen gedefinieerd zijn.



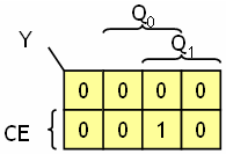
Voorbeeld : stel dat we in de vorige teller een uitgang Y invoeren die 1 wordt wanneer we in toestand Count3 zijn en als CE 1 is. (Y is 1 wanneer we terug naar Count0 overgaan).

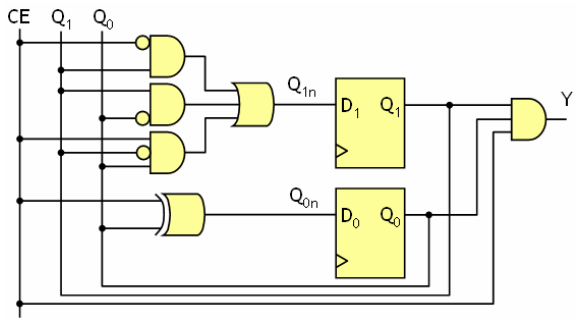
Ontwerp : We veronderstellen weer dezelfde codering van de toestanden, en specifiëren in de tabel van het toestandsdiagram bij elke volgende toestand de uitgang (met een "/" erachter).

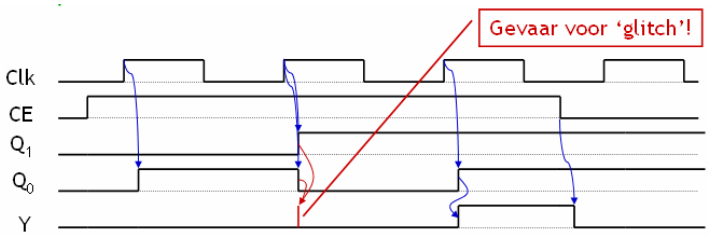
⇨ Aangezien Y nu afhangt van de toestand en de ingang, zullen we voor Y een karnaugh-kaart moeten voorzien met als ingangen alle Q's van de ff en de ingangen.

⇨ De uiteindelijke implementatie en tijdssimulatie met weer kans op een glitch :

Huidige toestand Q_1Q_0	Volgende toestand/uitgangen $Q_{1n}Q_{0n}/Y$ CE=0	CE=1
00	00/0	01/0
01	01/0	10/0
10	10/0	11/0
11	11/0	00/1

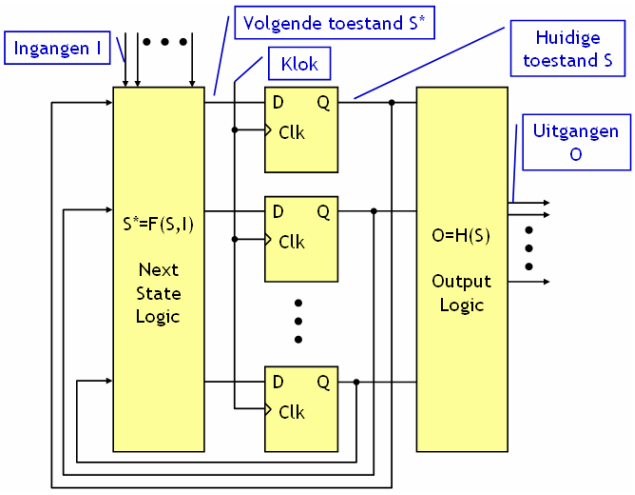




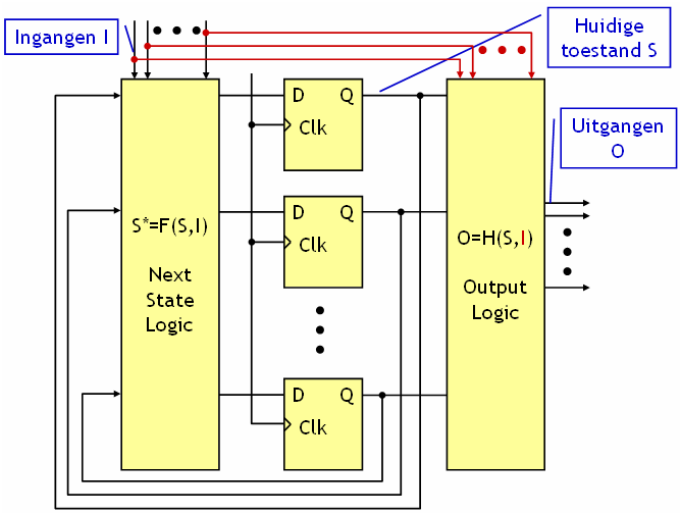


Algemene vorm FSM : Een FSM bestaat uit een **aantal flipflops**, aantal afhankelijk van hoeveel toestanden er zijn, die **bijhouden in welke toestand** het systeem is. Deze worden aangestuurt door de **next-state-logic** die ervoor komt en die vanuit de huidige toestand en de ingangen de volgende toestand bij de volgende actieve kolkflank bepaald. De **output-logic** bepaald dan de uitgang uitgaande van de toestand (Moore) of van de toestanden en de ingangen (Mealy)

Toestandgebaseerd model : Moore



Inputgebaseerd model : Mealy

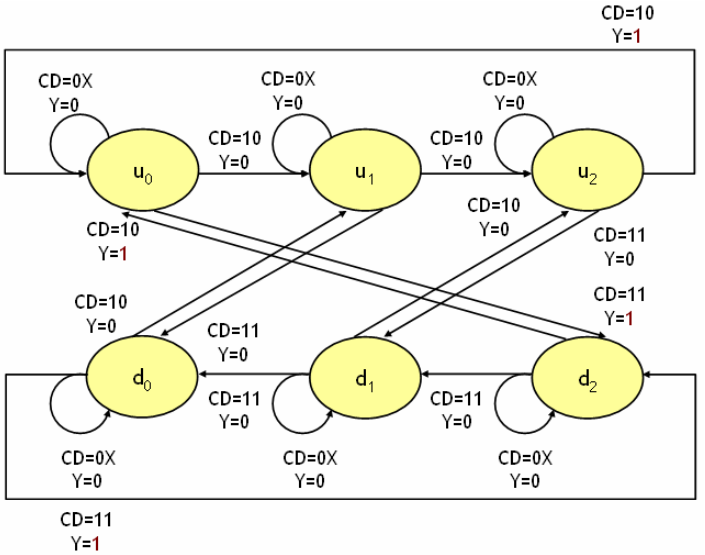


2) Het ontwerp in detail

Toestandsdiagramma : constructie FSM

Het **ontwerp** van een **FSM** begint bij het **maken van het toestanddiagram** vanuit de beschrijving van het probleem. We moeten **alle toestanden** definiëren, de **overgangen** ertussen en de **uitgangen** ervoor.

Voorbeeld : modulo 3 opteller en aftrekker. We zullen onze vorige teller uitbereiden met een mogelijkheid om af te trekken. We hebben nu een ingang C (count) die bepaald of er al dan niet opgeteld of afgetrokken wordt bij de volgende klokflank, en een ingang D (direction) die bepaald of we optellen (D=0) of aftrekken (D=1). We maken ook een uitgang Y die we 1 maken voor optellen bij Count2 en voor aftrekken bij Count0.



1) Kijk eerst of het systeem **Moore of Mealy** is, dus of de uitgang toestands- of inputsgebaseerd is.
Vb: het voorbeeld is inputgebaseerd omdat de uitgang Y afhangt van de toestand (tellerwaarde) en de ingangen "C" & "D"

2) Teken het toestandsdiagram :

⇨ definieer eerst de **initiële toestand** (het systeem start)

⇨ vanuit de initiële toestand zoeken we **voor alle mogelijke ingangscombinaties** de **nieuwe toestanden**. En dan analoog voor elke nieuwe toestand tekenen we voor elke combinatie van ingangen de overgang naar de volgende toestand tot we alle toestanden gevonden hebben (tot voor elke toestand elke ingangscombinatie naar een andere gekende toestand leidt)

⇨ dan zullen we **alle uitgangswaarden toekennen** aan elke toestand als het systeem toestandgebaseerd is, of aan elke overgang voor inputgebaseerd. Wanneer we verschillende uitgangen hebben zullen we meestal eerst een volgorde bepalen voor de uitgangen en ze dan gewoon achter elkaar schrijven.

Bv. wanneer we uitgang x=1 en y=0 willen schrijven doen we xy=10

Minimaliseren van het aantal toestanden (en ff)

Wanneer we voor een complex systeem **alle mogelijke toestanden** zoeken, zou het kunnen dat we zonder het te beseffen **equivalente toestanden hebben gedefinieerd**. Dit zijn toestanden die eigenlijk **volledig gelijk** zijn aan elkaar, waardoor ze eigenlijk **op één na allemaal overbodig** zijn. Wanneer we nu ons aantal toestand en flip-flops willen **minimaliseren**, zullen we **alle groepen van equivalente toestanden vervangen door één van hen.**

⇨ **Twee FSM** zijn **equivalent** als
beide dezelfde **uitgangssequentie** produceren voor **eenzelfde ingangssequenties**

⇨ **Twee toestanden** zijn **equivalent** als
- beide toestanden **dezelfde uitgangen** produceren voor **dezelfde ingangen** (of voor Moore dezelfde uitgang hebben)
- beide toestanden voor **dezelfde ingangen** naar **dezelfde equivalent volgende toestanden** overgaan (dus voor elke mogelijke ingangscombinatie gaan beide toestanden over naar dezelfde equivalente nieuwe toestand)

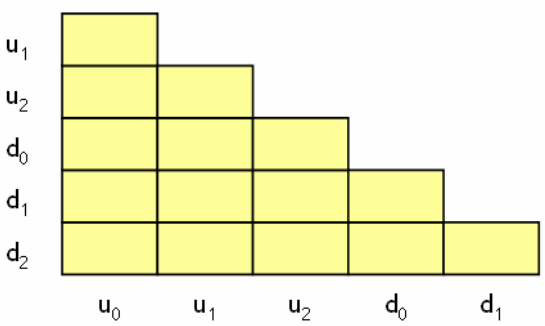
Eén van de 2 is dan **overbodig** en valt weg.

Zoeken van equivalente toestanden :

1) Stel een **toestandstabel** op waarbij de volgende toestanden/uitgangen gegeven zijn in functie van de huidige toestand (per rij) en van de ingangen (per kolom).
Je kan dit makkelijk doen vanuit het toestandsdiagram.

Huidige toestand	Volgende toestand / Uitgangen		
	CD = 0X	CD = 10	CD = 11
u_0	$u_0/0$	$u_1/0$	$d_2/1$
u_1	$u_1/0$	$u_2/0$	$d_0/0$
u_2	$u_2/0$	$u_0/1$	$d_1/0$
d_0	$d_0/0$	$u_1/0$	$d_2/1$
d_1	$d_1/0$	$u_2/0$	$d_0/0$
d_2	$d_2/0$	$u_0/1$	$d_1/0$

2) We maken een **implicatietabel**. Deze bevat 1 **vierkantje** per **combinatie van 2 toestanden**. We kunnen dit doen door een 3hoekige tabel te maken en de vakjes als volgt te benoemen : verticaal de eerste toestand weglaten en horizontaal de laatste toestand. We zullen nu voor elk vakje kijken of zijn 2 toestanden (één horizontaal en één verticaal) equivalent zijn.

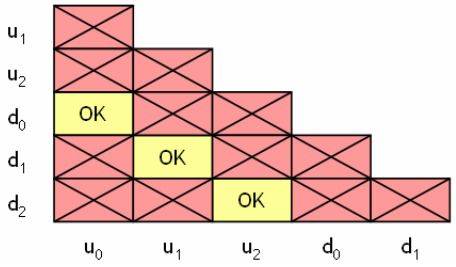


Systeem :

Huidige toestand	Volgende toestand / Uitgangen CD = 0X	CD = 10	CD = 11
u_0	$u_0/0$	$u_1/0$	$d_2/1$
u_1	$u_1/0$	$u_2/0$	$d_0/0$
u_2	$u_2/0$	$u_0/1$	$d_1/0$
d_0	$d_0/0$	$u_1/0$	$d_2/1$
d_1	$d_1/0$	$u_2/0$	$d_0/0$
d_2	$d_2/0$	$u_0/1$	$d_1/0$

⇨ We **schrappen** (groot kruis) alle vakjes waarvan de toestanden **verschillende uitgangen** hebben, omdat deze per definitie niet gelijk kunnen zijn.

⇨ Bij de overige vakjes kijken voor beide toestanden van het vakje **naar de volgende toestanden** waarnaar wordt overgegaan voor **elke mogelijke ingang**. We noteren dit per mogelijke ingang in het vakje, met een streepje tussen. De toestanden van het vakje zijn nu **equivalent** als **alle koppels in het vakje ook equivalent** zijn (omdat ze dan beide overgaan naar dezelfde toestanden voor dezelfde ingangen). We noteren met andere woorden in de vakjes de **toestanden die equivalant moeten zijn opdat de toestanden van het vakje dat ook zouden zijn**. We schrijven OK wanneer we 2 toestanden vinden die equivalent zijn zonder dat er nog andere toestanden daarvoor equivalent moeten zijn.

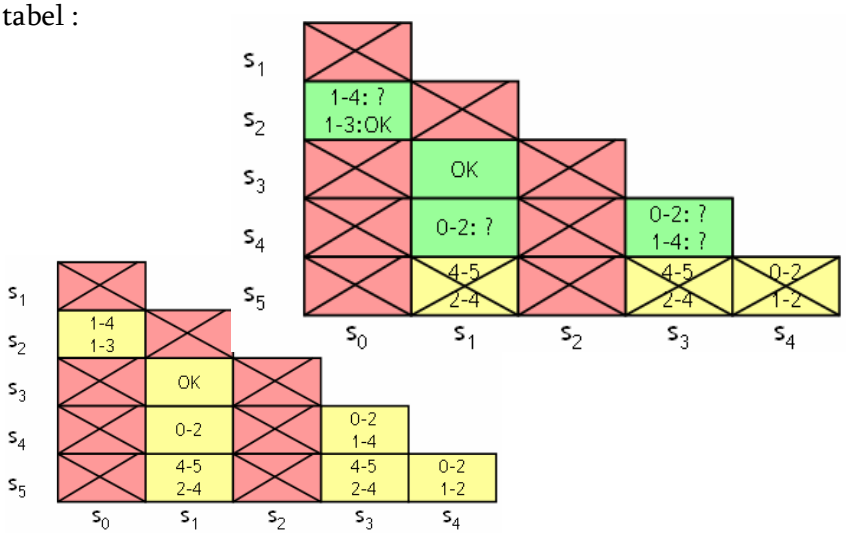


Bv. Wanneer voor ingangen DC=01 de ene toestand overgaat naar 1 en de andere naar 5, en beide toestanden hebben dezelfde uitgang, dan noteren we in het vakja van de 2 toestanden 1-5

We zullen nu de **overige vakje** proberen te **schrappen** door te kijken naar de **koppels** die erin staan. Elk koppel komt ook overeen met een vakje : wanneer het vakje van een koppel al geschrapt is, mogen we het vakje waar het koppel instaat ook schrappen.

We **illustreren** dit even met een fictieve tabel :

Huidige toestand	Volgende toestand / Uitgangen AB=00	AB=01	AB=10
s_0	$s_4/1$	$s_2/0$	$s_1/1$
s_1	$s_2/0$	$s_5/1$	$s_4/1$
s_2	$s_1/1$	$s_0/0$	$s_3/1$
s_3	$s_2/0$	$s_5/1$	$s_4/1$
s_4	$s_0/0$	$s_5/1$	$s_1/1$
s_5	$s_2/0$	$s_4/1$	$s_2/1$

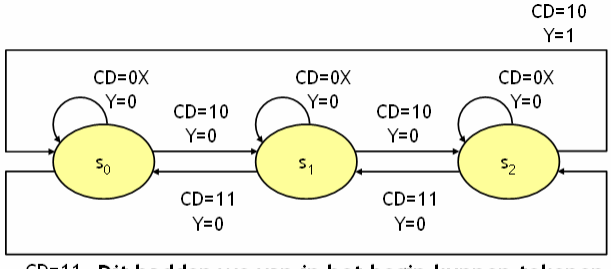


⇨ De vakjes die we nu **nog niet geschrapt** hebben, daarvan zijn de **toestanden equivalent**. We **selecteren** uit elk koppel equivalente toestanden er **één** en **schrappen de andere**. We kunnen nu **nieuwe namen** uitdelen en een **nieuwe tabel** opstellen.

Minimum aantal toestanden: 3
$\{u_0,d_0\} = s_0$
$\{u_1,d_1\} = s_1$
$\{u_2,d_2\} = s_2$

Huidige toestand	Volgende toestand / Uitgangen CD = 0X	CD = 10	CD = 11
s_0	$s_0/0$	$s_1/0$	$s_2/1$
s_1	$s_1/0$	$s_2/0$	$s_0/0$
s_2	$s_2/0$	$s_0/1$	$s_1/0$

Bv. in ons voorbeeld zijn er 3 vakjes over. Deze betekeken equivalantie tussen d_0 en u_0, tussen d_1 en u_1 en tussen d_2 en u_2. We hadden 6 toestanden, we hebben er 3 keer 2 equivalante, dus hebben we 3 toestanden over. We noemen deze s_0, s_1 en s_2.



Dit hadden we van in het begin kunnen tekenen, maar beter 1 toestand te veel dan 1 te weinig !

Codering van de toestanden

We zullen nu **elke toestand een code moeten geven** om in de schakeling mee te werken. De toestand waarin het systeem zich bevindt is **opgeslagen met deze code in de ff's** van het systeem. Wanneer we **n toestanden** hebben hebben we minstens **log2(n) flipflops** en dus bits nodig om deze toestanden te coderen. We hebben **n! mogelijkheden** om de codes toe te kennen aan de verschillende toestanden. Elk van deze mogelijke coderingen zal leiden tot een andere combinatorische schakeling voor de overgangslogica en outputlogica, elk met hun eigen complexiteit en kostprijs. We zullen dus proberen een **codering te kiezen** die zorgt voor een **minimale schakeling** en dus **minimale kostprijs**.
Er zijn **3 manieren** van coderen : **Straightforward** (voor de hand liggend), **Minimum-bit-change** en **One-hot** (één-actief). Er is echter altijd nog verdere verfijning mogelijk (bv. Prioritized-adjacency strategy (cfr. boek))

⇨ **Straightforward** : We kiezen hier de **meest voor de handliggende code**. Deze is makkelijk toe te kennen en makkelijk om mee te werken. We zullen dit kunnen gebruiken wanneer de **toestanden op elkaar volgen of getallen voorstellen**, wanneer het **toestandsnummer een fysieke betekenis** heeft. (bv. een teller) We kunnen dan de codering als **binaire nummering** laten overeenkomen met de echte volgorde of nummering van de toestand.

⇨ **nadeel** is dat er **meestal meerdere ff's tegelijk** moeten veranderen bij overgang tussen toestanden en dat kan problemen met zich meebrengen:

- deze zullen **zelden exact tegelijk** veranderen -> gevaar voor glitches
- elke bitverandering **verbruikt vermogen** -> meer vermogenverbruik
- elke extra bitverandering vereist **extra overgangslogica** -> grotere en dus duurdere schakeling

⇨ **Minimum-bit-change-codering** : we maken het **aantal bitveranderingen** voor alle **toestandsovergangen samen** zo **klein mogelijk**. Zorg bijvoorbeeld dat bij elke overgang de 2 toestanden slechts 1 bit verschillen, zodat er maar 1 ff moet schakelen bij elke overgang. Dit zijn meestal de schakelingen met het minst vermogenverbruik en de laagste kostprijs. (CMOS)
bv. de gray-code teller voor modulo 4 telt 00->01->11->10->00.

⇨ **One-hot-codering** : We voorzien **één flip-flop per toestand**, dus voor n toestanden zijn dat n ff's en n bits. We coderen nu elke toestand met **alle bits 0**, **behalve telkens 1 bit** die 1 is (one-hot). De codes **verschillen nu in de plaats van de 1** : de eerste toestand heeft een 1 op de eerste plaats, de 2de toestand op de 2de plaats, enz.

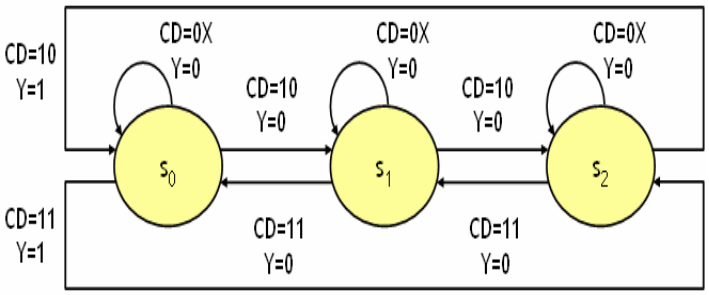
⇨ **Nadelen** : n flip-flops nodig in plaats van log2(n), waardoor **flip-flop-deel van de chip duurder** is en enkel gebruikt wordt voor systemen met een klein aantal toestanden (bv. controller)

⇨ **Voordelen** : het ontwerp van de **combinatorische overgangslogica is zeer eenvoudig** en dus zeer snel en **goedkoop**. De **FPGA** heeft bv. in een halve CLB een kleine combinatorische schakeling en een ff dus one-hot is ideaal voor FPGA (behalve voor tellers omdat die teveel toestanden hebben)

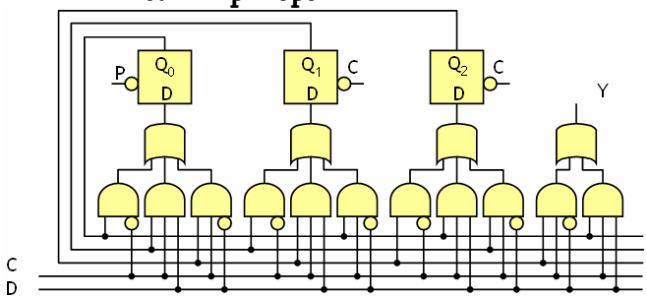
Bij de **impementatie** van One-hot met flip-flops moeten we een **keuze** maken in het soort ff :

⇨ **D-ff**: Elke overgang die in een toestand toekomt vereist een AND-poort

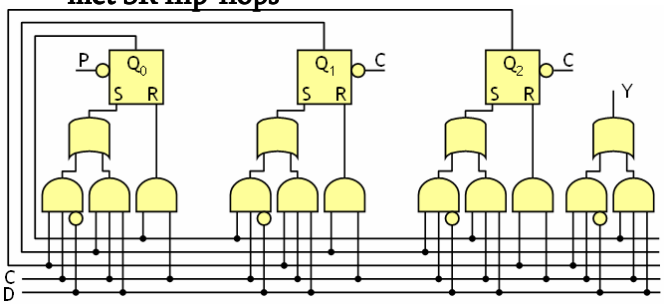
⇨ **SR-ff**: Elke overgang die van een andere toestand toekomt vereist een AND-poort op de S-ingang. Elke overgang die naar een andere toestand vertrekt vereist een AND-poort op de R-ingang



met D flip-flops



met SR flip-flops



Keuze van het type flip-flop

De **4 types flipflop** hebben elk hun **eigen voordelen en nadelen**. Een belangrijke parameter is het aantal **don't cares** : hoe **meer don't cares** hoe **kleiner de schakeling** en dus hoe **goedkoper** en **sneller** de schakeling. Hoe meer don't cares hoe **duurder** uiteraard.

⇨ Om de **goedkoopste** schakeling te vinden moeten alle varianten uitgeprobeerd worden

⇨ Als een **korte ontwerptijd** belangrijk is, zijn D-flip-flops de beste keuze

⇨ FPGA's bevatten enkel D-flip-flops

Overzicht :

	JK	SR	D	T
Kost FF	duurst	goedkoop	goedkoop	goedkoop
Eenvoud ontwerp	--	-	++	+
# don't care	++	+	-	-
Typische toepassing	Verschillend signaal sets en resets FF		Tijdelijk waarde onthouden	Tellers & frequentie-delers

Realiseren van de combinatorische logica

We zullen nu de **realisatie** van de **combinatorische logica** vergelijken voor de **verschillende typses flip-flops**. We zien dat hoe **meer don't cares**, hoe **eenvoudiger de combinatorische** schakeling.

We zullen als voorbeeld deze exitatietabellen uitwerken :

Bepaal de excitatiefuncties voor FF 1

Huidige toestand	Volgende toestand / Uitgang		
	CD=0X	CD=10	CD=11
00	00/0	01/0	10/1
01	01/0	10/0	00/0
10	10/0	00/1	01/0

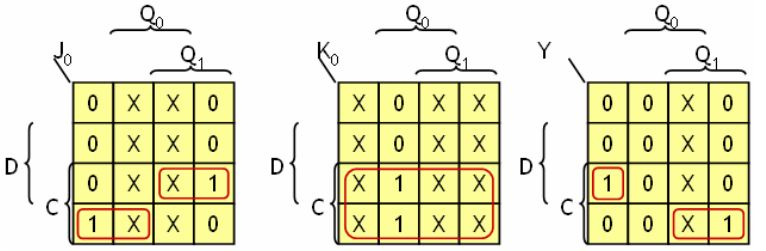
Bepaal de excitatiefuncties voor FF 0

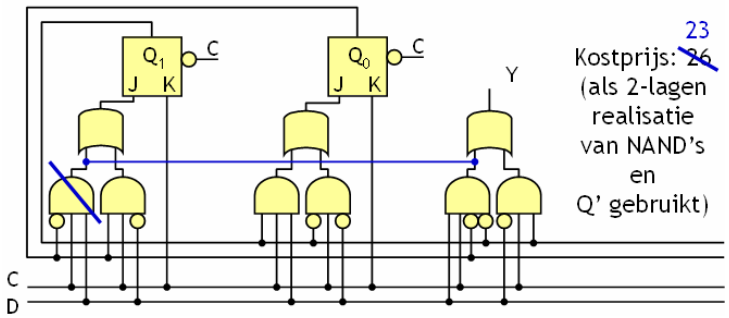
Huidige toestand	Volgende toestand / Uitgang		
	CD=0X	CD=10	CD=11
00	00/0	01/0	10/1
01	01/0	10/0	00/0
10	10/0	00/1	01/0

⇨ **JK flipflops** :

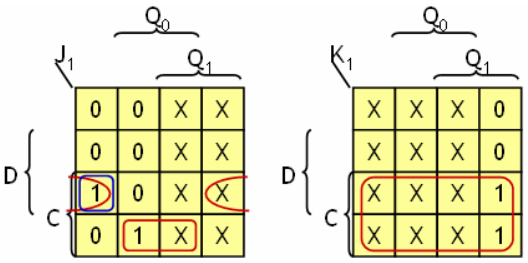
Excitatietabel JK-FF

Q	Q(next)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0



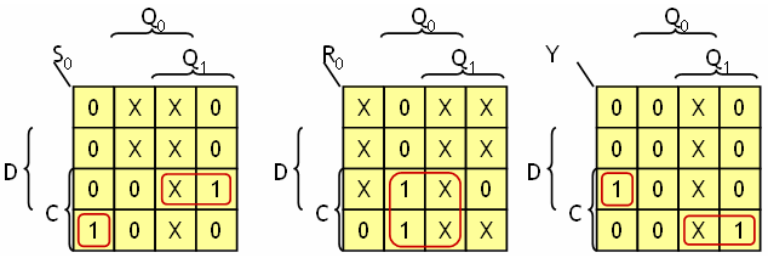


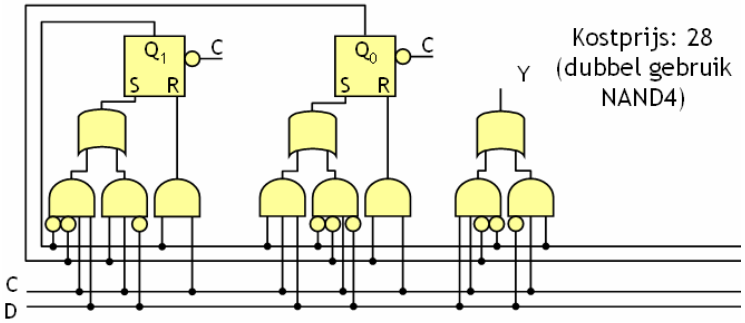
Kostprijs: ~~26~~ 23
(als 2-lagen realisatie van NAND's en Q' gebruikt)



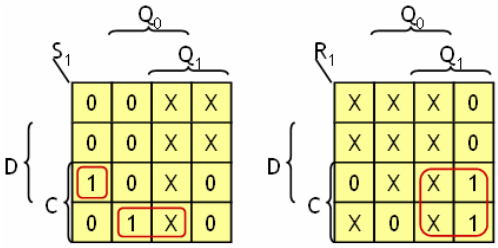
⇨ **SR flipflops** :

Q	Q(next)	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0





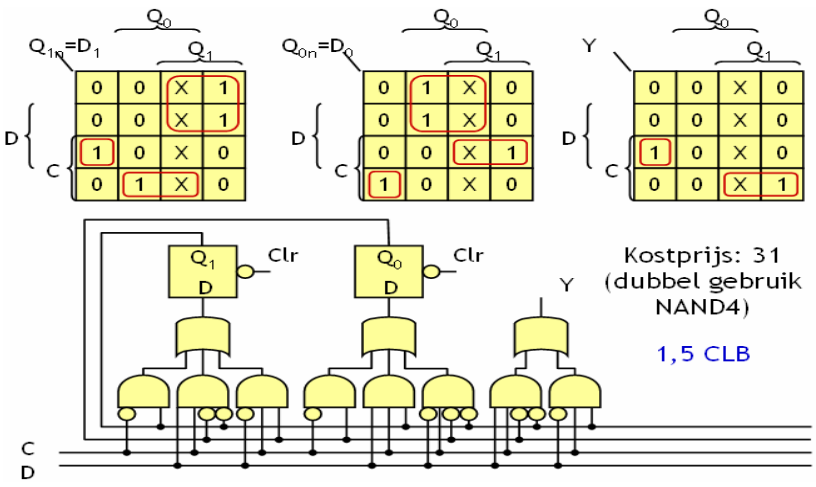
Kostprijs: 28
(dubbel gebruik NAND4)



⇨ **D flipflops** :

Excitatietabel D-FF

Q	Q(next)	D
0	0	0
0	1	1
1	0	0
1	1	1

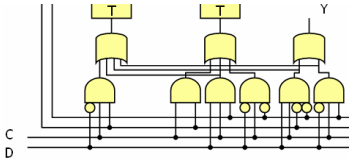


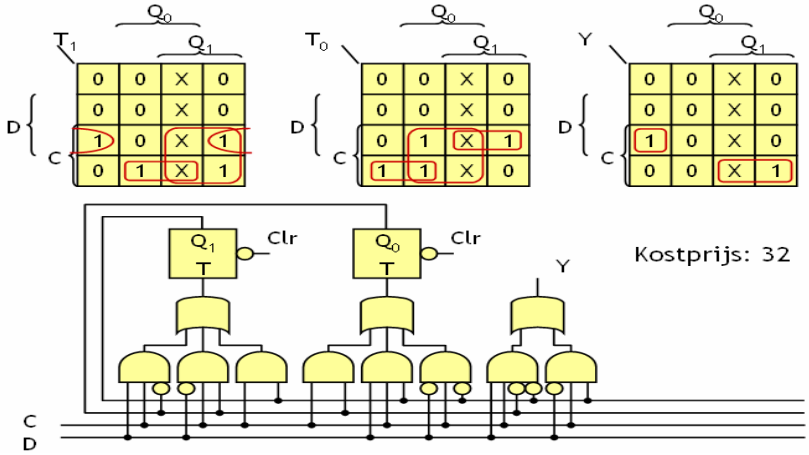
⇨ **T flipflops** :

Excitatietabel T-FF

Q	Q(next)	T
0	0	0
0	1	1
1	0	1
1	1	0

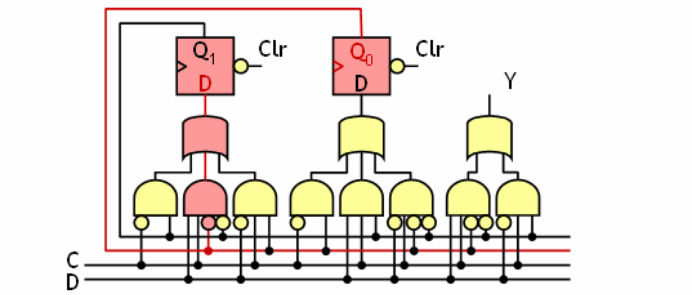
Of beter :





Analyse van het tijdsgedrag

De ***maximale klokfrequentie*** van een **sequentiële schakeling** bepalen we als de **inverse van de vertraging van het kritisch pad**.
⇨ Het maximum aantal klokcycli in een seconde is dus het aantal keer dat het kritische pad kan doorlopen worden in 1 seconde.
(kritisch pad is de langste combinatorische vertraging tussen 2 actieve klokflanken)

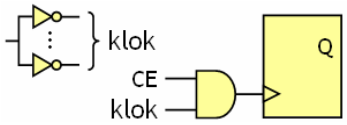


Wanneer we een chip **ontwerpen**, moeten we zijn **tijdsgedrag** zo goed mogelijk **analyseren** en de problemen die daarbij opduiken oplossen of verwaarlozen. We zullen nu een aantal **problemen** bespreken die in de tijd bij de werking van de chip zullen optreden. We moeten hiermee **rekening houden** en kijken of ze de **correcte werking van onze chip niet beïnvloeden** :

1) De ***'clock skew'*** is het **zeer minieme verschil** in de **tijdsogenblikken** waarop een **klokflank** op verschillende plaatsen in de schakeling waargenomen worden. (in verschillende stukken van de schakeling) Dit tijdverschil tussen de klokflanken in de verschillende flip-flops is te wijten aan :

⇨ de verschillen in de vertragingstijd van **combinatorische logica op de klokpaden** : bv. :

- er kunnen **buffers** zitten op het klokpad om de fan-out van het kloksignaal te vergroten
- een kloksignaal kan **samen met een CE signaal eerst door een AND** gaan ('clock enable', om de klok af te zetten voor die ff) en dan pas in de flip-flop komen.



⇨ Verschillende **stijg/daaltijden** t.g.v. **verschillende capacitieve belasting** van de kloklijnen. Elke kloklijn heeft een weerstand R en een capaciteit C met de grond, dus RC-gedrag.

⇨ Verschillende **looptijd** van het kloksignaal over een **verbinding** door :

- een draad is een **transmissielijn** bij hoge frequenties : verschillende draden hebben verschillende vertragingen
- verschillende doorlooptijden van de **schakelmatrices** van de FPGA

⇨ Verschillende ff's triggeren op een verschillend triggerniveau (trigger op verschillend voltage)

2) We moeten ***glitches*** proberen te vermijden

3) De ***metastabiliteitsproblemen*** :

Metastabiliteit is een **toestand** waarin de schakeling een bepaalde **stabiele waarde op een uitgang** heeft die **niet overeenkomt met een logisch niveau** (0, 1 of Z).

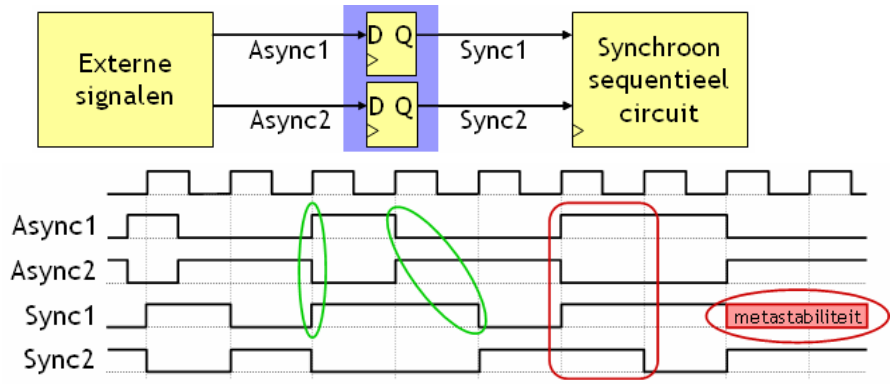
Metastabiliteit kan **ontstaan** in volgende situaties :

⇨ bij **slecht ontworpen synchrone schakelingen** (dit is de metastabiliteit van de flipflops die we vroeger hebben uitgelegd)

-> oplossingen : **lagere klokfrequentie** nemen of **snellere logica** waardoor de tijd waarin metastabiliteit kan ontstaan kleiner wordt.

⇨ bij **synchronisatie** : wanneer we **asynchrone externe ingangen** willen gebruiken in een synchroon sequentieel circuit, moeten we deze eerst **synchroniseren** met de klok van het systeem. Dit doen we door ze eerst **door een D-ff** te halen. Een verandering in niveau van het externe signaal wordt dan pas in het interne circuit doorgelaten op een klokflank, zodat de **veranderingen van het signaal intern synchroon gebeuren** met die van de rest van de schakeling. Het externe signaal kan echter eender wanneer in de tijd veranderen, en dus ook binnen de **set-up- en de houdtijden van de ff**. Dit kan metastabiliteit van de ff tot gevolg hebben wanneer het externe signaal samen met de klok veranderd.

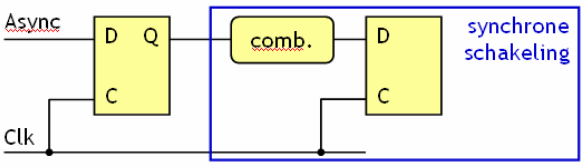
-> oplossingen : volgend blad



We zien dat wanneer 2 ingangn veranderen op een klokflank, dit verschillende effecten kunnen hebben : beide veranderen op de flank, beide veranderen pas op de volgende klokflank, ze doen verschillende dingen, of ze geraken metastabiel. Dit kan bij de D-ff enkel bij overgang van hoog naar laag wanneer de klok van laag naar hoog gaat, omdat dan beide veranderingen in het midden van het voltage kunnen stoppen op een metastabiel voltage.

$t_{meta} \leq t_r$ (metastability resolution time)

Probleem :



$t_r = t_{klok} - t_{comb} - t_{set\text{-}up}$

⇨ Het probleem doet zich voor wanneer de D-ingang veranderd van 1 naar 0 en de klok van 0 naar 1 gaat (of beide omgekeerd). **Klok en D-ingang veranderen dus samen en naar een tegengesteld niveau**. Beide kunnen dan blijven steken op een metastabiel voltage waarbij beide gelijk zijn.

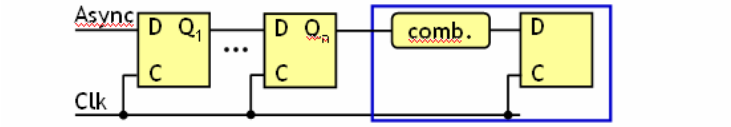
⇨ Door **ruis** zal dit **evenwicht verstoord** worden en door het niet stabiel zijn van het evenwicht zullen we **automatisch overgaan naar een stabiele toestand**. Het probleem lost zich dus na een tijd vanzelf op, maar in de tijd van metastabiliteit kan de schakeling wel volledig ovoorspelbaar reageren :

⇨ Het **metastabiele signaal** uit de eerste flipflop (**synchronisatie ff**) komt de synchrone schakeling binnen **in de combinatorische schakeling** en daarna in de **eerste ff** van het synchrone systeem. We moeten nu zorgen dat het **ingangssignaal niet meer metastabiel** is in de set-**up-tijd van de 2de ff** opdat de 2de ff niet metastabiel zou worden, en daarbij moeten we er rekening houden dat dit signaal vertraagd wordt door de combinatorische logica.

⇨ De **tijd** waarin het signaal uit de eerste flipflop dus **metastabiel mag zijn** (t_{meta}) moet dus kleiner dan t_r (de metastability resulotion time) met $t_r = t_{klok} - t_{comb} - t_{set\text{-}up}$. We hebben dus een hele klokcyclus de tijd om te stabiliseren, min de set-up-tijd en vertraging van de combinatorische logica. Hoe **groter** t_r, hoe **minder kans** op metastabiliteit.

We kunnen helaas niet kiezen hoe lang een signaal metastabiel blijft, en daarom zoeken we een oplossing :

Oplossingen :



$t_r = (n-1) \times (t_{klok} - t_{set\text{-}up}) + (t_{klok} - t_{comb} - t_{set\text{-}up})$

Om $p_{meta}(t_r)$ klein genoeg te maken volstaat n = 2 of 3

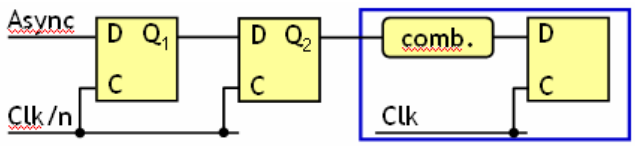
We zullen het **asynchrone signaal** eerst **door verschillende ff's** laten passeren (2 of 3) voor we het binnen brengen in het synchrone systeem. Wanneer nu de uitgang van de eerste ff metastabiel is, zal die hopelijk niet meer metastabiel zijn in de set-up-tijd van de 2de ff. Wanneer wel, wordt de uitgang van de 2de ff ook metastabiel, en we hopen dat dan deze metastabiliteit zichzelf oplost voor de set-up-tijd van de 3de ff, enz. We verhogen op deze manier t_r . We kunnen dit systeem op 2 manieren toepassen :

⇨ **snellere ff's** gebruiken met kortere set-up- en houdttijden (die verbruiken echter wel meer vermogen). Hierdoor wordt t_r groter. Ook snellere combinatorische logica helpt.

⇨ we kunnen een **zo traag mogelijke klok** gebruiken zodat de periodes van mogelijke metastabiliteit verder uit elkaar liggen. Ook dit maakt t_r groter.

We kunnen bv. onze **flokfrequentie kleiner** maken (pas om de n klokflanken een actieve klokflank nemen)

⇨ Toepasbaar wanneer de externe signalen niet veel veranderen t.o.v. klok en als er geen onmiddellijke reactie op verandering vereist is



$t_r = n \times t_{klok} - t_{set\text{-}up} + (t_{klok} - t_{comb} - t_{set\text{-}up})$

.

Ontwerp van Asynchrone sequentiële schakelingen

Bij **asynchrone schakelingen** kunnen **uitgangen & toestand veranderen zodra een ingang** verandert, **zonder te wachten** op de volgende **klokflank**.
We beperken ons tot de **fundamentele modus** : dit houdt in dat
- geen twee (of meer) ingangen mogen tegelijkertijd veranderen
- een ingangsverandering mag slechts optreden als alle effecten van de vorige verandering uitgestorven zijn

Dit is **toepasbaar** op **kleine asynchrone schakelingen** wanneer we bv. twee synchrone eilanden met verschillende klokken of klokken met een ongekende skew koppelen.

1) Toestandsdiagramma

Q	I	E	Toestand
0	0	0	a
0	0	1	b
0	1	0	c
0	1	1	d
1	0	0	onmogelijk
1	0	1	e
1	1	0	onmogelijk
1	1	1	f

- Een **toestand** van een **synchrone** schakeling is elke combinatie van zijn **uitgangen**. De ingangen hebben daar enkel effect op de uitgangen op de klokflanken.
- Een **toestand** van een **Asynchrone** schakeling is elke mogelijke combinatie van **ingangen en uitgangen**, omdat een verandering van een ingang een directe verandering van de uitgang tot gevolg gheeft.

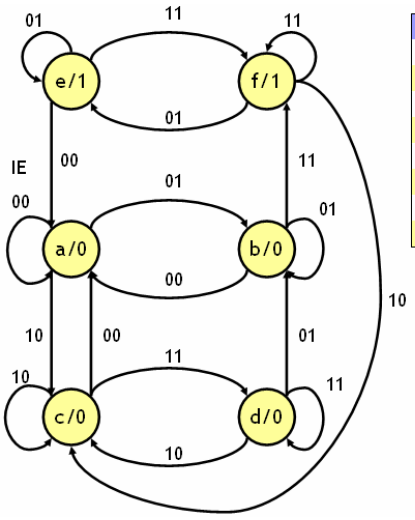
Bv. ontwerp een schakeling met één uitgang Q en twee ingangen I en E waarvoor :

⇨ E = 1 & stijgende flank van I ⇒ Q wordt 1 :
dus overgang b → f wanneer QIE overgang 001 → 111

⇨ Q blijft 1 tot E 0 wordt :
dus overgang e → a en of f → c waneer QIE overgang 1X1 → 0X0

⇨ E = 0 ⇒ Q = 0 :
dus de combinatie QIE 1X0 is onmogelijk

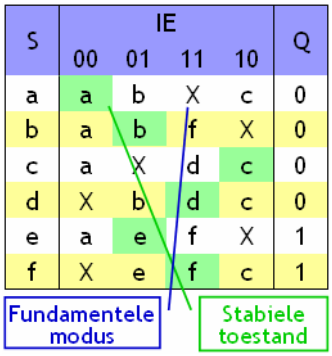
We kunnen dit **vertalen** naar het volgende **toestandsdiagram** :
Vanuit **elke mogelijke toestand** vertrekken **pijlen naar andere toestanden** met voor elke pijl de ingang die vereist is om die overgang te maken. Elke toestand heeft een bepaalde uitgang en bepaalde ingangen. Bij een verandering van de ingangen wordt overgaan naar de volgende toestand die overeenkomt met de pijl met die ingangscombinatie



Q	I	E	Toestand
0	0	0	a
0	0	1	b
0	1	0	c
0	1	1	d
1	0	0	onmogelijk
1	0	1	e
1	1	0	onmogelijk
1	1	1	f

Q wisselt als
- Q = 0 & 01 → 11
- Q = 1 & X1 → X0

2) Minimalisering aantal toestanden



S	IE 00	01	11	10	Q
a	a	b	X	c	0
b	a	b	f	X	0
c	a	X	d	c	0
d	X	b	d	c	0
e	a	e	f	X	1
f	X	e	f	c	1

We zullen ook hier proberen ons **aantal toestanden te beperken** door de **onmogelijke weg te laten**.
We stellen nu een **overgangstabel** op waarin we voor elke toestand aangeven naar welke volgende toestand we overgaan voor elke ingang.

⇨ Wanneer we hier met **fundamentele modulus** werken kunnen er nooit 2 ingangssignalen tegelijkertijd veranderen en hebben we ook hier een aantal overgangen die onmogelijk zijn.

⇨ De **stabiele toestanden** zijn de toestanden waarin we overgaan naar dezelfde toestand als vanwaar we komen, dus wanneer alles gelijk blijft.

Om het aantal **toestanden te minimaliseren** moeten we nu de **compatibele toestanden samensmelten**. Dit zijn de toestanden die voor **gelijke ingangen dezelfde volgende toestanden** en **uitgang** hebben of **don't cares**.

Dit is **meer restrictief** dan de **equivalente toestanden** van de synchrone schakelingen : compatibiliteit vereist namelijk het volgende extra :

⇨ de **toestanden** waar we naar **overgaan** moeten **gelijk** zijn voor compatibiliteit in plaats van alleen maar equivalent

⇨ dit is nodig **wegens don't cares** in de volgende toestanden

⇨ **compatibiliteit** is **niet associatief** (equivalentie wel!): compatibel(a,b) en compatibel(b,c) betekent **niet** dat compatibel(a,c)

S	IE 00	01	11	10	Q
a	a	b	X	c	0
b	a	b	f	X	0
c	a	X	d	c	0
d	X	b	d	c	0
e	a	e	f	X	1
f	X	e	f	c	1

S	IE 00	01	11	10	Q
α	α	α	f	c	0
c	α	X	d	c	0
d	X	α	d	c	0
e	α	e	f	X	1
f	X	e	f	c	1

S	IE 00	01	11	10	Q
α	α	α	γ	β	0
β	α	α	β	β	0
γ	α	γ	γ	β	1

In het **voorbeeld** zien we volgende compatibiliteiten :
a & b; a & c; a & d; c & d; e & f
Dit houdt dus in dat voor elk koppel alle IE's en Q's gelijk zijn, behalve voor de don't cares

We kunnen nu bv. a&b, c&d en e&f samennemen .

We **vervangen compatibele toestanden** nu door een **gemeenschappelijk teken**, omdat ze toch **gelijkaardig reageren**. Wanneer we geen compatibile toestanden meer hebben krijgen we de **transistietabel** : minimalisering van de toestandstabel

3) Codering

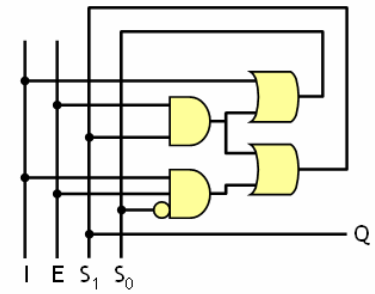
We zullen nu op dezelfde manier als bij synchrone schakelingen coderen.

S	IE 00	01	11	10	Q
α	α	α	γ	β	0
β	α	α	β	β	0
γ	α	γ	γ	β	1

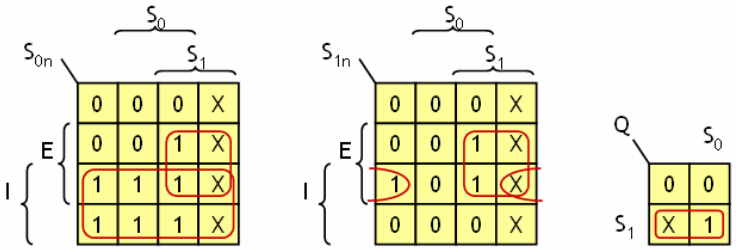
α : 00
β : 01
γ : 11

S	IE 00	01	11	10	Q
00	00	00	11	01	0
01	00	00	01	01	0
11	00	11	11	01	1

4) Realisatie



S	IE				Q
	00	01	11	10	
00	00	00	11	01	0
01	00	00	01	01	0
11	00	11	11	01	1

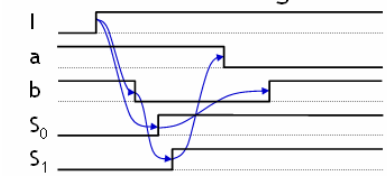


5) Analyse tijdsgedrag : Races & Hazards

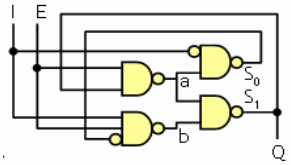
Een **Race** in een schakeling is het feit dat bij een **bepaalde toestandsovergang 2 of meer toestandsvariabelen** moeten **veranderen** voor **1 veranderde ingangsbit**. Het optreden van een race en het soort race hangt af van de **vertragings-karakteristieken** van de poorten.
Een race zelf leidt **niet tot een foute uitgang**.

vb race : wanneer E=1 en we I veranderen van 0 naar 1, gaan we over van toestand 00 naar toestand 11. Hier **veranderen** dus **beide toestandsvariabelen** terwijl slechts **één ingang** veranderd is. De **verandering** moet daarbij **doorgeven** worden van de schakeling van de ene variabele naar de andere.

Verwacht gedrag: 00 ⟶ 01 ⟶ 11
als a = 0 voor b = 1 terug



S	IE				Q
	00	01	11	10	
00	00	00	11	01	0
01	00	00	01	01	0
11	00	11	11	01	1

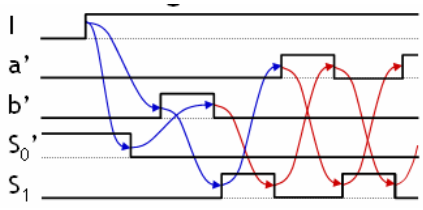


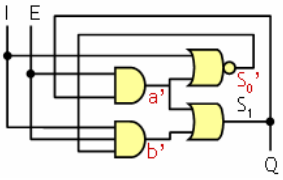
Nadelen van een race :

- het vraagt **veel tijd vraagt** om deze hele verandering door te voeren (grote vertraging) omdat eerst de eerste toestandvariabele veranderd en onder invloed daarvan pas de 2de.
- doordat eerst de eerste uitgang verandert en pas even daarna de 2de is de schakeling dus even **in een foute toestand**. Zulk gedrag dient vermeden te worden.

⇨ **Cycle** : Race die tot een oscillatie tussen twee toestanden leidt. Dit is volledig fout gedrag.

vb Cycle (andere schakeling !) : ook hier zien we dat door één ingangsvariabele aan te passen de beide uitgangsvariabelen om de beurt veranderen onder invloed van elkaar.

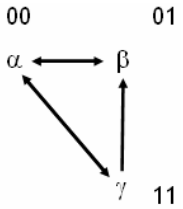




⇨ **Critical race** : Race die tot een verkeerde toestand leidt. Dit krijgen we in het vorige voorbeeld bijvoorbeeld wanneer de puls van b' ongeveer nul is.

⇨ We kunnen deze **'critical races' elimineren** door een **toestandscodering** te gebruiken die altijd **maximaal 1 toestandsvariabele** mag wijzigen t.g.v. 1 ingangsverandering. Dit is niet altijd mogelijk (zie voorbeeld)

S	IE 00	01	11	10	Q
α	α	α	γ	β	0
β	α	α	β	β	0
γ	α	γ	γ	β	1



⇨ Doe een zo **goed mogelijke toestandscodering**

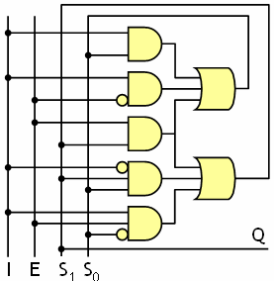
⇨ **Identificeer** de overblijvende **problemen**

⇨ Leid de **overgangen** die nog niet in orde zijn (meer dan 1 toestandsvariabele veranderen) om via bijkomende toestanden zodat niet langer 2 of meer toestandsvariabelen moeten wijzigen

S	IE 00	01	11	10	Q
00	00	00	10	01	0
01	00	00	01	01	0
11	10	11	11	01	1
10	00	X	11	X	X

Don't care omdat beide nieuwe overgangen een verandering van Q vereisen. Deze verandering mag dus al dan niet in de overgangstoestand gebeuren.

We maken in het voorbeeld een **extra toestand** bij tussen die waar er **2 toestandsvariabelen moesten veranderen** bij overgang. Deze dient enkel om de **overgang** door te geven op een gecontroleerde manier. Dit zorgt voor de volgende **implementatie** :



⇨ **Initiële toestand** : wanneer we de **stroom op het circuit aansluiten**, komt de schakeling in een **willekeurige toestand** terecht. Het is mogelijk dat deze willekeurige toestand zo'n **don't care** is. Deze don't cares zijn echter wel **ingevuld** eens de implementatie voltooid is, en dus is het mogelijk dat deze don't care een **stabiele toestand** op zich is geworden, hoewel die stabiele toestand **eigenlijk geen deel mag zijn** van de schakeling. Normaal komt de schakeling **nooit in die toestand**, maar **initieel zou dat wel kunnen**. We kunnen dus best de don't cares **zelf zo invullen** dat hij voor **eender welke ingangstoestand** naar een **stabiele toestand** gaat met dezelfde ingang. In ons voorbeeld moeten we zo zorgen dat het systeem initieel niet in de doorgeeftoestand blijft steken, want dan werkt het niet.

S	IE 00	01	11	10	Q
00	00	00	10	01	0
01	00	00	01	01	0
11	10	11	11	01	1
10	00	00	11	01	X

Race, maar niet kritisch

⇨ Bij **asynchrone schakelingen** moeten **hazards** op de **toestandsvariabelen** vermeden worden omdat ze tot een **verkeerde toestand** kunnen leiden

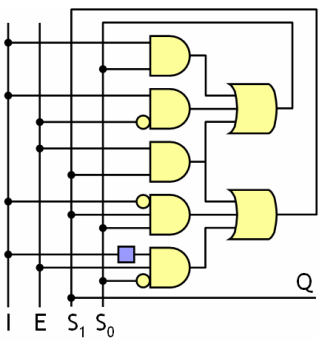
- Dit geldt zowel voor **statische als dynamische** hazards
- Bij **synchrone schakelingen** geldt dit niet omdat glitches daar moeten uitgestorven zijn voor de volgende klokflank

De vorige implementatie van het voorbeeld was reeds hazard-vrij

⇨ **problemen met skew op de uitgangen** : Stel dat er ergens in de schakeling **op 1 plaats een extra vertraging** zit (bv. door gebruik van een lange lijn). Dit kan de werking van de schakeling drastisch in de war sturen.
Bv. stel we starten met toestand 01 en ingangen IE 11 en we veranderen i van 1 naar 0.

- de **vooropgestelde werking** stelt dat de toestand veranderd **naar 00**
- de **reële werking** met de vertragin zorgt ervoor dat I voor de onderste poort later veranderd.

We zien dat de volgende verandering : **01 → 00 → 10 → 11**



⇨ **essentiële hazard** : toestand waarbij **één enkele ingangsverandering** de hele schakeling in een **verkeerde toestand** brengt. We kunnen dan voor **éénzelfde ingangs- en toestandscombinatie** overgaan naar **verschillende andere toestanden**.

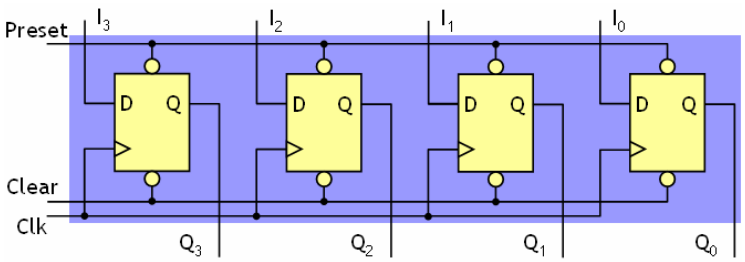
S	IE				Q
	00	01	11	10	
00	00	00	10	01	0
01	00	00	01	01	0
11	10	11	11	01	1
10	00	11	11	01	1

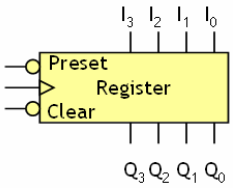
- We **herkennen** deze essentiële hazard aan het feit dat we **een andere eindtoestand** krijgen wanneer we de **ingang 1 of 3 keer veranderen**. (wanneer we in 01 de ingang veranderen naar 01, komen we in toestand 00. Wanneer we nu echter terug overgaan naar ingang 11 en dan weer naar 01, komen we in toestand 11, die verschilt van 01)
- We kunnen deze essentiële hazards **vermijden** door **toestandsvariabelen pas te wijzigen** wanneer **alle ingangsveranderingen** aan de **poorten gekomen** zijn. Dit vraagt een **zorgvuldig ontwerp** op electrisch niveau, bv door extra vertragingen in te voeren.

Basisbouwblokken op RTL-niveau

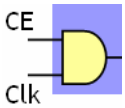
We zullen nu enkele essentiële onderdelen van systemen op register-transfer-niveau bespreken

1) Register





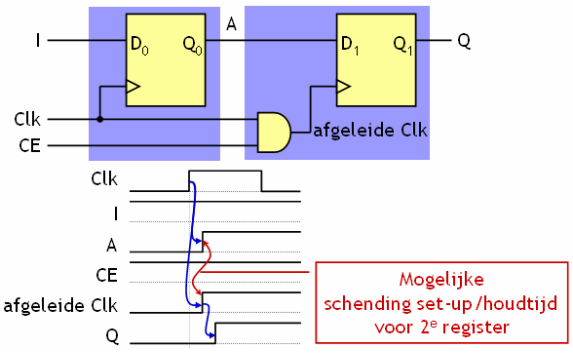
Een register is het **meest eenvoudige opslag-component** dat bestaat uit **n D-ff's** met **gemeenschappelijke klok**. Het register heeft dan **n ingangen** en **n uitgangen** om data in en uit de n ff's te laden. De **klok** komt in theorie overal in de ff's tegelijkertijd. We kunnen een **actief lage Preset en een Clear** toevoegen om alle ff's uitgang 1 of 0 te maken. Clear is meestal **asynchroon**, preset soms ook. Ze zijn dan onafhankelijk van de klok en de ingangen. (Deze preset en clear moeten dan even 0 worden : gebruikt bij initialiseren van de systemen).



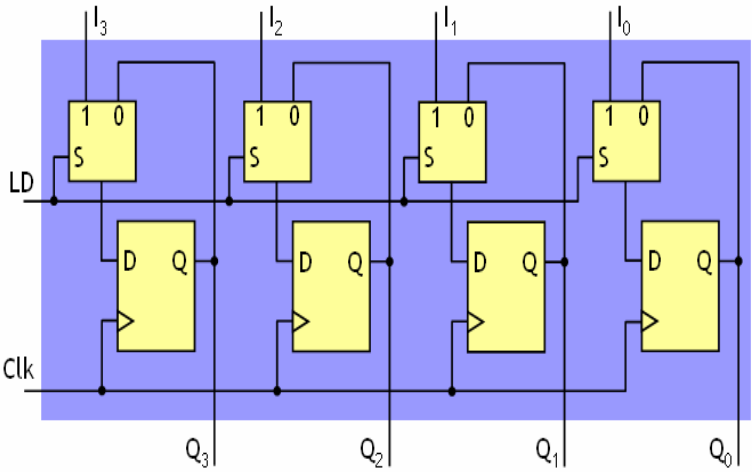
⇨ We kunnen een register **laadbaar** maken door de **klok** samen met een **enable** door een **AND** te laten passeren. We zullen dan **slechts nieuwe data** in het register laten komen wanneer de **CE (enable) 1 is.** Wanneer deze 0 is blijft de **vorige data opgeslagen** in het register.

Voordeel : we moeten de **ff's enkel schakelen** wanneer er een **nieuwe waarde** moet ingelezen worden in plaats van op elke klokflank, hetgeen **vermogenbesparend** is.

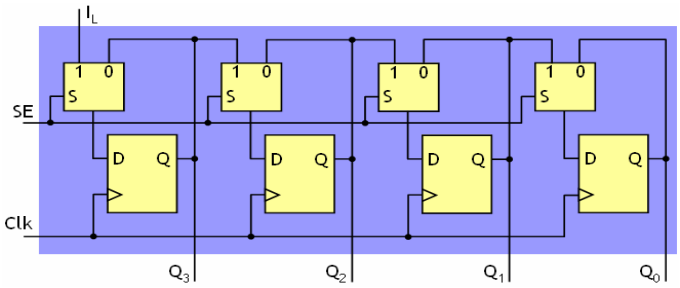
Nadeel : - de **CE** mag **alleen veranderen** wanneer de **clk=0**
- de **afgeleide** (gated) **klokken** zijn **gevoelig** voor **klok-skew**: dat wil zeggen dat niet alle klokken van alle registers tegelijkertijd veranderen. Wanneer we registers achter elkaar plaatsen is het mogelijk dat we door de vertragingen van de klok en van het eerste register de set-up of houdtijd van het 2de register schenden

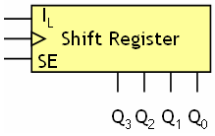


⇨ Een **oplossing** en meteen een **andere manier** om een **laadbaar** register te maken is met **selectors** : We sturen elke ingang eerst door een selector samen met de uitgang. De CE bepaald via deze selector of de nieuwe ingang de huidige uitgang is (huidige waarde behouden) of de externe ingang is (nieuwe data inladen). Dit **lost** de **klok-skew problemen op** doordat alle ff's dezelfde klok gebruiken, maar deze oplossing is **duurder** door de selectors en **verbruikt meer vermogen** omdat er elke klokflank geschakeld wordt.



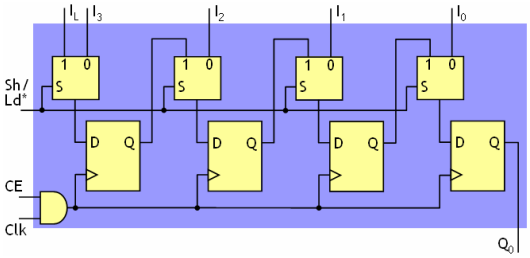
2) SchuifRegister





Een **schuifregister** is een register met **slechts 1 ingang**, waarbij telkens als de **ingang ingelezen** wordt **elke bit 1 ff opschuift**. We onthouden dus als het waren een **opeenvolging van bits**. De **laatste bit valt uiteraard telkens weg** wanneer er een nieuwe bit ingelezen wordt. We kunnen de **waarde van elke ff** uitlezen via **uitgangen**.
De realisatie gebeurt met **selectors** : we **selecteren** telkens als **nieuwe waarde** voor elke ff ofwel de **uitgang** van **dezelfde ff** (onthouden) ofwel de waarde van de **vorige ff** (alle bits 1 ff opschuiven). De selectors sturen we aan met de **ingang SE**, en deze bepaalt dus of er geshift wordt of dat de waarden gewoon behouden blijven bij de volgende klok.
⇨ **Toepassingen** : - Ontvangstregister (Rx) seriële poort
- Vertragingslijn FIR/IIR filters

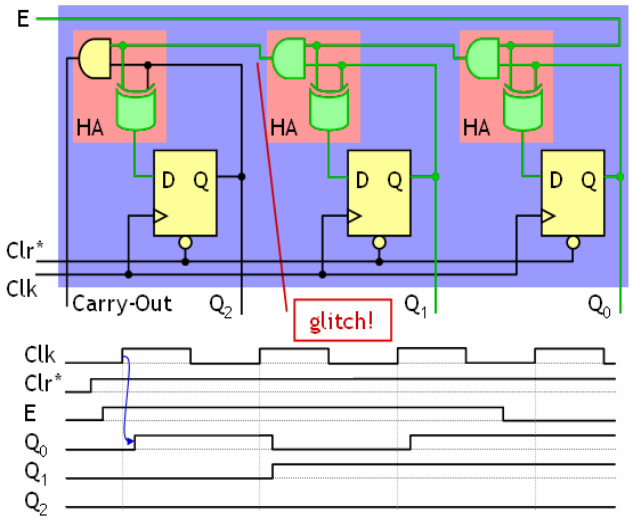
⇨ **2 Soorten** schuifregisters :
⇨ **Serial-In/Parallel-Out** schuifregister : 1 ingang en n uitgangen voor n ff's : We kunnen de waarde van elke ff uitlezen, en we kunnen slechts 1 bit data tegelijk setten. Dit is wat we net beschreven hebben



⇨ **Parallel-In/Serial-Out** schuifregister : we kunnen ook voor n ff's n ingangen voorzien en slechts 1 uitgang (voor de laatse ff). Met de ingang voor de selectors SH of LD kunnen we dan kiezen tussen 2 manieren om het register te gebruiken :
⇨ ofwel kunnen we voor elke ff nieuwe data inlezen via de n ingangen
⇨ ofwel gebruiken we het register als schuifrgister waarbij we telkens van de shift-ingang telkens een waarde in lezen op de klok en de andere telkens één ff opschuiven.
We kunnen in beide gevallen slechts de data van de laatste ff uitlezen. In beide gevallen kunnen we de klok nog door een AND sturen om te kiezen of we al dan niet schakelen
⇨ **Parallel-In/Parallel-Out** of **Serial-In/Serial-Out** : schuifregisters zijn een combinatie van de 2

3) Tellers

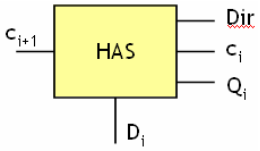
Synchrone tellers



Een **teller** heeft **verschillende uitgangen** die samen **binair een getal** voorstellen. De teller **telt** bij dit getal telkens 1 op op elke klokflank als zijn **ingang E** (enable) 1 is. Wanneer E 0 is gebeurt er niets. De teller heeft ook een **Clr (clear) ingang**, die de waarde van de teller terug **op nul zet**. **Synchrone teller** betekend dat **alle ff's aan éénzelfde kolksignaal** hangen. In deze tellers bestaat de kans voor **glitches**.

⇨ We kunnen ook een **ingang D** (**direction**) toevoegen die zorgt dat we voor D=0 telkens 1 **optellen** en voor D=1 telkens 1 **aftrekken**. Dit noemt met dan een **bidirectionele** teller. We moeten dan de **HA** (half-adders) vervangen door **HAS** (half-adder-substractor).

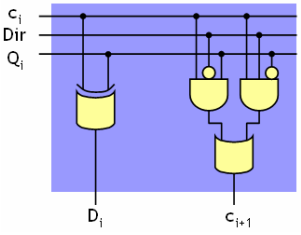
Werking HAS :



⇨ *Ingangen* :

- **Dir** is de **richting** van tellen : optellen 0 en aftrekken 1
- **Ci** is de **telEnable** waarbij we dus moeten tellen wanneer ci=1
- **Qi** is de **huidige waarde** van deze teller (onthouden door ff)
- **Di** is de **volgende waarde** (na het tellen) voor deze ff
- **Ci+1** is de **overdracht** naar de volgende teller en dus meteen de telEnable van de volgende (de volgende moet tellen wanneer deze overdracht heeft).

c_i	Dir	Q_i	D_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	1	0
1	0	1	0	1
1	1	0	1	1
1	1	1	0	0



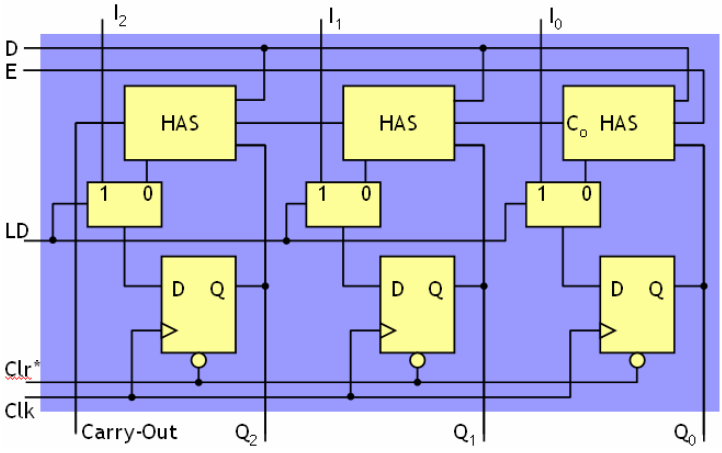
⇨ *Werking*

- **Bit i telt** (wisselt) als **ci = 1** ⇒ dan wordt $D_i = Q_i$ XOR c_i. (was de waarde van de teller al 1 dan wordt hij 0, anders wordt het 1)
- **Bit i+1 telt** als we op deze teller **overdracht** hebben of als we bij aftrekken **te weinig** hebben : dan moet $c_{i+1} = 1$:

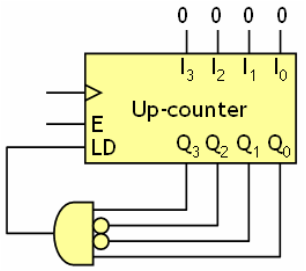
  voor **Dir=0** (optellen) moet $c_{i+1}=1$ als bit i 1→0 dus $c_{i+1} = c_i$ AND Q_i

  voor **Dir=1** (aftrekken) moet $c_{i+1}=1$ als bit i 0→1 dus $c_{i+1} = c_i$ AND Q_i

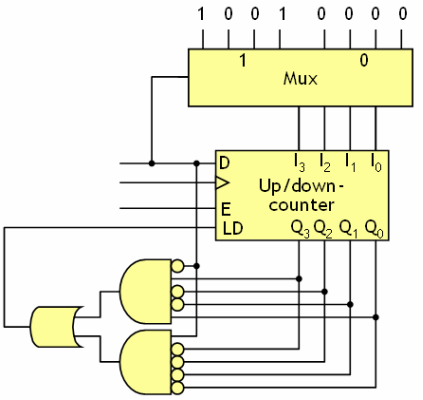
⇨ We kunnen ook **voor elke ff een ingang** voorzien en een **extra ingang LD** (**load**) waarmee we als **LD=1 stoppen met optellen** en gewoon de **ingangen in de ff's laden**. We voeren dan selectors in om te selecteren tussen de ingangen en de teller.



⇨ **BCD teller** is een teller die **binair modulo 10** telt. We tellen dus 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, … We kunnen dit **implementeren** met een gewone **4-bit teller** die **gereset** wordt op de volgende klok wanneer zijn **teller op 9** staat. Dit doen we met een **AND** poort die **1 geeft** wanneer de **teller 1001 heeft**. We zorgen op **alle ingangen** dan voor **0** en de **AND** hangen we op de **LD** zodat er **0 in de ff's geladen** wordt.

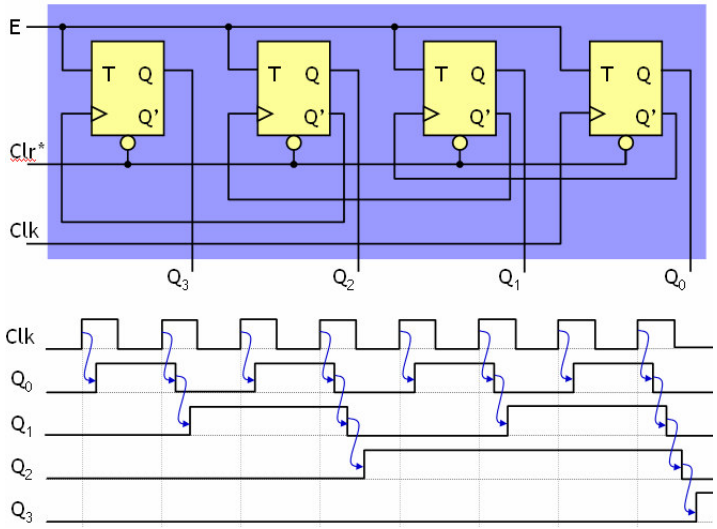


We kunnen ook en **BCD opteller-aftrekker** maken die reset naar 9 wanneer we 0 hebben en naar 0 als we 9 hebben. Hiervoor voorzien we een selector op de ingangen van de teller die selecteert tussen het laden van 0 en 9. We hebben dan ook 2 AND's nodig en een OR om te kijken of het 0 of 9 is een een aftakking van de richting (voor 9 en optellen wordt 0 en voor 0 en aftrekken wordt 9).



Asynchrone tellers

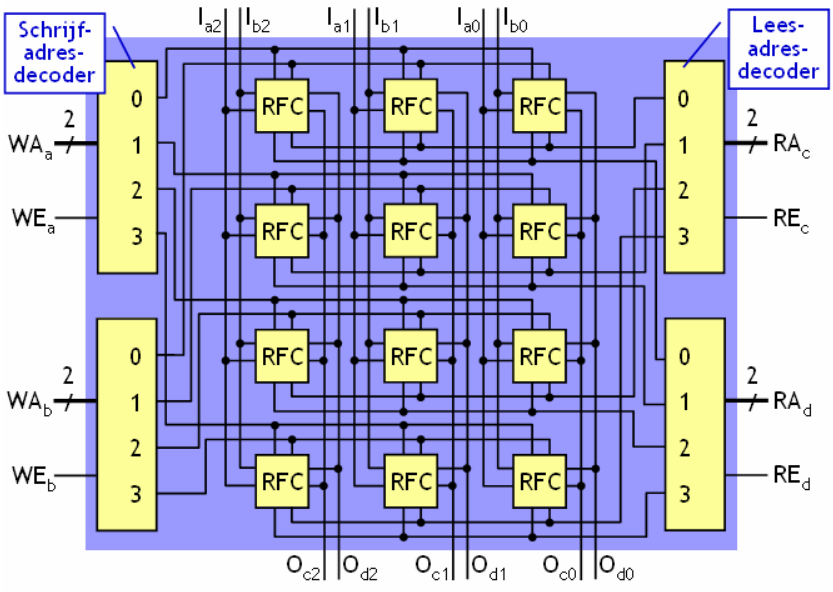
Een **asynchrone opteller** is een teller waarbij **niet elke ff** op de **klok** zit aangesloten. We zetten de klok **enkel op de eerst ff** en verbinden telkens de **inverse uitg**ang van een ff met de **klokingang** van de volgende ff. Hierdoor zal elke **volgende ff** schakelen wanneer de **vorige van 1 naar 0 overgaat** (klokflank). Dit is een **perfect systeem** om **binair** te schakelen. Telkens we overgaan van 1 naar 0 hebben we **overdracht naar de volgende ff,** die dan waarde 1 krijgt. Wanneer er nog eens overdracht plaatsvindt wordt de volgende ff terug 0 en wordt die daarop 1, enz. We gebruiken hiervoor **T ff's**. Deze schakelen om op een stijgende flank van de klokingang.



4) Geheugen : registerbank en RAM

Een **registerbank** is een **geheugenelement** dat een bepaald aantal bits kan opslagen. We kunnen het beschouwen als een **verzameling van 2^n registers** die elk m bits kunnen opslaan. De registerbank kan telkens de **m bits** voor 1 register tegelijk inlezen op de **m ingangen** en de m bits van 1 register weergeven op de **m uitgangen**. De bank bevat daarvoor **2 decorders** (multiplexen) om, aangestuurd met **n adresingangen**, te selecteren in welk register de m ingangswaarden moeten geschreven worden of de m uitgangen uitgelezen moeten worden.

Ingangen en Uitgangen :



⇨ **m ingangen D_{in}** om m bits te schrijven

⇨ **n adresingangen** WA (write adress) voor het schrijven die bepalen in welk register de m ingangen geschreven moeten worden

⇨ **1 WE ingang** (write enable) die 1 wordt wanneer er geschreven moet worden (dus wanneer ingangen en adressen klaar zijn). Wanneer deze 0 is wordt er niets geschreven

⇨ **m uitgangen D_{out}** om m bits uit te lezen

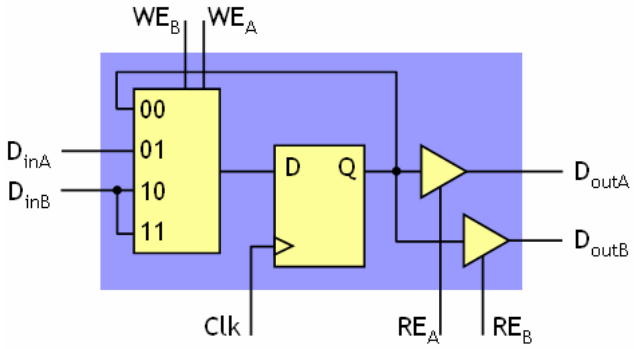
⇨ **n adresingangen RA** (read adress) voor het lezen die bepalen uit welk register de m uitgangen gehaalt moeten worden

⇨ **1 RE ingang** (read enable) die 1 wordt wanneer er gelezen kan kan worden (dus wanneer de adressen klaar zijn). Wanneer deze 0 is zijn de uitgangen hoog-impedant.

⇨ **1 klok** die gebruikt wordt om de ff's van de registers te schakelen (waarden worden getransporteerd en veranderd op de klok)

⇨ Het **voordeel** van de registerbank is dat er **minder draden** aan verbonden zijn dan aan 2^n apparte registers, en dat er **maar m ff's tegelijk** gebruikt worden dus dat we de bank veel **goedkoper** (met minder transistors) kunnen maken. We kunnen wel slechts aan **één van de 2^n registers tegelijk** aan. Registerbanken zijn een **snelle en dure** manier om geheugen te maken, en dus enkel gelimiteerd toegepast in microprocessoren en waar er snel en weinig geheugen van doen is.

⇨ Op de voorstelling zien we een **4x3 bit registerbank**. Hij heeft dus **4 registers** van elk **3 bits**. Er zijn **2 keer 3 ingangen** en **2 multiplexen** met elk 2 ingangen om elke 3 ingangen naar de juiste 3 RFC's te brengen. Analoog voor de uitgang. Het geheel bevat ook nog een klok, maar die is voor de overzichtelijkheid weggelaten.



Een registerbank bestaat dus uit 2^n keer m bits, dus uit net zoveel ff's. Deze ff's zitten in de basisbouwblokken van de registerbank, namelijk de **RFC's** (register file cell). Elke registerbank bestaat dus uit 2^nx m RFC's, een **inputdecorder** en een **outputdecorder**. We zien nu dat de meeste RFC's **2 ingangen en altijd 2 uitgangen** hebben. De 2 **uitgangen** zijn gewoon beide dezelfde uitgang van de ff en kunnen met een **RE afzonderlijk hoog-impedant** gemaakt worden. (het aantal RE's is gelijk aan het aantal uitgangen zodat ze elk appart aan of uit gezet kunnen worden). De 2 **ingangen** worden samen met de uitgang naar een **selector** gebracht die dan selecteert welke van de 3 opnieuw moet worden opgeslagen. (deze selectie gebeurt met de WE, die telkens één van deze mogelijke ingangen selecteert). Deze **2voud** van in en uit volgt uit de manier waarop **bewerkingen** gebeuren : wanneer we met de **m bits** van één register **m bewerkinen** willen uitvoeren, hebben we **2 keer m operandi** nodig (een bewerking combineert 2 getallen tot 1 nieuw getal). Vandaar de 2 uitgangen. We kunnen deze resultaten in dezelfde klokcyclus nog **terug opslagen** in dezelfde ff's. We moeten daarna wel in staat zijn om deze m resultaten te combineren met m nieuwe bits. Daarom moet een RFC in staat zijn te selecteren tussen 2 ingangen, namelijk tussen nieuwe input of resultaten van een bewerking.

RAM (random acces memory) is een **registerbank** waar **niet-sequëntiele geheugentoegang** mogelijk is. Het is een **wijzigbaar** geheugen dat meestal **vluchtig gebruikt** wordt (veel schrijven en herschrijven : dus niet voor data-opslag (ROM)).
Verschillen met de registerbank :

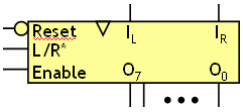
⇨ slechts **m in/uitgangen** in plaats van 2m : we gebruiken deze **om de beurt** als ingang of als uitgang, **afhankelijk van de RWS** (read-write-select). Wanneer RWS 1 is wordt er geschreven op de ingangen, bij 0 gelezen. Er is ook een ingang CS (chip select) waarmee de RAM geactiveerd wordt of gewoon passief blijft. Gebruikt wanneer we meerdere stukken RAM combineren tot één grote RAM.

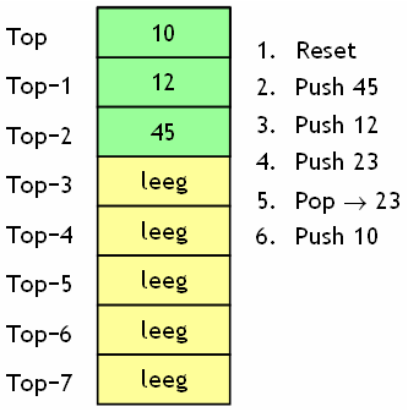
- we lezen (RE) dus als CS • R/W* (chip staat aan en we lezen wel of schrijven niet)
- we schrijven (WE) dus als CS • (R/W*)' (chip staat aan en we lezen niet of schrijven wel)

⇨ RAM **makkelijker groter te maken** dan registerbanken :

- RAM heeft **compactere RFC's** doordat het **minder poorten** heeft door dat er slecht dataverkeer in 1 richting tegelijkertijd toegestaan is (in of uit) en er dus minder adreslijnen en minder in-uit lijnen zijn. Statische RAM heeft ongeveer 4 à 6 transistors nodig per RFC, Dynamische RAM slechts één transistor (ook wel minder goed).
- RAM is **trager** doordat **schrijven en lezen niet tegelijk** gaat en RAM **opgebouwd** is uit **kleine delen** die niet tegelijk gebruikt kunnen worden. Niet bruikbaar dus wanneer er kleine tijdsverschillen zijn tussen tussenresultaten van bewerkingen.

5) LIFO : stapelgeheugen





Een **LIFO** (last in first out) is een **geheugen** dat een aantal bits kan **opslagen** en **weergeven** op een **speciale manier** : Met de **ingang Push** kunnen we telkens **1 bit toevoegen** : deze bit komt dan telkens bovenaan en alle vorige bits **schuiven dan telkens één plaats** op naar onder (soort schuifregister). We hebben nu ook een **ingang Pop** die de **laatst toegevoegde bit er terug uithaalt** : deze wordt dan **weergegeven** op de uitgang en al de andere bits **schuiven één plaats op naar boven**. Dit systeem is analoog geïllustreerd met getallen in de afbeelding.

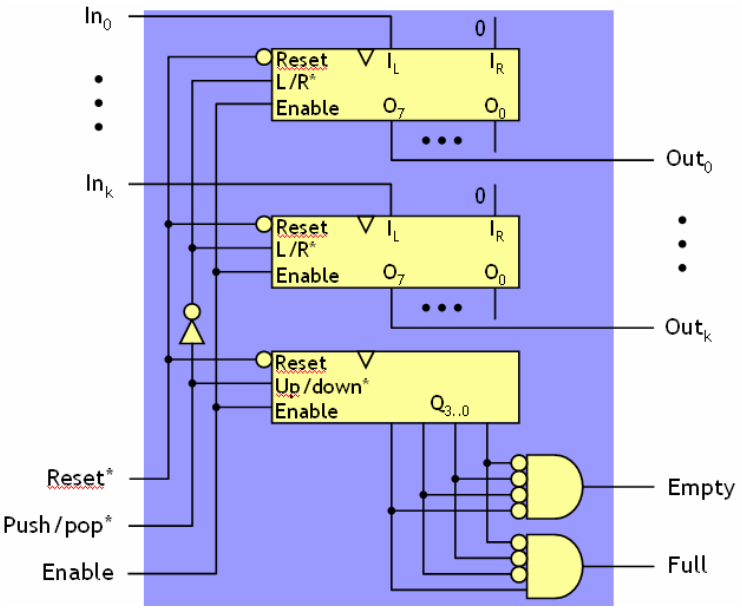
De **concrete realisatie** van een **LIFO** met **diepte 8** :

- ⇨ **enable ingang** die een **actie** op de volgende klokflank indiceert wanneer hij 1 is.
- ⇨ wanneer enable 1 is hangt de **actie** van de LIFO **af van de pop/push* ingang**. Wanneer deze 0 is zullen we Poppen, bij 1 Pushen.
- ⇨ voor een **bidirectionele LIFO** zullen we 3 ingangen voorzien om ofwel **links een waarde te kunnen pushen/poppen ofwel rechts**. De eerst **ingang is de L/R*** die kiest tussen rechts (0) en links (1), de 2 andere ingangen zijn de linkse of rechtse waarde die gepusht/gepopt moet worden.
- ⇨ **8 uitgangen** die de **waarde van de 8 geheugenplaatsen** weergeven.
- ⇨ **2 uitgangen** die weergeven of de LIFO **vol** is (full=1) of **leeg** is (empty=1)

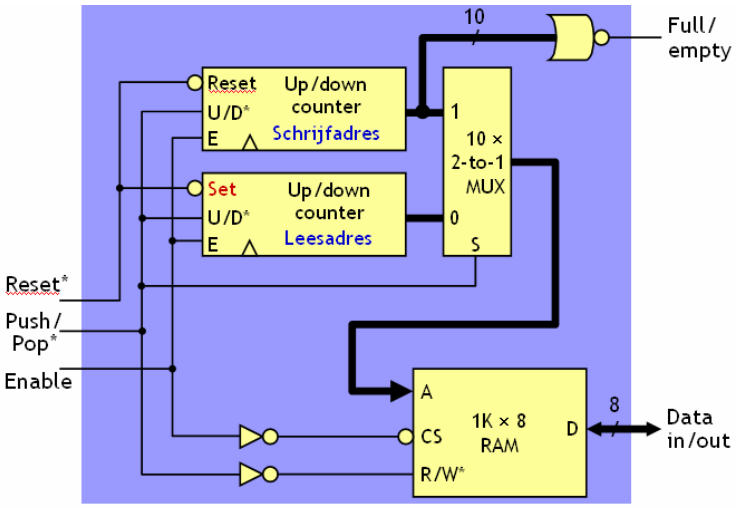
Enable	Push/pop*	Bewerking
0	0	–
0	1	–
1	0	Pop
1	1	Push

Teller	Empty	Full
0000	1	0
0001	0	0
0010	0	0
0011	0	0
0100	0	0
0101	0	0
0110	0	0
0111	0	0
1000	0	1

Implementatie van een LIFO **k keer diepte 8** : k enkelvoudige LIFO's met diepte 8 en een **teller** modulo 8 om de **empty en de full** weer te geven.



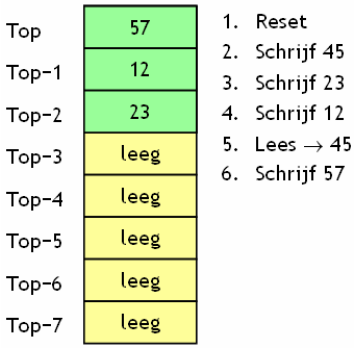
We kunnen een LIFO ook **implementeren zonder schuifregister** (zonder effectief bits te verschuiven) maar door **gewoon met schijf- en leesadressen te werken**. We laten **elke waarde gewoon staan**, maar houden bij waar we de **volgende waarde** moeten pushen of waar we de **vorige waarde** moeten halen om te poppen. Telkens we **pushen schrijven** we de waarde op de plaats aangeduid in de schrijfplaats en tellen we bij schrijf en lees 1 op. Telkens we **poppen halen** we eerst de waarde die we moeten poppen op en trekken dan 1 af van lees- en schrijfadres. We **implementeren** dit met 2 tellers die schrijf- en leesaders bijhouden en een RAM-geheugen om de waarden in te steken.

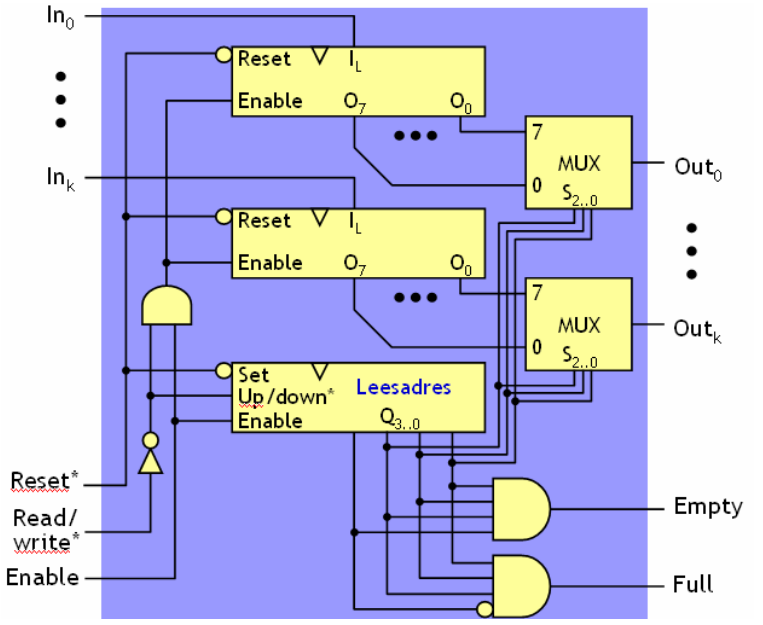


Als RAM niet geklokt is, controleer het tijdsgedrag van het (geklokte) adres t.o.v. CS en R/W* zodat niet 2 × gelezen/geschreven wordt!

6) FIFO : stapelgeheugen

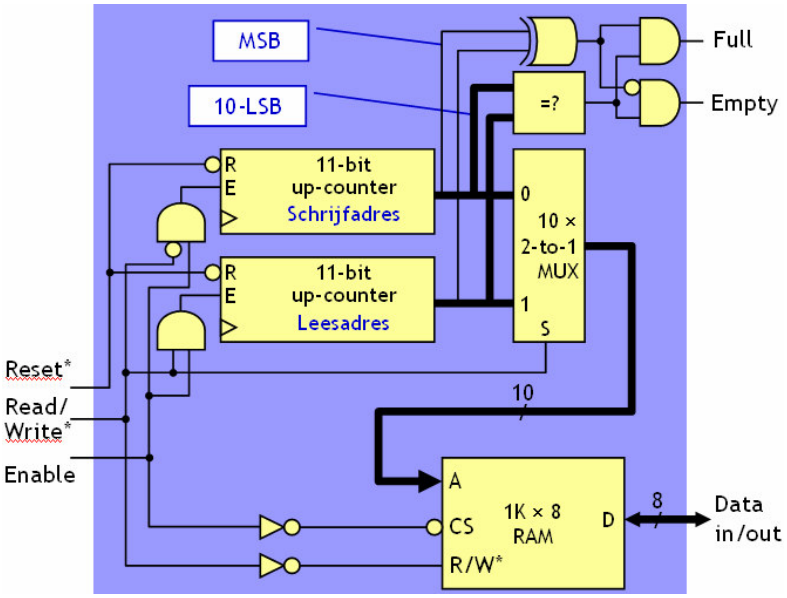
De FIFO (first in first out) werkt **zeer analoog** aan de LIFO, behalve dat we hier de **oudste waarde poppen** in plaats van de nieuwste. We halen dus telkens bij het poppen de **waarde op die al het langst** in de FIFO zit.
De **implementatie** is analoog, met dat verschil dat we hier telkens moeten **bijhouden met een teller** wat onze **eerste ingelezen waarde** was zodat we die terugvinden om te poppen. We moeten dus **tellen** hoeveel waarden er al in onze FIFO zitten (bv. 5 waarden, dan moeten we de 5de (onderste) eruit halen). Met een mux en de teller als sturing selecteren we de juiste uitgangen.



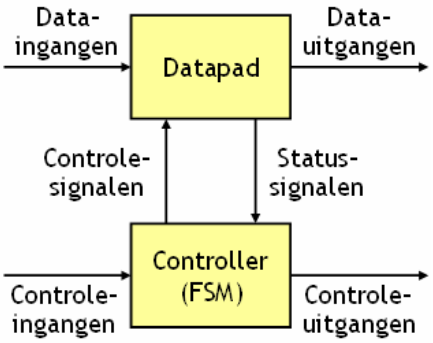


Ook hier geldt het **trucje met het veranderen van de lees- en schrijfadressen**, hetgeen de implementatie een pak gemakkelijker maakt.

In de werking van de FIFO is het soms belangrijk te weten wanneer de LIFO **helemaal vol is of helemaal leeg** is. Hiervoor wijzigen we de implementatie als volgt : we gebruiken **(n+1)-bit tellers** voor lees en schrijfadres bij een diepte van 2^n. We gebruiken dan **de n minst beduidende bits als adressen** voor de RAM. Wanneer de **n minst beduidende bits** van lees- en schrijfadres **gelijk** zijn is de FIFO **ofwel vol ofwel leeg.** We kijken dan naar de **meest beduidende bits** van de 2 tellers : zijn deze **gelijk dan is de FIFO vol**, zijn ze **verschillend is hij leeg**. Analoge oplossing voor de LIFO.



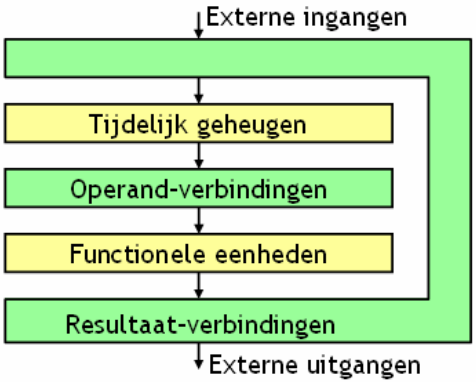
Hfdst 5) Niet-programmeerbare processoren



Een **niet-programmeerbare processor** is een **FSDM** (finite state machine with datapath). Hij bestaat uit een **datapad** voor de **berekeningen** (bewerkingen) en een **controller** die het datapad zegt wanneer **welke berekening** op **welke data** moet worden **uitgevoerd**. Voor een niet-programmeerbare processor voert de controller **altijd hetzelfde programma** uit (algoritme).

FSDM

1) Datapad



Het **datapad** voert op **commando** van de **controller bewerkingen** uit op **data die hij binnenkrijgt** en stuurt de **resultaten weer naar buiten**. Hij bestaat uit 3 delen :

1) ***Functionele eenheden*** (FU) die **datamanipulaties** uitvoeren: ALU, vermenigvuldiger, rotator, comparator, …Deze voeren effectief de **bewerkignen** uit
2) ***Tijdelijk geheugen*** voor tussenresultaten : register(bank), FIFO's, … Deze zijn de **directe geheugens** waaruit de data voor de bewerkingen in- en uitgehaald wordt.
3) ***Verbindingen :*** bussen met **multiplexers & 3-state buffers** om alle **data naar de juiste plaats** te brengen en om de juiste bewerkingen uit te voeren.

De **externe data** wordt **opgeslagen** in tijdelijk geheugen, daarna stelt de **controller** de **verbindingen** in om in de **functionele eenheden** de **juiste bewerking** op de **juiste data** uit te voeren, en daarna worden de **resultaten verwerkt**.

Constructie van het datapad :

⇨ elke **variabele & constante** komt overeen met een **register**
⇨ elke **operator** komt overeen met een **functionele eenheid**
⇨ maken van de **verbindingen** :
- registeruitgangen verbonden met de ingangen van de functionele eenheden
- de uitgangen van de functionele eenheden moeten weer naar de registeringangen

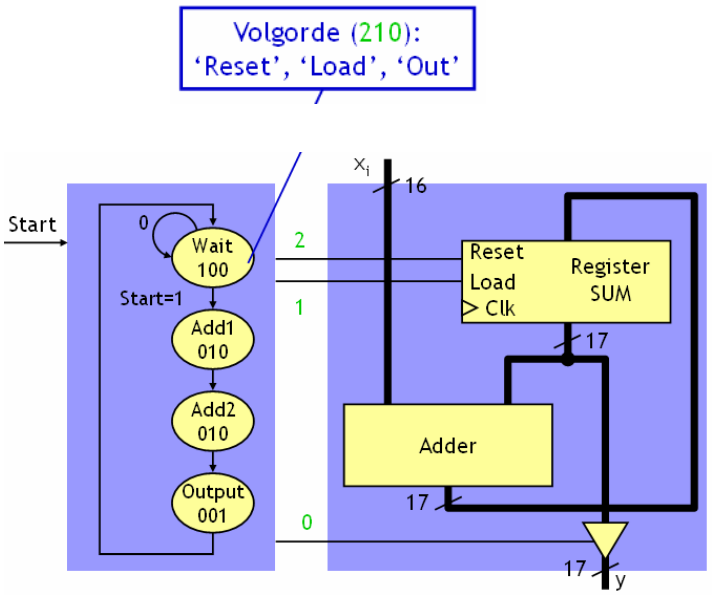
We gebruiken hiervoor **MUX** (multiplexen) of **bussen** met 3-state buffers als meerdere uitgangen verbonden zijn aan één ingang

⇨ Wanneer we een **algoritme** ontleden moeten we onderscheid maken tussen de **controle-commando's** en de **bewerking-commando's.** De controle-commando's zijn voor de **controller**, de bewerkingen voor het **datapad**. We moeten het datapad zo ontwerpen dat het alle vereiste bewerkingen kan maken.

Taak: $y = \sum_{i=1}^{2} x_i$

Algoritme:
```
sum = 0
FOR i = 1 TO 2
    sum = sum + x_i
ENDFOR
y = sum
```
Bewerking
Controle

⇨ We werken het **vorige voorbeeld** uit :
We willen op een **uitgang y** de som van **2 keer data op de ingang x**. We hebben dan **4 toestanden**. Een toestand die wacht op eerste input van x en onze som 0 stellen, een toestand waar de som het eerste getal is en we wachten op 2de invoer van x, een toestand waar we het 2de getal optellen bij het eerste en een toestand waar we op de uitgang het resultaat tonen. We kunnen dit realiseren met een **register dat de som SUM** bijhoud en een **opteller** die telkens de ingang optelt bij de waarde in het register en dat terug in het register steekt. Aan de **uitgang** hangen we **buffers** zodat we de uitgang aan en uit kunnen zetten.



De **controller** moet dit proces **coördineren** en heeft **3 commando-signalen : Reset, Load en Out**. **Reset** reset het register wanneer we opnieuw beginnen (SUM=0). **Load** beveelt een bewerking : we tellen som op bij de ingang x en stoppen het in het register. **Output** beveelt dat de output klaar is om getoond te worden.

⇨ Een **2de voorbeeld** : een **algoritme** dat het **aantal 1-tjes telt** in een reeks bits (woord). We hebben hiervoor een **variabele OCnt** die het aantal **telt**, een **Data** die de **invoer** voorstelt, en een **Mask** die zegt wat we zoeken. We doen dan zolang we invoer hebben het volgende :
We **vergelijken data met Mask** en steken dat in **Temp** (is 1 wanneer er een 1 in Data stond). We **tellen** dan **Temp** op bij de teller **OCnt** en nemen een **nieuwe data-bit**.

Taak: tel het aantal bits dat 1 is in een woord

Algoritme:

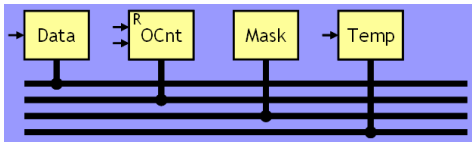
```
Data = Inport || OCnt = 0 || Mask = 1
WHILE Data <> 0 DO
    Temp = Data AND Mask
    OCnt = OCnt + Temp || Data = Data >> 1
ENDWHILE
Outport = OCnt
```

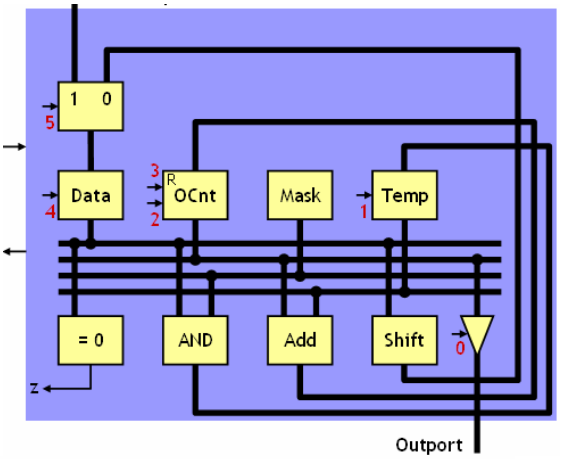
Opmerkingen over de code :

- ⇨ **Parallelle** ('concurrente') **beschrijving van de commando's** : alle instructies gescheiden door '||' worden tegelijkertijd uitgevoerd. (in 1 klokcyclus) We moeten deze dus tegelijk kunnen uitvoeren in het datapad
- ⇨ **Belangrijk verschil met software programma**: programma-instructies worden sequentieel uitgevoerd, slechts één op een bepaald ogenblik. Deze code voert zoveel mogelijk dingen tegelijkertijd uit: hardware componenten werken altijd (niet afzetbaar) en in parallel
- ⇨ **Compromis** mogelijk: **snelheid/performantie ↔ kostprijs**. Hoe sneller en beter de uitvoer van het algoritme hoe duurder

Implementatie van het voorbeeld 2 :
Elke **variabele** is een **register** met daaraan een **bus**:

- ⇨ bus 1 is **Data**
- ⇨ bus 2 is **OCnt**.
- ⇨ bus 3 is **Mask**.
- ⇨ bus 4 is **Temp**.





Onder de bussen staan alle nodige **operaties** :

- ⇨ een **vergelijking** met 0 (om de lus te evalueren)
- ⇨ een **AND** om Mask en Data te vergelijken en in Temp te stoppen
- ⇨ een **Add** om OCnt en Temp op te tellen en in OCnt te stoppen
- ⇨ een **Shift** om de data 1 bit naar rechts te shiften en daarna Data terug naar de input van data te sturen door een selector. Deze selecteert ofwel nieuwe input ofwel de de oude input opnieuw

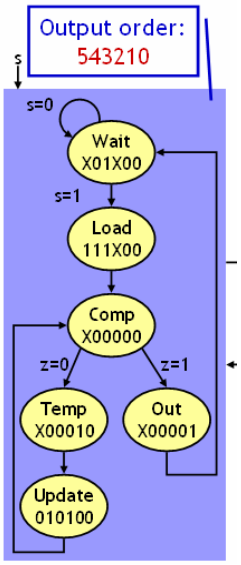
Taak: tel het aantal bits dat 1 is in een woord

Algoritme:

```
Data = Inport || OCnt = 0 || Mask = 1
WHILE Data <> 0 DO
    Temp = Data AND Mask
    OCnt = OCnt + Temp || Data = Data >> 1
ENDWHILE
Outport = OCnt
```

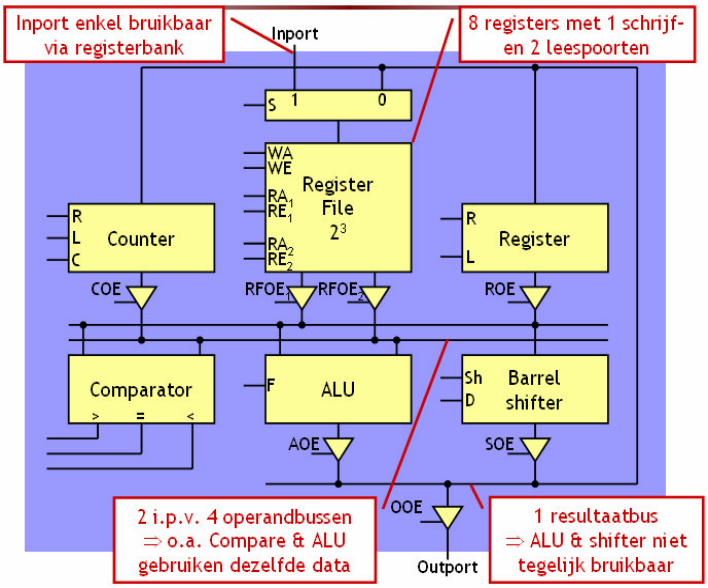
De **controller** :
Heeft deze keer **6 toestanden** :

- ⇨ **Wait** : we wachten op een signaal s dat 1 wordt wanneer we mogen beginnen tellen en wanneer er dus input is
- ⇨ **Load** : we laden de nodige variabelen Data, Ocnt en Mask (zie algoritme)
- ⇨ **Comp** : we vergelijken de Data met nul. Afhankelijk of data 0 is doen we 2 verschillende dingen :
- ⇨ is **Data niet gelijk aan 0** (dan is Data 1) dan gaan we over naar **Temp** en we steken in Temp het resultaat van Data AND Mask. Daarna doen we **Update**, hetgeen de 2de lijn van het algoritme uitvoert. We gaan terug naar Comp achteraf.
- ⇨ is **Data wel gelijk aan 0**, dan is het woord 1'tjes afgelopen en **beginnen we opnieuw**. We tonen dan het resultaat op de **uitgang** Out.

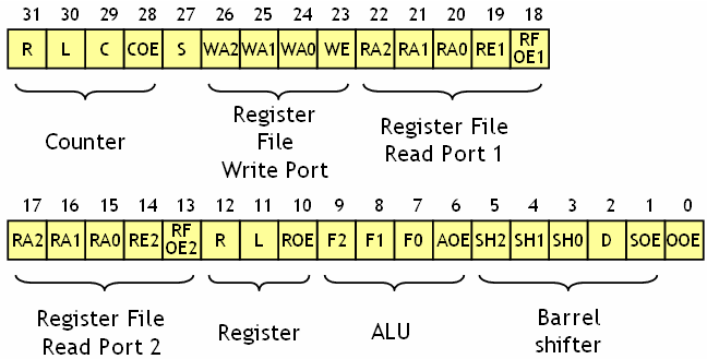


We kunnen de **implementatie** van het **datapad** nog wel **optimaliseren** :

- ⇨ we gebruiken **1 resultaatbus** ⇒ de **ALU** (Aritmische logische eenheid die het resultaat berekent) & de **shifter** kunnen dan **niet tegelijk gebruikt** worden maar dat is niet nodig. We hebben de shift toch enkel nodig wanneer er geen resultaat is dus zo besparen we een bus.
- ⇨ we gebruiken **2 i.p.v. 4 operandbussen** ⇒ dit omdat o.a. **Compare & ALU dezelfde data** gebruiken. We hebben **eigenlijk gewoon maar 2 bussen** tegelijk nodig.
- ⇨ **8 registers** (de register file) met **1 schrijf- en 2 leespoorten** : kan zijn data op de 2 operand-bussen zetten.
- ⇨ **Inport** enkel **bruikbaar** via **registerbank** : de data wordt eerst opgeslagen in een registerbank voor ze verwerkt wordt.



⇨ De **communicatie** tussen de **controller** en het **datapad** gebeurt via **instructiewoorden**. Elke **klokcyclus** stuurt de **controller** een **vast aantal bits** naar het datapad. Op voorhand is vastgelegd **welke aan te sturen component** welke **controlebits** nodig heeft en **waar** die in het **codewoord** staan. Bij **aankomst** in het datapad wordt het instructiewoord **in stukken gekapt** en gaat **elke bit** naar **desbetreffende component.**



⇨ een **geheugensteun** ('mnemonic') voor **bitpatronen** van (deel van) **instructiewoord**, bijv. ADD maakt het **herkennen** van de instructiewoorden **eenvoudiger en duidelijker.**

⇨ De **grootte** van het codewoord kan **verminderd** worden omdat **sommige operaties** niet **tegelijkertijd** kunnen: (tussen haakjes staat hoeveel bits we minder nodig hebben)

- 1e operandbus bevat ofwel Register File Read Port 1 ofwel Register Read Port (–1)
- 2e operandbus bevat ofwel Register File Read Port 2 ofwel Counter Read Port (–1)
- Register File REi = RFOEi (–2)
- ALU & Shift niet tegelijkertijd nodig: 1 bit nodig voor keuze operator en 4 bits voor controle operator (–2)
- als ALU gebruikt wordt mag zijn uitgang direct op de resultaatbus; idem voor Barrel shifter (–2)
- teller 'Count' & 'Load' zijn exclusief (–1)

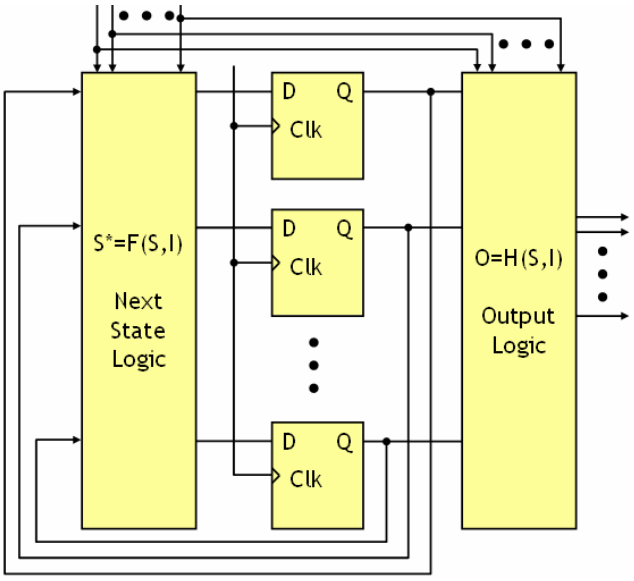
We kunnen zo tot 7 bits besparen in het instructiewoord

⇨ **Extra beperkingen** qua **parallellisme** zijn **mogelijk**, maar dit resulteert in een **grotere uitvoeringstijd**.

2) Controller

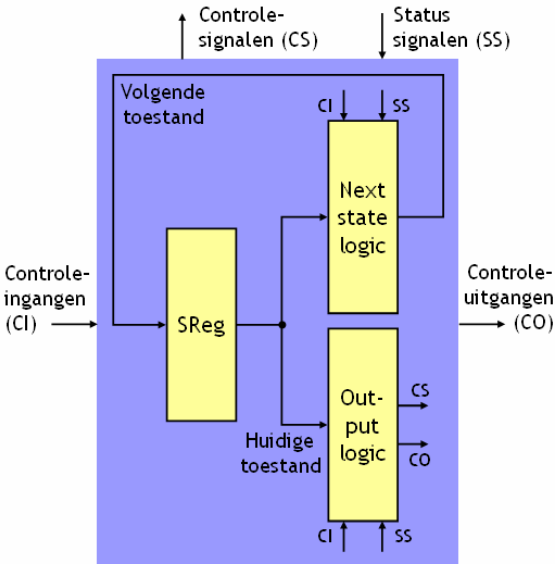
Sneller ontwerp voor grote FSM

De **standaard FSM-controller** ziet er als volgt uit : We hebben een **aantal ff's** die de **toestand** waarin we zijn onthouden. Op basis van de **uitgangen** van die **ff's** en de **ingangen** berekenen we de **uitgangen** (output-logic, eventueel voor datapad) en de **volgende toestand** (next state logic) die dan in de ff's wordt gestoken.



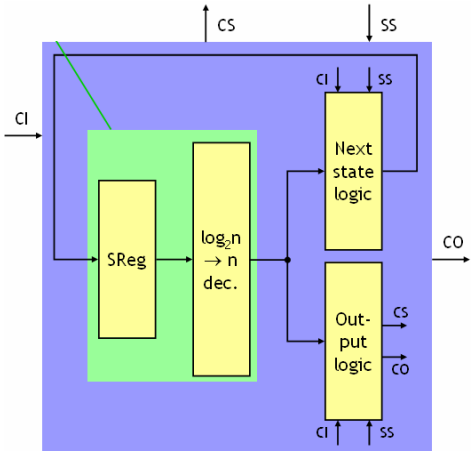
⇨ Voor **FSMD's hertekenen** we dit wanneer we met **grotere schakelingen** werken :
We hebben **ingangen CI** (controle-ingangen van de gebruiker) en
SS (status-signalen van het datapad).
We hebben **uitgangen CS** (controle-signalen voor de gebruiker) en
CO (controle uitgangen die het datapad besturen).

De **toestand** wordt **bijgehouden** in een **toestandsregister SReg**. Deze heeft voor **n toestanden** ofwel log2(n) geheugenplaatsen wanneer we minimum-bit-change codering toepassen hebben of gewoon n geheugenplaatsen voro one-hot.
Vanuit dit toestandsregister berekenen we samen met de 2 ingangen de **next state** (die dan in het register wordt geladen op de volgende klok) en **de 2 uitgangen**.

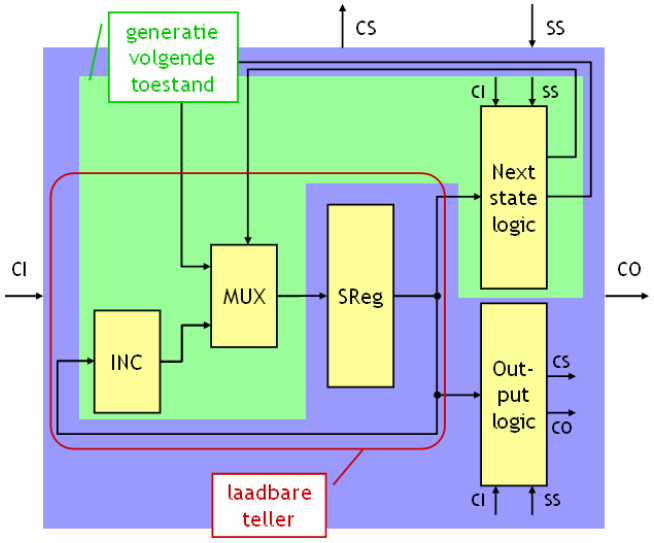


We kunnen **enige wijzigingen** aanbrengen om het geheel **sneller of goedkoper** te maken :

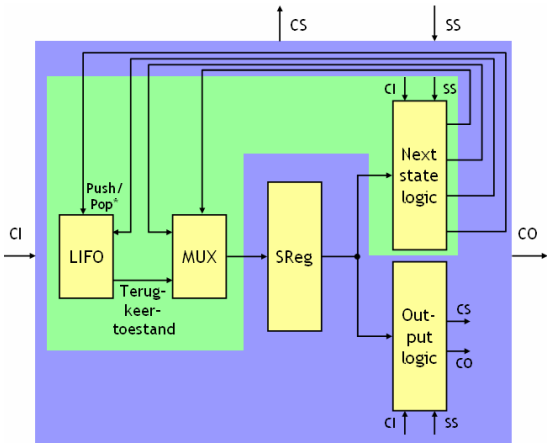
1) **Combinatie** van **one-hot codering** (eenvoudig ontwerp en beperkte logica) en **straightforward** codering (weinig ff's) :
We **slagen de toestanden op** in ff's in de vorm van **straightforward**, en **ontleden** die dan in een **logische blok** tot **one-hot-uitgangen.** We hebben dan en weinig ff's en eenvoudige en beperkte logica, met als enige prijs het omvormen van de straightforwarde ff's naar one-hot-uitgangen.

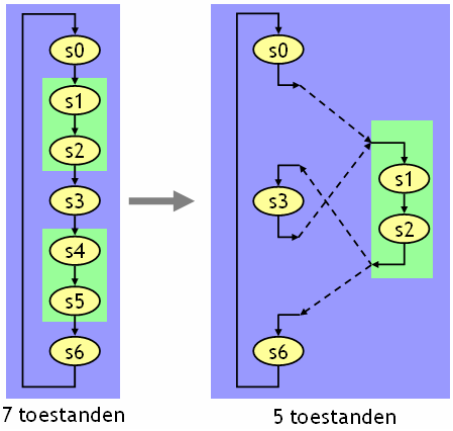


2) **Dikwijls volgen de toestanden mekaar onvoorwaardelijk op**, op een paar uitzonderingen na : de **volgende toestand** is dan de **vorige + 1**. We kunnen dan een **laadbare teller** gebruiken als toestandsregister. De 'Next state logic' wordt dan eenvoudiger: we moet enkel nog de **conditionele voorwaarden** bepalen om **over te gaan** naar de volgende toestand, en **niet meer de volgende toestand** zelf.

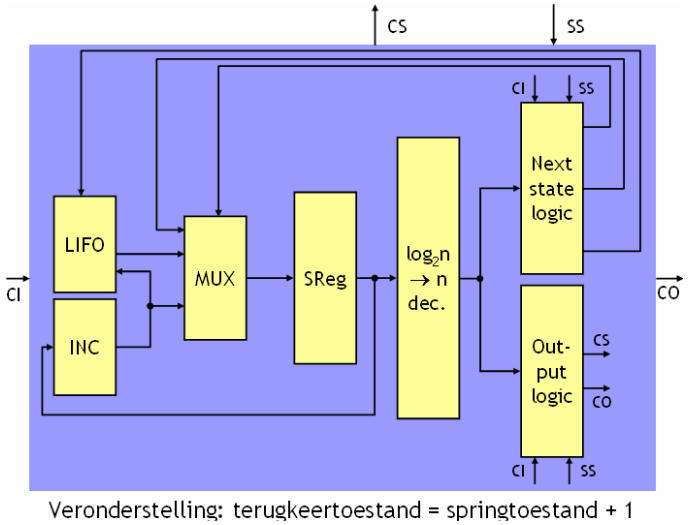


3) Soms bevat een toestandsdiagramma een **aantal toestanden die verschillende keren herhaald** wordt (**subroutines**). De **terugkeertoestand** (= toestand na het subroutine-einde) is echter **slechts gekend** op het **moment van uitvoering**. Hiervoor kunnen we handig een **LIFO gebruiken**.



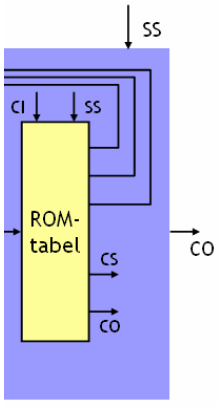


⇨ **Alle 3 de wijzigen tesamen** :

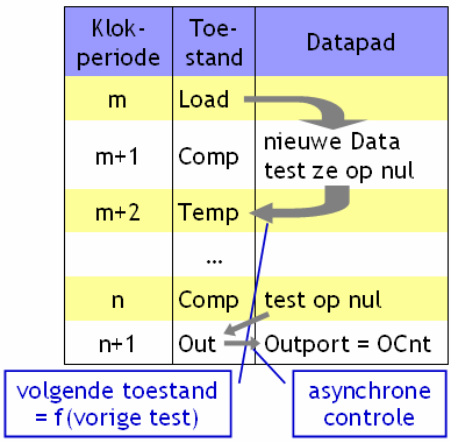


Veronderstelling: terugkeertoestand = springtoestand + 1

⇨ De **implementatie** van de **next state** en de **output logic** kan gebeuren met een **minimale AND-OR** implementatie volgens **karnaugh**, of met een **micorprogramma-controle** (een ROM-geheugen met waarheidstabel die gewoon de tabel volgt)



2) Tijdsgedrag



Interactie datapad - controller

Er zit altijd **1 klokperiode vertraging** tussen de **twee delen van synchrone systemen**. Neem nu de lus : op een **eerste klokflank** doen we een **bewerking** op data. Pas op de **volgende klokflank** kunnen we de **test** (om door te gaan met de lus) uitvoeren, want dan pas zijn we **zeker dat alle bewerkingen uitgevoerd** zijn. Eens de **test gebeurd** is, moeten we weer **op de volgende klokflank wachten** om een **volgende bewerking** uit te voeren. Elke bewerking duurt dus **in totaal 2 klokflanken**.

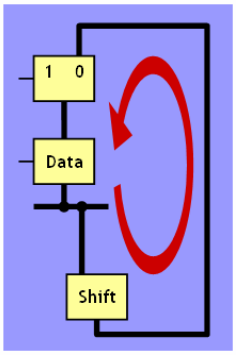
⇨ We kunnen dit **samenvatten** als **elke volgende toestand** is **functie van de vorige**. Om de volgende te bepalen moet de vorige volledig gedaan zijn en we moeten dus telkens een klokflank wachten. (tenzij we asynchroon werken natuurlijk)

⇨ Deze **2 klokflanken** zijn dus het resultaat van de **construtie datapad-controller**. Op de eerste klokflank schakelt de controller, dan het datapad, dan weer controller, enz.

Om deze **vertraging te vermijden** (wanneer dat echt nodig is) kunnen we 2 dingen doen :

⇨ Laat de **controller reageren** in **dezelfde klokperiode** als zijn **ingangen** :
we krijgen dan een inputgebaseerde controller

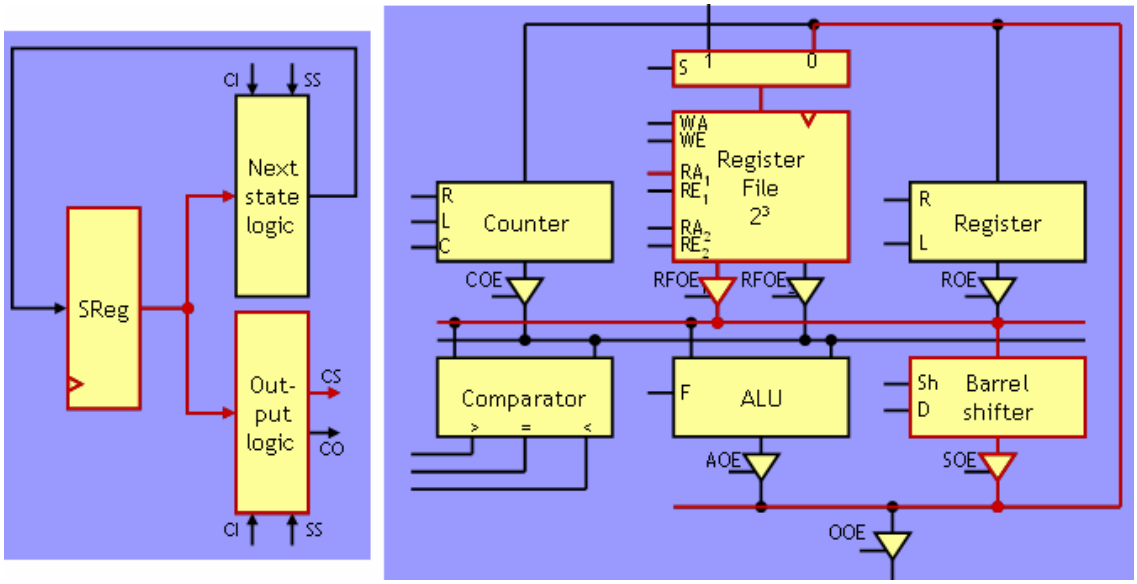
⇨ Laat **datapad reageren** in **dezelfde klokperiode** als zijn **controlesignalen** :
we moeten dan asynchrone controle-signalen gebruiken (LD, reset, …)



Let op voor terugkoppelingen: er bestaat gevaar op doorrimpelen!

Kritisch pad

Het kritische pad is hier het **langste combinatorisch pad** dat afgelegd moet worden door signalen **tussen 2 klokflanken** in **beide delen samen**. Voorbeeld zie tekening.



Beschrijving van een algoritme

1) Toestand-actie-tabel

Een **toestand-actie-tabel** beschrijft **in functie van de huidige toestand** de **volgende toestand** (vanuit de ingangen), de **uitgangen** en de **Datapadactiviteiten**. Zo'n tabel is vaak zeer **onoverzichtelijk** omdat :

⇨ dikwijls de volgende toestand slechts afhangt van enkele ingangen
(niet van allemaal, veel don't cares ook)

⇨ dikwijls de datapadvariabelen niet of niet veel wijzigen

Huidige toestand	Volgende toestand f(Start, Zero)				Uitgang	Datapadvariabelen			
	00	01	10	11	Outport	Data	OCnt	Temp	Mask
S_0	S_0	S_0	S_1	S_1	Z	X	X	X	X
S_1	S_2	S_2	S_2	S_2	Z	Inport	0	X	1
S_2	S_3	S_5	S_3	S_5	Z	Data	OCnt	X	Mask
S_3	S_4	S_4	S_4	S_4	Z	Data	OCnt	Data AND Mask	Mask
S_4	S_2	S_2	S_2	S_2	Z	Data >> 1	Ocnt + Temp	X	Mask
S_5	S_0	S_0	S_0	S_0	Ocnt	Data	Ocnt	X	X

Huidige toestand	Volgende toestand		Controle- & datapad-acties	
	Conditie	Toestand	Conditie	Acties
S_0	Start = 0 Start = 1	S_0 S_1		Output = Z
S_1		S_2		Data = Inport Ocnt = 0 Mask = 1
S_2	Data = 0 Data = 0	S_3 S_5		
S_3		S_4		Temp = Data AND Mask
S_4		S_2		Ocnt = Ocnt + Temp Data >> 1
S_5		S_0		Output = Ocnt

We kunnen deze tabel **aldus enigszins inkorten** en **vereenvoudigen** :
het geheel wordt een **compacte voorstelling van de toestandstabel**.

⇨ **conditionele tabel**: niet alle ingangscombinaties worden vermeld, we vermelden alleen de voorwaarden om naar de alle mogelijke toestanden over te gaan

⇨ **voor elke actie** geven we **alleen verandering** van **variabele** aan, en **niet de stand** van **alle** variabelen (de meeste blijven toch gewoon constant)

2) ASM schema

Een **ASM-chart** (algoritmic state machine chart) is een **visuele voorstelling** van de **toestand-actie-tabel** die veel **overzichtelijker** is dan de tabel. De chart heeft als **eigenschappen** :

⇨ **Eenvoudiger te verstaan** voor een mens

⇨ **Elke rij** in de toestand-actie-tabel komt overeen met een **ASM-blok** en dus met een **toestand**. In deze toestand worden de **ingangen gecheckt**, en op basis daarvan beslist **naar welke toestand we overgaan** (nieuwe ASM-blok). De ASM-blok moet ook de **uitgangen** en de **datapad-variabelen** definiëren voor zijn toestand.

⇨ Elke **ASM-blok** bestaat uit **één of meerdere ASM-elementen**. Er zijn 3 soorten ASM-elementen

⇨ **state box** : Komt in elke ASM **slechts 1 keer** voor aan de ingang en **definiëert de toestand** van die ASM-blok (elke toestand heeft dus zijn eigen ASM-blok en state-box). Hij bevat dus de **set onvoorwaardelijke toekenningen** van **uitgangen en datapad-signalen** die deze toestand kenmerken, en is dus **relevant** voor het **datapad**. (voor elke toestand zijn de uitgangen en datapad-commando's anders, en deze worden dus ingesteld met de state-box van de toestand)

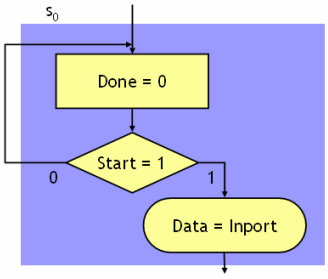
Onconditionele toekenningen

⇨ **decision box** : **Vanuit elke toestand** moeten we kunnen **overgaan** naar een **volgende toestand**, waarbij de **keuze** van de volgende toestand afhangt van de **status- of controlesignalen**. Hiervoor dienen deze **boxen**. Elke box kijkt naar **één ingang** en stelt deze een **conditie**. Wanneer de ingang **voldoet aan deze conditie** zullen we de **pijl 1** volgen, **anders pijl 0** (ingang voldoet niet). In **elke decision box** komt dus **1 pijl toe** en **vertrekken weer 2 pijlen**. Er zijn zoveel decision-boxen als er ingangen zijn, min het aantal don't cares.

⇨ **condition box** : Na een **decision box** kan het zijn dat we **bepaalde uitgangs- of datapadsignalen moeten veranderen**. (wanneer we bv overgaan naar een nieuwe toestand) Deze **toekenning of verandering** van signalen gebeurt met **condition-boxen**. Ze komen dus **enkel voor na decision-boxen**, zijn aldus **afhankelijk** van de **ingangen** op die manier en bestaan dus enkel bij **inputgebaseerde FSMD's**. Ze zijn enkel relevant voor het datapad.

Conditionele toekenningen

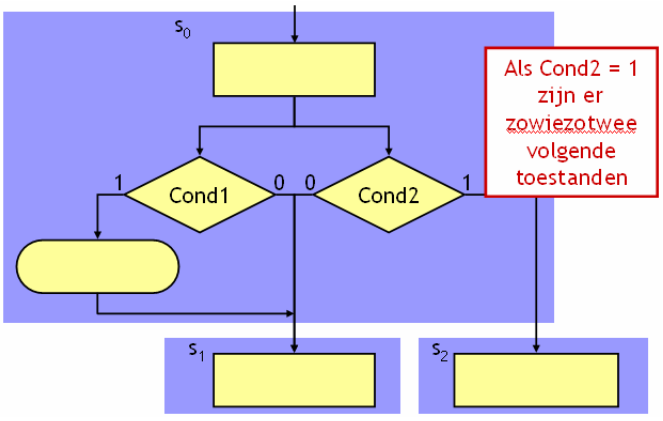
Deze ASM-blokken stellen **hardware** voor : alles wat sa**men in 1 ASM-blok staat** (dus 1 toestand) wordt **tegelijkertijd uitgevoerd**.

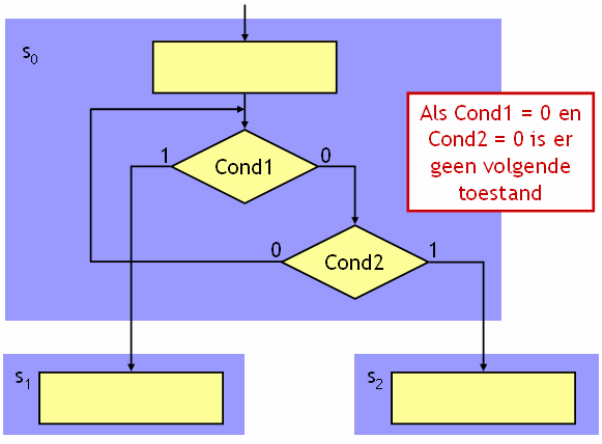


De ASM-blokken moeten aan een **aantal voorwaarden** voldoen om **geldig** te zijn (om een correcte werking voor te stellen) :

⇨ In een geldig ASM-blok moet **elke combinatie van ingangen** tot **exact 1 volgende toestand** leiden. Het kan dus niet dat er voor bepaalde ingangen 0 of 2 volgende toestanden zijn !

Voorbeeld van ongeldige ASM-blokken :





⇨ Wanneer er **meerdere uitdrukkingen** in **één kader** staan, worden ze in **parallel (tegelijkertijd) uitgevoerd**. Wanneer we een bepaalde variabele y **eerst willen berekenen** en dan **daarmee een andere bewerking willen doen,** moeten we dit **in 2 stappen** doen en mogen we ze niet samen in 1 kadertje zetten !

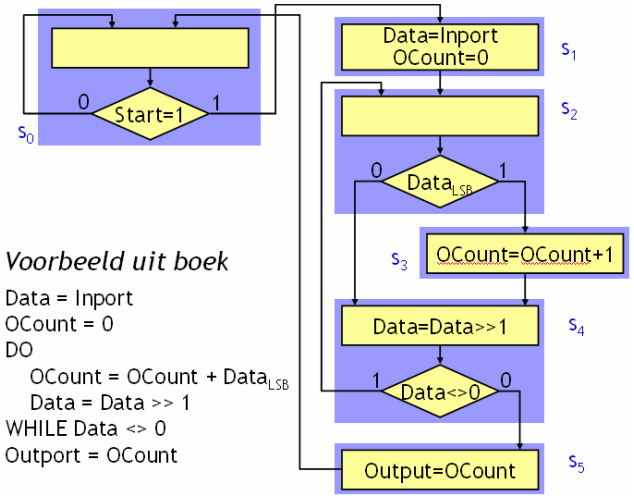
Programma:
y = x + 1
z = y + 2
≡
y = x + 1
z = y + 2
≠
y = x + 1
z = y + 2

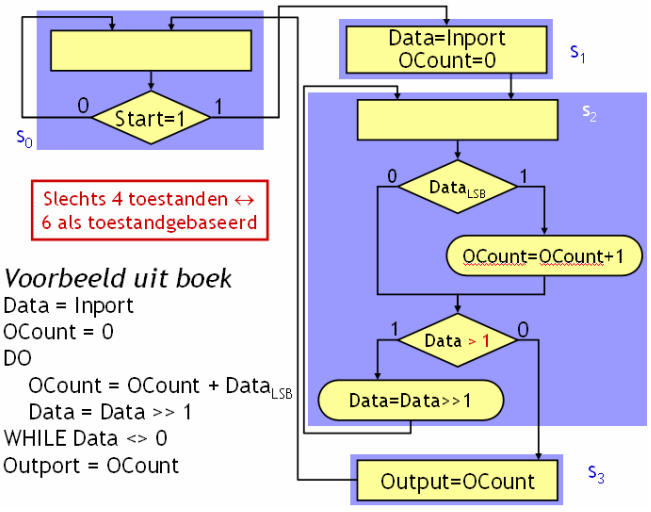
Vermits de bewerkingen in parallel uitgevoerd worden is hun **volgorde onbelangrijk**. Dit geldt niet alleen voor ASM-elementen maar ook voor een ASM-blok als geheel!

x = y + 1
y = x + 2
≡
y = x + 2
x = y + 1

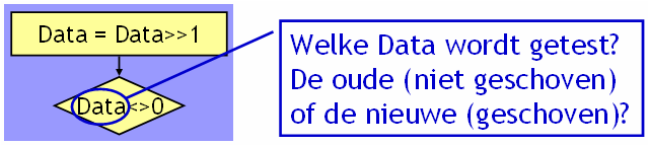
⇨ We zien **verschil** tussen **INPUTgebaseerde** en **TOESTANDSgebaseerde** ASM-charts. Zoals we boven vermeldde **conditionboxen enkel voor bij inputgebaseerde** ASM's. Verder zien we dat **inputgebaseerde** meestal **minder toestanden** en dus **minder boxen** hebben, maar dat de boxen wel **ingewikkelder** zijn. De **werking** van beide is wel **gelijk**, aangezien ze hetzelfde Algoritme uitvoeren.

Toestandsgebaseerd | *Inputgebaseerd*

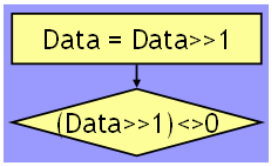




⇨ **Mogelijk probleem** : Wanneer we in **éénzelfde blok** eerst **een variabele v veranderen** en die daarna **nog eens willen gebruiken**, is de vraag of we de oude of de nieuwe waarde voor v zullen gebruiken voor de 2de bewerking. Aangezien **alle bewerkingen tegelijkertijd** uitgevoerd worden, zullen de **2de maal nog steeds zijn oude waarde gebruiken** in plaats van zijn nieuwe die volgt uit de eerste bewerking.



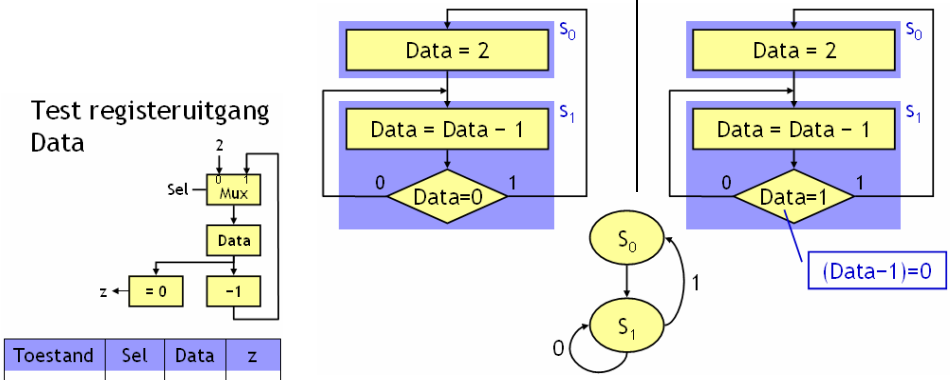
Dit kan vreemd lijken omdat we in de oude stroomschema's telkens de nieuwe waarde gebruikten, maar hier worden **alle bewerkingen tegelijk uitgevoerd** en dus zal **v zijn oude waarde nog hebben** bij het begin van de operaties. Wanneer we de nieuwe waarde willen gebruiken moeten we dat specifiek aangeven (zie vb).



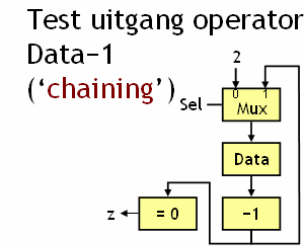
Een **voorbeeld** : We willen de sequentie 2 1 0 2 1 0 2 … op de variabele data krijgen. We zullen hiervoor aanvankelijk Data 2 stellen en dan telkens 1 aftrekken tot we 0 hebben en dan terug 2 in Data steken. Dit zijn 2 toestanden en dus 2 ASM-blokken :

Links is <u>FOUT</u> : in S1 wordt de controle op Data uitgevoerd op de waarde van data zoals die de totale toestandskader binnenkomt. Het systeem zal hier pas reset worden wanneer Data=0 binnenkomt, en op die moment zullen we ook 1 aftrekken en dus -1 op Data krijgen. Dit is fout !

Rechts is <u>JUIST</u> : Wanneer Data=1 binnenkomt wordt 1 afgetrokken en wordt Data 0. Op dat ogenblik zullen we ook resetten wordt dus bij de volgende klok het systeem gereset op 2. We krijgen dan de sequentie die we willen, dus zonder dat Data op een bepaald ogenblik -1 wordt.



Toestand	Sel	Data	z
S_1	1	1	0
S_1	1	0	1
S_0	0	-1	0
S_1	1	2	0
S_1	1	1	0
S_1	1	0	1



Toestand	Sel	Data	z
S_1	1	1	1
S_0	0	0	0
S_1	1	2	0
S_1	1	1	1
S_0	0	0	0

We kunnen dit echter wel **oplossen** door de **hardware-implementatie te veranderen**. Voor de 2de bewerking takken we dan Data **niet aan het register** maar **na de eerste bewerking** af. Dit heet **chaining**. Het duurt wel een **klokflank langer** ! (zie implementaties boven)

Synthese naar hardware

1) Basisprincipes

Vertrekkend van een **toestand-actie-tabel** of een **ASM-schema**, kunnen we zowiezo een **FSMD realiseren** als volgt:

- ⇨ elke **variabele** komt overeen met een **register**
- ⇨ elke **operatie** komt overeen met een **FU** (funtional unit)
- ⇨ het **gebruik** (lezen) van een **variabele** is een **verbinding** van een **register** naar een **FU**
- ⇨ het **wijzigen** (schrijven) van een **variabele** is een **verbinding** van een **FU** naar een **register**
- ⇨ elke **ASM-blok** en dus elke rij van de toestand-actie-tabel komt met een **toestand** van de **controller** overeen.

Dit leidt **niet tot een minimale realisatie** ! We moeten dus de schakeling die we zo bekomen gaan **optimaliseren**. bv. elke '+' resulteert in een aparte opteller, hetgeen niet echt efficiënt is wanneer we die niet allemaal tegelijk nodig hebben.

Deze **optimalisatie** (of minimalisatie) moeten we doen voor de **controller** en het **datapad** :

⇨ Voor de **controller** komt dit neer op het **minimalisering FSM-ontwerp** (zie vroeger)

⇨ Voor het **datapad** zullen we proberen zo **weinig mogelijk en zo klein mogelijke onderdelen** te gebruiken. Dit impliceert dat we **zoveel mogelijk compenten herbruiken** (voor verschillende operaties in verschillende toestanden) : **Sharing** :

- ⇨ ***Register sharing***: **variabelen die niet tegelijkertijd** gebruikt worden kunnen samen een **register** gebruiken. Het hangt van de toestand af welke variabele het register voorstelt, en de andere variabele is op dat ogenblik niet gebruikt.
- ⇨ ***Functional unit sharing***: **eenzelfde FU** voert **meerdere operaties** uit die niet op hetzelfde ogenblik gebeuren. Wanneer een aantal **verschillende toestanden** elke **één optelling** moeten uitvoeren, hebben we **genoeg aan één teller** die gebruikt kan worden in elke toestand.
- ⇨ ***Connection sharing***: **eenzelfde draad** wordt voor **meerdere verbindingen** gebruikt die **niet op hetzelfde ogenblik** informatie vervoeren. We sluiten een aantal componenten dan met 3-state-buffers op de draad aan en afhankelijk van de toestand hebben een aantal componenten toegang tot de draad en de anderen niet. (hebben op dat ogenblik geen verbinding nodig)
- ⇨ ***Register port sharing***: een **registerbank** verzamelt meerdere **registers** waarvan de poorten **niet tegelijkertijd gebruikt** worden. Wanneer we een **aantal registers** niet tegelijkertijd nodig hebben (we zullen er maar op elk ogenblik max 1 gebruiken) kunnen we deze verzamelen in een **registerbank, hetgeen kleiner en dus is goedkoper**.

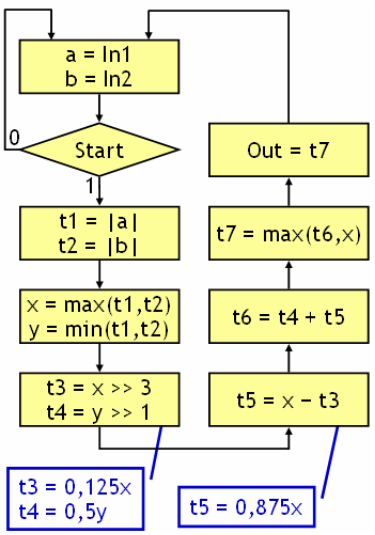
Als **voorbeeld** van **minimalisatie en optimalisatie** bespreken we de **implementatie** van een eenvoudig algoritme om de **wortel van een kwadratische som SRA** te berekenen. (square root aproximation: gebruikt om bv het vermogen van een transmissiesignaal te berekenen) :

$$\sqrt{a^2+b^2} \approx \max\left((0{,}875x+0{,}5y),x\right)$$

$$\text{met}\quad x=\max\left(|a|,|b|\right)\ \text{ en }\ y=\min\left(|a|,|b|\right)$$

Het **bijbehorende ASM-schema** voor de SRA : Wanneer start 1 wordt zullen we de ingangen inlezen, absolute waarde nemen, minimum en maximum nemen, x 3 keer delen door 2 en y 1 keer delen door 2, x min vorige resultaat van x om 0.875x te bekomen, de bewerkte versie van x en y optellen, het maximum nemen en dit op de uitgang zetten.

	S1	S2	S3	S4	S5	S6	S7
a	X						
b	X						
t1		X					
t2		X					
x			X	X	X	X	
y			X				
t3				X			
t4				X	X		
t5					X		
t6						X	
t7							X
#	2	2	2	3	3	2	1



⇨ **Variabelen** : De **levensduur** van een variabele is de **tijd dat ze bestaat** (**nodig is) in de schakeling**. Dit is dus een **aantal toestanden** beginnend met die waar ze **gedefinieerd** wordt en eindigend waar ze de **laatste keer gebruikt** wordt. Wat nu belangrijk is is de **combinatie** van de **levensduur** van **alle nodige variabelen**, en meer bepaald het **maximum aantal** variabelen die **tegelijkertijd levend** zijn (het max aantal variabelen in een toestand). We zien dat in vorige schakeling **in één toestand maximaal 3 variabelen** voorkomen, en dus hebben we slechts **3 registers** nodig voor heel de schakeling.

⇨ **FU's** : We zien **hetzelfde** bij de **FU's**. We moeten **nooit meer dan 2 operaties** tegelijkertijd uitvoeren in **één toestand** dus hebben we **slechts 2 FU's nodig**. Of we kunnen **1 FU nemen die 2 bewerkingen ineens** kan doen (let wel : hoe meer bewerkingen ineens hoe complexer de FU)

	S1	S2	S3	S4	S5	S6	S7	#
abs	2							2
min		1						1
max		1				1		2
>>			2					2
−				1				1
+					1			1
#	2	2	2	1	1	1		9

	a	b	t1	t2	x	y	t3	t4	t5	t6	t7
abs1	I		O								
abs2		I		O							
min			I	I		O					
max1			I	I	O						
max2					I					I	O
>>3					I		O				
>>1						I		O			
−					I		I		O		
+									I	I	O

⇨ **Verbindingen** : We zien ook iets **analoogs** voor de **verbindingen**. We tellen hier weer **per toestand** de **nodige ingangen en uitgangen** op voor **alle bewerkingen** van die toestand. We zien dat we in één toestand **nooit meer dan 4 ingangen** en **2 uitgangen tegelijkertijd** nodig in één toestand. We voorzien dus een **ingangsbus** met 4 lijnen en een **uitgangsbus** met 2 lijnen.

⇨ **verdere Optimaliseringen** in de **FU's**:

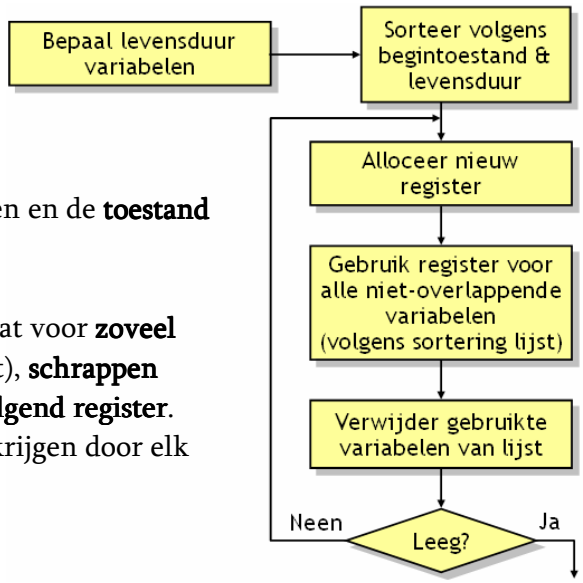
⇨ **Bewerkingen** die in **parallel** ***kunnen*** **gebeuren** ***mogen*** **ook sequentieel uitgevoerd** worden, als de **specificaties** dit toelaten. Hiermee **verkleinen** we het **aantal nodige FU's** (meer herbruik).

⇨ **Herschrijf** het **algoritme** om een optimalere implemetatie te kunnen maken, maar we moeten wel de **specificaties behouden** ! voorbeelden van algoritme-aanpassingen :

- ***Chaining***: zoveel mogelijk **bewerkingen na elkaar** uitvoeren **binnen 1 klokcyclus**
- ***Multicycling***: we laten een FU toe om **meer dan 1 klokcyclus te gebruiken voor een berekening** waardoor ingewikkeldere berekeningen gemaakt kunnen worden
- ***Pipelining***: **splits** een **bewerking** op in **verschillende onderdelen**, die dan **ieder 1 klokcyclus** gebruiken. Zo hoeven we niet te multicyclingen

De **verwerkingskracht** van de schakeling **verhoogt** uiteraard met een **hogere klokfrequentie** en wanneer we **meer bewerkingen** kunnen doen **in 1 klokcyclus**.

2) Register sharing

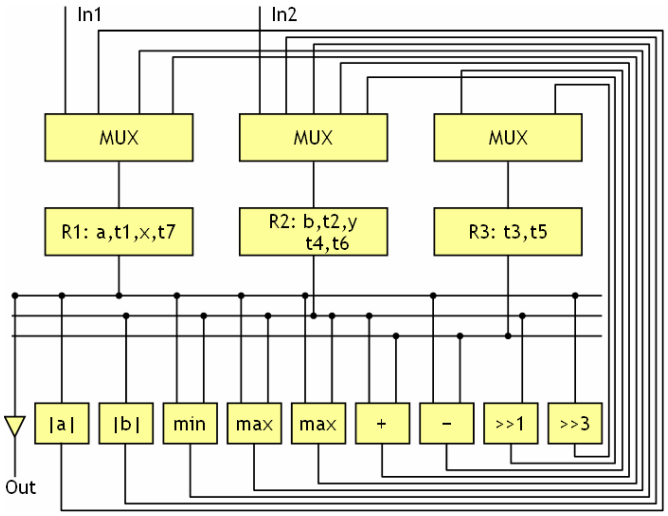


Linkerrand-algoritme

⇨ We bepalen de **levensduur** van alle nodige variabelen en de **toestand** waarin ze **aangemaakt** worden
⇨ We **sorteren** de lijst op vorige dingen
⇨ We **creëren** nu telkens een **register**, we gebruiken dat voor **zoveel mogelijk niet-overlappende variabelen** (volgens de lijst), **schrappen** alle voorziene variabelen van de lijst en nemen een **volgend register**.
De bedoeling is dat we zo **zo weinig moglijk registers** krijgen door elk register **zoveel mogelijk te gebruiken**.

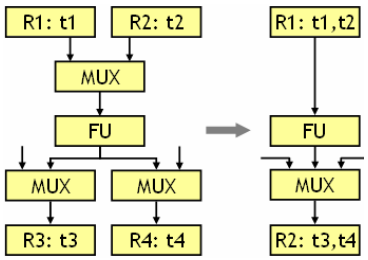
	S1	S2	S3	S4	S5	S6	S7
a	X						
b	X						
t1		X					
t2		X					
x			X	X	X	X	
y			X				
t4				X	X		
t3				X			
t5					X		
t6						X	
t7							X

R1 : a t1 x t7
R2 : b t2 y t4 t6
R3 : t3 t5



Dit geeft volgende **implementatie** :

⇨ Deze **implementatie** kan echter **nog beter** : naast het **kleinst aantal registers** zullen we ook zoeken naar het **kleinste aantal MUX-poorten**. Er zijn namelijk voor het minimale aantal registers nog **verschillende implementaties** mogelijk, en daaruit zullen we nu de **eenvoudigste halen** (dus die met de minste MUXen). We zullen daarvoor variabelen die dezelfde ingang of dezelfde uitgang van een FU gebruiken samenvoegen in hetzelfde register.



⇨ **Impact** van het **samenvoegen** van **FU's** op de minimalisatie van het aantal MUX ? :
probleem is dat de MUXen slechts gekend zijn na samenvoegen FU's omdat we moeten weten welke FU's we hebben voor we ze kunnen verbinden met de variabelen! Dan eerst FU sharing? Neen, want dan hangt de keuze van de MUXen in die stap af van hoe de variabelen over de registers verdeeld zijn!
Deze **patstelling** ('deadlock') is **typisch een probleem van gekoppelde optimaliseringen**. **Oplossing**:
⇨ Optimaliseer **eerst** wat het **meest oplevert** en maakt hierbij een **schatting** van de **andere** optimaliseringen
⇨Optimaliseer dan de **andere aspecten** en **itereer tot een bevredigend resultaat** bereikt is

⇨ Wat brengt het meest op: het **samenvoegen van registers of van bewerkingen**?

⇨ **Meestal** het samenvoegen van **registers**:

– Er zijn **meer variabelen dan FU's**

– Het samenvoegen van **FU's** resulteert meestal in een **duurdere FU** dan ieder van de FU's afzonderlijk, wat **niet het geval is bij registers**

– Het is **gemakkelijker** om te **schatten** welke **bewerkingen** kunnen **samengenomen** worden dan welke registers

⇨ **Soms** het samenvoegen van **FU's**

– Als **slechts één soort FU** gebruikt/aanwezig is

– Als de **kostprijs** van een **register verwaarloosbaar is t.o.v. de kostprijs van een FU**

-> in een FPGA is het register aan de FU-uitgang gratis

Gebruik van een compatibiliteitsgraaf

Om de zoektocht naar het **kleinst aantal registers** te **combineren** met **MUX-reductie**, maken we gebruik van de methode van de **compatibiliteitsgraaf** :

⇨ Stel een **compatibiliteitsgraaf** op

⇨ **Splits** ze met het **max-cut algoritme**

In het **SRA-voorbeeld** maken we de **volgende schatting** van samenvoegen FU's:

- Combineer de **twee max-bewerkingen** in **één**
- Combineer de **optelling** en de **aftrekking** in **één adder-subtractor**

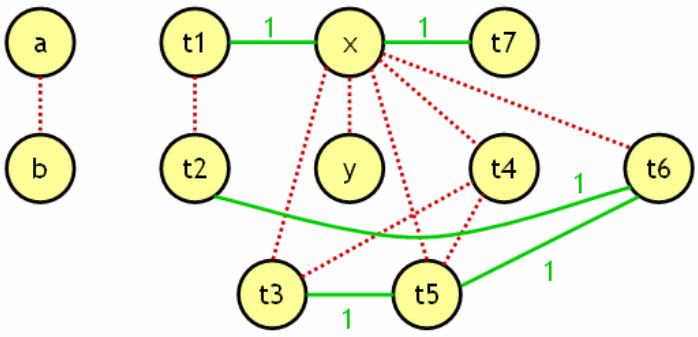
	S1	S2	S3	S4	S5	S6	S7	#
abs	2							2
min		1						1
max		1					1	2
>>			2					2
-				1				1
+					1			1

De **compatibiliteitsgraaf** bestaat uit **drie delen**:

⇨ **Knopen ('nodes')**: dit zijn de variabelen (elke variabele is een knoop)

⇨ **Incompatibiliteitsranden**: tussen knopen die onverenigbaar (niet samen te voegen) zijn omdat ze een overlappende levensduur hebben.

⇨ **Prioriteitsranden**: we zullen vooral knopen samenvoegen die goed verenigbaar zijn. We zullen dus elke compatibele verbinding van 2 knopen een prioriteit gewicht geven (waarbij het gewicht aangeeft hoe goed de knopen verenigbaar zijn). Knopen zijn goed verenigbaar wanneer ze (de variabelen) aan dezelfde ingang of uitgang van dezelfde FU gebruikt worden. Het gewicht is dan het aantal keren dat de twee variabelen aan de vorige voorwaarde voldoen.

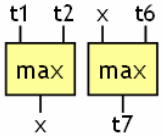


Incompatibiliteitsranden:
variabelen met
overlappende levensduur

	S1	S2	S3	S4	S5	S6	S7
a	X						
b	X						
t1		X					
t2		X					
x			X	X	X	X	
y			X				
t4				X	X		
t3				X			
t5					X		
t6						X	
t7							X

	a	b	t1	t2	x	y	t3	t4	t5	t6	t7
abs1	I		O								
abs2		I		O							
min			I	I		O					
max			I	I	O/I					I	O
>>3					I		O				
>>1						I		O			
-/+					I		I	I	O/I	O	

Prioriteitsranden:
variabelen met
dezelfde FU-ingang of
dezelfde FU-uitgang



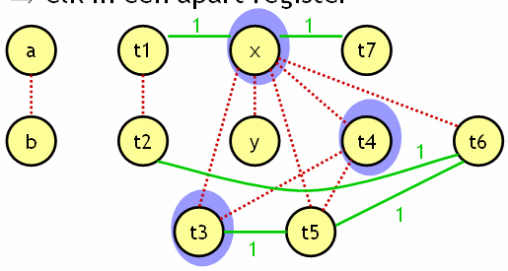
x en t4 zijn incompatibel: geen prioriteitsrand

⇨ Eens de graaf opgesteld is moeten we hem **opsplitsen** : **verdeel** de graaf in het **kleinst aantal groepen** van **verenigbare knopen** zodat het **totale gewicht** van **alle groepen samen maximaal** is.

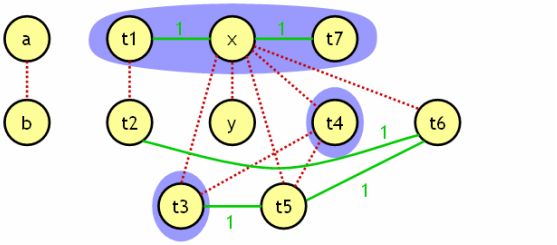
Het **totale gewicht** van een **groep** is de **som** van alle **prioriteitsranden** binnenin de groep.

We zullen dit **visueel** doen (implementaties van algoritmes hiervoor zouden ons te ver leiden)
= max-cut graph partition

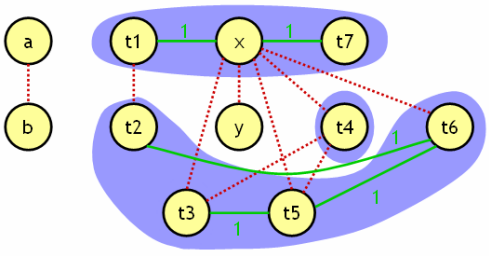
x, t3 en t4 zijn onderling onverenigbaar
⇒ elk in een apart register



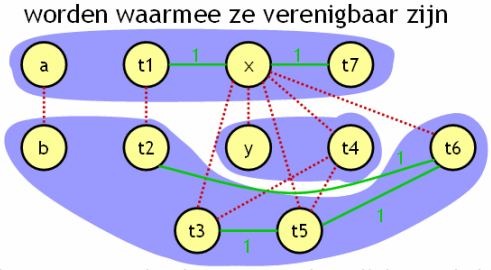
t1 & x en t7 & x zijn verenigbaar en verbonden door een prioriteitsrand met het hoogste gewicht (nl. 1) ⇒ alle drie in hetzelfde register



t3, t5, t6 en t2 zijn verenigbaar en verbonden door prioriteitsranden ⇒ in hetzelfde register



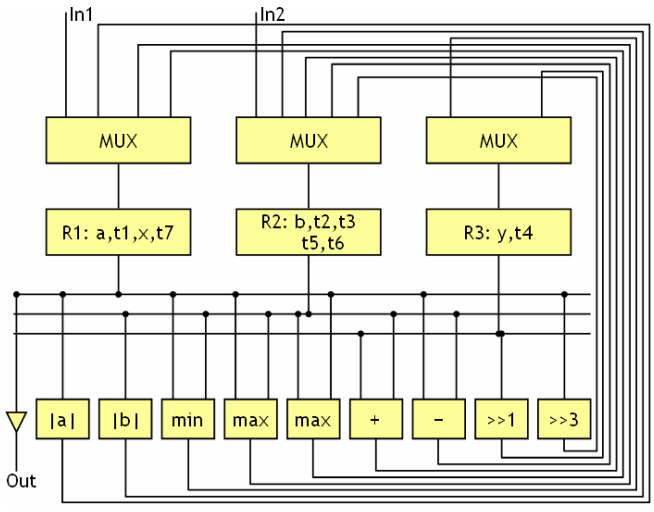
Overblijvende variabelen hebben geen prioriteitsranden
⇒ kunnen bij gelijk welk register gevoegd worden waarmee ze verenigbaar zijn



Resultaat max-cut algoritme:
R1: a, t1, x, t7
R2: b, t2, t3, t5, t6
R3: y, t4

Resultaat linkerrand-algoritme:
R1: a, t1, x, t7
R2: b, t2, y, t4, t6
R3: t3, t5

⇨ **Implementatie** na graaf-analyse :



⇨ **Kostprijsberekening** :

- Kostprijs van een **1-bit register** met **CE** en **asynchrone preset** of **clear** :

  ½ CLB / 8 poorten / 38 tors

- Kostprijs van 1-**bit MUX** : zie tabel

	2-to-1	3-to-1	4-to-1	5-to-1
CLB	0,5	1	1	1,5
poorten	3	4	5	6
tors	12	18	24	30

- In een **FPGA** komt er **na elke MUX** een **gratis register**

Kostprijsberekening voor een **32-bit datapad** :

⇨ **Originele FSMD** (geen optimalisering):

⇨ **11 registers** van 32 bits

- 11 reg × 32 bit/reg × ½ CLB/bit = 176 CLB's
- 11 reg × 32 bit/reg × 8 poort/bit = 2816 poorten
- 11 reg × 32 bit/reg × 38 tor/bit = 13376 tors

⇨ **Na samenvoegen registers**:

⇨ **1 register** van 32 bits met een **4-to-1 MUX**

- 1 CLB/MUXREGbit × 32 bit = 32 CLB's
- (5 poort/MUXbit + 8 poort/REGbit) × 32 bit = 416 poorten
- (24 tor/MUXbit + 38 tor/REGbit) × 32 bit = 1984 tors

⇨ **1 register** van 32 bits met een **5-to-1 MUX**

- (1 CLB/4MUXbit + ½ CLB/2MUXREGbit) × 32 bit = 48 CLB's
- (6 poort/MUXbit + 8 poort/REGbit) × 32 bit = 448 poorten
- (30 tor/MUXbit + 38 tor/REGbit) × 32 bit = 2176 tors

⇨ **1 register** van 32 bits met een **2-to-1 MUX**

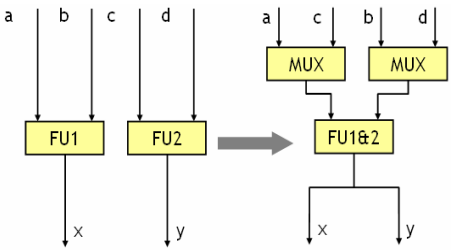
- 1/2 CLB/MUXREGbit × 32 bit = 16 CLB's
- (3 poort/MUXbit + 8 poort/REGbit) × 32 bit = 352 poorten
- (12 tor/MUXbit + 38 tor/REGbit) × 32 bit = 1600 tors

Optimaliseringen beïnvloeden elkaar:

Samenvoegen registers verandert bijvoorbeeld het aantal verbindingen. Telkens we iets optimaliseren veranderen we de configuratie van de rest en het is moeilijk de impact van een bepaalde op de rest van de optimaliseringen te zien.

⇨ We kunnen wel schattingen maken van de reductie in bijvoorbeeld verbindingen bij het samenvoegen van registers.

3) Functional-unit sharing : bewerkingen samenvoegen



⇨ Het **samenvoegen** van **bewerkingen** is het **vervangen van FU's** die **niet tegelijkertijd gebruikt** worden door **een FU** die de **functionaliteit van beide** verwezenlijkt en met een **MUX** aan zijn **ingangen** om te kiezen welke data verwerkt worden op welke wijze.

⇨ we kunnen dit doen voor **identieke FU's** (bijv. 2 × MAX) waar we dan de **functionaliteit niet** moeten **uitbereiden** en er gewoon **één kunnen schrappen**, en we kunnen dit doen voor **gelijkaardige FU's** (bijv. ADD & SUBTRACT) waar we dan met een **kleine uitbereiding van de functionaliteit** van de FU **beide bewerkingen kunnen doen**.

⇨ We zullen dit **enkel doen** wanneer de **nieuwe combinatie** FU & MUX **goedkoper** is dan de **originele oplossing** van 2 FU's!

⇨ Het **samenvoegen** van **FU's beïnvloedt** uiteraard de **totale MUX-kost**

⇨ Wanneer we **2 niet-tegelijkertijd uitgevoerde gelijke bewerkingen** (bv. 2 keer max) willen **verenigen in een één FU** (één keer max), en wanneer **beide bewerkingen data** uit **hetzelfde register** halen, dan hebben we zelfs **helemaal geen MUX nodig** aan de ingangen om de data te kiezen **aangezien die toch telkens uit dezelfde registers** komt. Vandaar het belang om een **goeie keuze** te maken bij **welke variabele** we in **welke registers** steken.

⇨ We kunnen ook voor het **samenvoegen van bewerkingen** het **max-cut algoritme** van de **comatibiliteitsgraaf** toepassen voor de FU's :

⇨ **Knopen** : deze stellen nu de **bewerkingen** voor

⇨ **Incompatibiliteitsranden** : twee bewerkingen die in **eenzelfde toestand gebruikt** worden kunnen niet samen te voegen en zijn dus **incompatibel**

	S1	S2	S3	S4	S5	S6	S7
abs	2						
min		1					
max		1					1
>>			2				
-				1			
+					1		
#	2	2	2	1	1	1	

Knopen zijn bewerkingen

Incompatibiliteitsrand: twee bewerkingen in dezelfde toestand

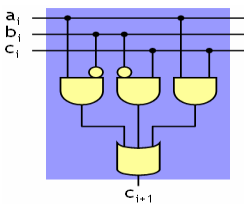
Prioriteitsrand: gewicht = winst door samenvoegen

⇨ **Prioriteitsranden** : twee of meer bewerkingen **die door eenzelfde (nieuwe) FU** kunnen **uitgevoerd** worden zijn **compatibel**. Het **gewicht** van deze compatibele combinatie is gelijk aan de **winst in kostprijs** door het **samenvoegen van de twee**.

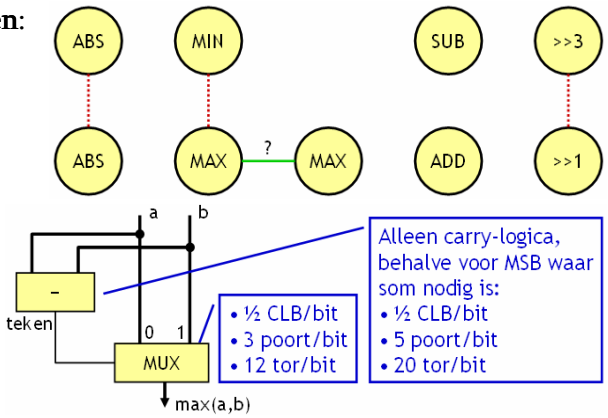
⇨ We zullen nu **elke compatibele combinatie** van **bewerkingen** uit het voorbeeld **uitwerken** en **uitrekenen** welke **besparing** ons dat **oplevert** bij het verenigen in één FU (= het gewicht).

A) Effect van het **samennemen** van de **2 max-bewerkingen**:

⇨ **Kostprijs** van een **MAX-bewerking** op **1 bit** :
1CLB / 8 poorten / 32 transistors

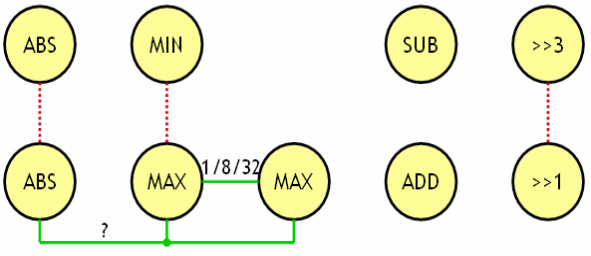


Werking : beide ingangen worden **van elkaar afgetrokken** en we zetten het **teken** om in een bit 0 wanneer positief en 1 wanneer negatief. Een **MUX selecteert** dan **a** als **grootste** wanneer het verschil **positief** is en **b** als **grootste** wanneer het verschil **negatief** is.

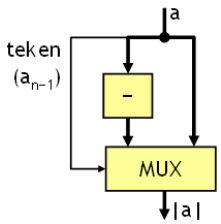


⇨ De 2 max-bewerkingen halen beide hun **data** uit **dezelfde registers**, waardoor een **mux** voor het selecteren van de data **niet nodig** is. We sparen dus hier de exra kost van een max-FU uit, zonder extra kost van een MUX -> **besparing van 1CLB / 8 poorten / 32 transistors**

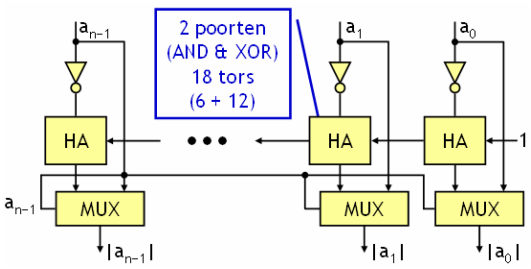
B) We kunnen ook **beide max-operaties** kunnen samennemen met het nemen van de **absolute waarde**.



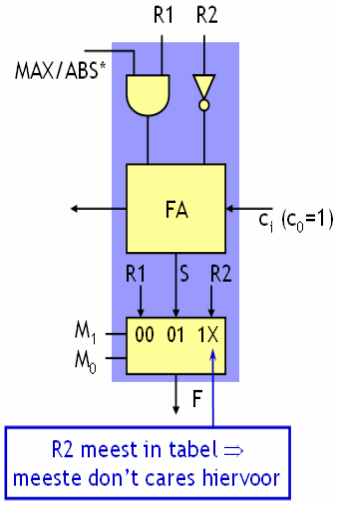
⇨ **Kostprijs** van een **ABS-bewerking** op **1 bit** :
½ CLB (wegens 'carry-chain') / 6 poorten / 32 tors



Werking abs-bewerking : We voorzien zowel de **gewone waarde** van een getal a als de **inverse** ervan, en **selecteren** dan gewoon via een **MUX** op basis van het teken welke de positieve waarde is die de absolute waarde voorstelt. Deze opbouw is **redelijk analoog** aan die van een **max**-bewerking.



⇨ **Realisartie** van de **max- & abs- FU** :

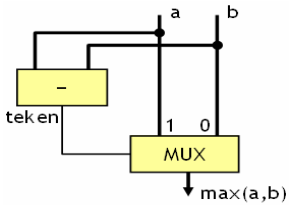


We voeren een **variabele MAX/ABS*** in, die voor **1** de functie **max(R1,R2)** uitvoert en voor **0** de functie **abs(R2)**. **Ingangen R1 en R2** worden dan door de FU verwerkt volgens de gekozen functie tot **1 uitgang F**. In de figuur staat de uitwerking voor 1 bit. Centraal staat de **Full Adder**, die zijn **ingangen opgeteld** presenteert op uitgang S. Om deze om te vormen tot een **aftrekking**, zullen we ingang **R2 inverteren**. De **carry** wordt **doorgegeven** naar de volgende bit-eenheid. De **ingang R1** wordt **gecombineerd** met **MAX/ABS*** zodat die 0 wordt wanneer we de abs-bewerking willen. Uiteindelijk staat er een **MUX** die **selecteert** wat er naar de **uitgang** gaat. (originele ingangen R1 en R2 of de som S van de FA). In de **waarheidstabel** staat wat we wanneer selecteren, in **functie** van de **functieselectie-ingang** en de vorige **R2** en **S**.

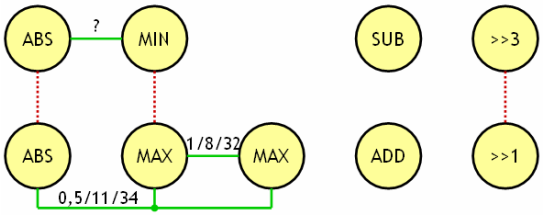
MAX/ABS*	$R2_{n-1}$	S_{n-1}	F	M_{10}
0	0	0	R2	1X
0	0	1	R2	1X
0	1	0	S	01
0	1	1	S	01
1	0	0	R1	00
1	0	1	R2	1X
1	1	0	R1	00
1	1	1	R2	1X

⇨ **Kostprijs per bit** : 1 CLB (FA & INV & AND) + 1 (MUX) = 2 CLB's
5 poorten (FA) + 1 (AND) + 1 (INV) + 4 (MUX) = 11 poorten
36 tors (FA) + 6 (AND) + 2 (INV) + 18 (MUX) = 62 tors
tegenover de prijs van 2 max- en 1 abs-bewerking is dit een **besparing** van **0,5 CLB / 11 poorten / 34 tors**

C) Realisatie van een **abs- en een min-bewerking** samen :



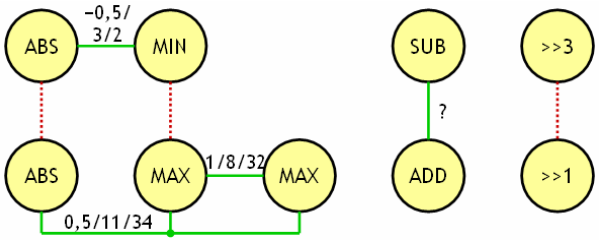
Werking MIN-bewerking : is eigenlijk volledig analoog aan die van een max-bewerking met het enige verschil dat we in de selector de andere waarde selecteren (de kleinste in plaats van de grootste dus). De kostprijs is volledig gelijk.



⇨ **Realisartie** van de **min- & abs- FU** is eveneens **volledig analoog** aan het vorige, evenals de **kostprijs van 2CLB's / 11 poorten / 62 tors**.

⇨ **Kostenbesparing** bij het vervangen van een min- en abs- bewerking door een FU die beide combineert : **-0.5 CLB / 3 poorten / 2 tors**

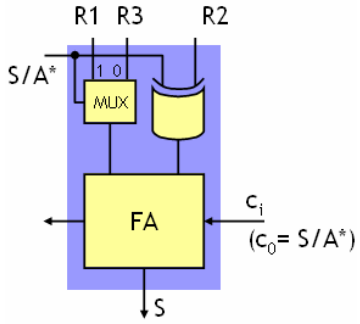
We zien dat het **niet altijd even goed** is deze minimalisering door te voeren : op gebied van **poorten en transistors** is er een **minime winst**, maar op **gebied** van **CLB's** kost deze 'minimalisering' een **halve CLB extra** (appart 1.5 CLB, tesamen 2 CLB). Wanneer we dit realiseren op een **FPGA**, kunnen we dus **beter de 2 apparte functies** gebruiken !



D) Realisatie van een **Adder en een Substractor** :

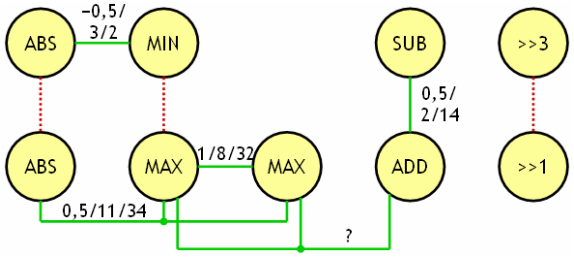
⇨ Prijs **opteller** : ½ CLB / 5 poorten / 36 tors

⇨ Prijs **aftrekker** : ½ CLB / 6 poorten / 38 tors
een aftrekking is gewoon een optelling waarbij het 2de getal is geïnverteerd. Dus gewoon extra kost voor invertor (extra poort ⇨ 2 extra tors)

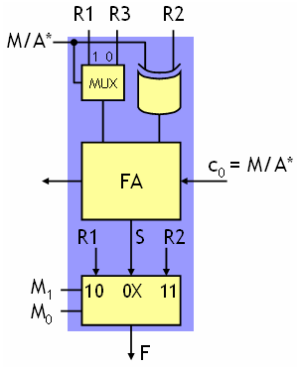


⇨ Prijs **FAS** (full adders substractor) : ½ CLB / 6 poorten / 48 tors
een FAS is een **combinatie** van de **2 vorige**. Aangezien het **verschil** tussen optelling en aftrekking alleen een **invertor** is op de 2de ingang, zorgen we voor een **instelbare invertor** : een **XOR** waarmee we kunnen **kiezen** tussen optellen of aftrekken.

⇨ In het geval van dit **voorbeeld** moeten is de **data** voor de **eerste ingang verschillend** voor de **optelling en de aftrekking**, en zullen we dus nog een extra mux moeten voorzien. De totale kost wordt hier dus ½ CLB (FAS & MUX) /
6 poorten (FAS) + 3 (MUX) = 9 poorten /
48 tors (FAS) + 12 (MUX) = 60 tors
dit betekend een **winst van 0,5 CLB / 2 poorten / 14 tors**



E) Realisatie van een **Adder** en een **Max-bewerking** :



In de **vorige implementaties** van de **Max-bewerking** werd gebruik gemaakt van een **Full Adder** om aan de hand van het **teken** van de bewerking het **maximum** van 2 getallen te selecteren. Daar deze **Full Adder toch al beschikbaar** is, kunnen we hem met een **minimale inspanning** perfect **benutten** om gewone **Add-operatie te verwezelijken**. We voegen dan een **ingang M/A*** toe die voor **0** de **som** van de 2 ingangen realiseert op de uitgang, en voor **1** het **maximum** van de 2 ingangen.

M/A*	S_{n-1}	F	M_{10}
0	0	S	0X
0	1	S	0X
1	0	R1	10
1	1	R2	11

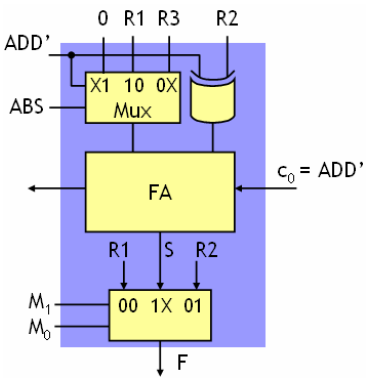
M_1 = MAX/ADD*
M_0 = S_{n-1}

⇨ **Kostprijs** : ½ CLB (FAS & MUX) + 1 (MUX) = 1,5 CLB /
6 poorten (FAS) + 3 (MUX) + 4 (MUX) = 13 poorten /
48 tors (FAS) + 12 (MUX) + 18 (MUX) = 78 tors

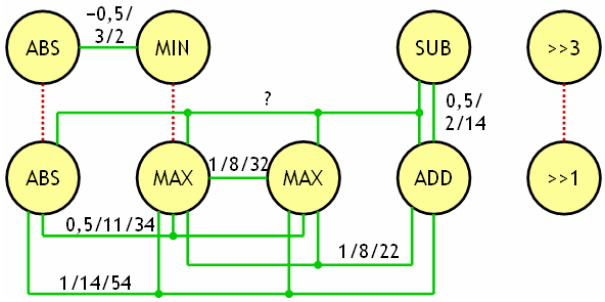
⇨ Wanneer we nu **1 abs-, 2 max- en 1 add-opertie** samenvoegen tot **1 FU**, kunnen we daar een **winst** uithalen van **1 CLB / 14 poorten / 54 tors**

We houden hierbij rekening **uit welke registers de data** voor de functie moet komen en dus welke **MUXen** we moeten gebruiken om telkens de **juiste data** bij de **juiste functie** te brengen.

F) We kunnen dit nog meer uitbereiden tot een FU die **abs-, max-, add- en sub-bewerkingen** uitvoerd :



⇨ Ook hier is de **basis** van al deze **operaties** het optellen met een **Full Adder**. We bereiden uit met een **XOR** om **aftrekkingen** mogelijk te maken en met een **MUX** om de **data** voor de eerst ingang te **kiezen**. We hebben dan een **FU** met **4 verschillende bewerkingen**.

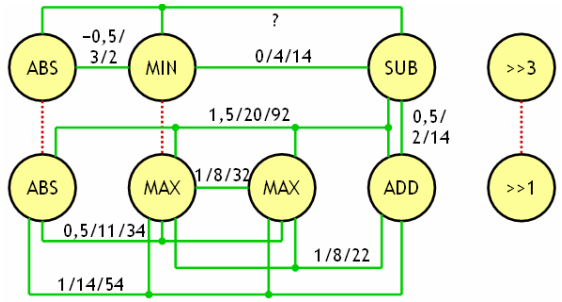


⇨ In ons ontwerp moet **afhankelijk** van **welke functie** we uitvoeren het **resultaat** naar een **ander register** gestuurt worden. Elke functie heeft dus zijn **eigen oplossingsregister** en dus **schrijven** we dat ook zo in de **vergelijkingen** van de bewerkingen.

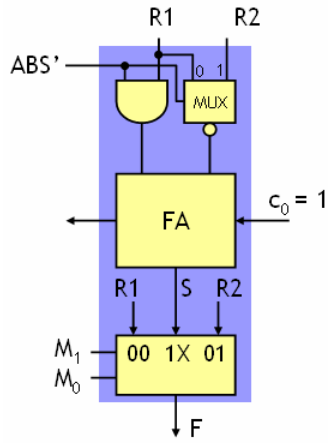
$$\begin{cases} R2 = ABS(R2) \\ R2 = ADD(R3,R2) \\ R2 = SUB(R1,R2) \\ R1 = MAX(R1,R2) \end{cases}$$

⇨ **Kostprijs per bit**: ½ CLB (FAS) + ½ (MUX) + 1 (MUX) = 2 CLB
6 poorten (FAS) + 3 (MUX) + 4 (MUX) = 13 poorten
48 tors (FAS) + 12 (MUX) + 18 (MUX) = 78 tors

H) Volledig analoge resultaten voor het combineren van een **min-, een abs- en een sub- operatie** met een winst van **-0.5 CLB / 7 poorten / 28 tors**.



$$\begin{cases} R1 = ABS(R1) \\ R3 = MIN(R1,R2) \\ R2 = SUB(R1,R2) \end{cases}$$

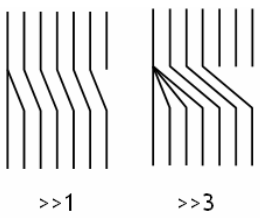


Op een **FPGA** is dit werderom **geen goed idee**, voor een **eigen ontworpen** chip **wel** aangezien we aanzienlijk wat poorten en trs uitsparen. Vooral het combineren van **abs- en sub** gaat goed omdat deze exact **dezelfde hardware-componenten** nodig hebben. Wanneer we echter alleen min- en sub- combineren hebben we geen netto winst of verlies op FPG : 0 CLB / 4 poorten / 14 tors

⇨ **Niet alle combinaties** zijn **onderzocht**, maar we nemen aan dat **die geen extra winst opleveren**

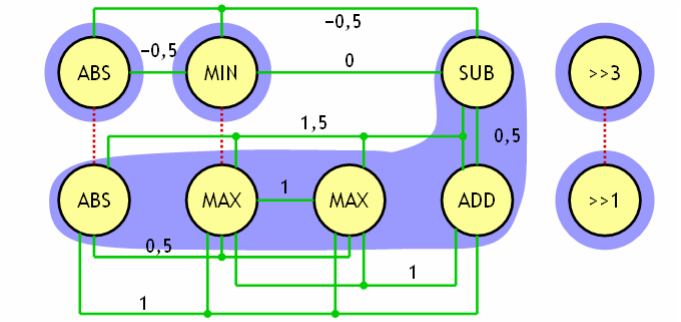
⇨ Het **max-cut algoritme** is **minder geschikt** als de winst van de combinatie van 3 knopen verschilt van de som van de winsten van 2 knopen!

⇨ Nu blijven enkel de **shift-operaties** nog over om te **combineren** met de anderen. Dit is echter **niet zinvol** omdat shift-operaties **gratis** zijn (ze kosten niets omdat er geen poorten of tors voor nodig zijn).

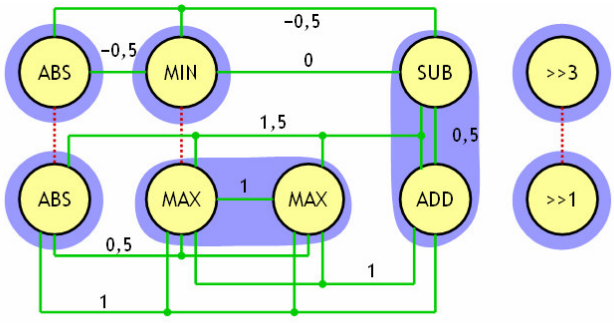


De **ideale combinatie** van **FU's** hangt nu af van de het **niveau** waarop we de chip **ontwerpen**:

⇨ **FPGA** : **minimaal aantal CLB's**. Hier zijn 2 mogelijkheden die elke evenveel CLB's nodig hebben. We zullen hier gaan voor het kleinste aantal FU's omdat dan de hoeveelheid bedrading ook minimaal is (minder ingewikkeld en dus minder duur)



Mogelijkheid 1: (ABS), (MIN), (ABS & MAX & MAX & ADD & SUB), (>>3), (>>1): winst 1,5 CLB's, kost 3,5 CLB's

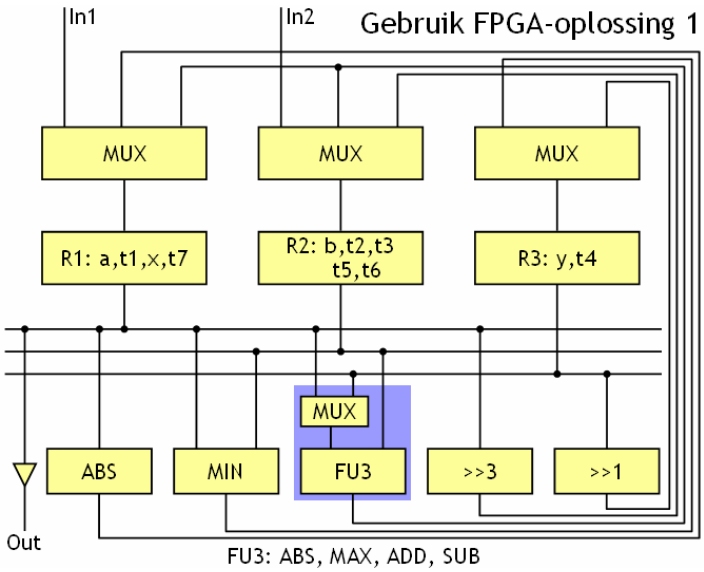


Mogelijkheid 2: (ABS), (MIN), (SUB & ADD), (ABS), (MAX & MAX), (>>3), (>>1): kost 3,5 CLB's, winst 1,5 CLB's

⇨ **Gate-arrays : minimaal aantal poorten**. Mogelijkheid 1: (ABS & MIN), (ABS & MAX & MAX & ADD & SUB), (>>3), (>>1): kost 24 poorten, winst 23 poorten

⇨ **CMOS : minimaal aantal tor's**. Mogelijkheid 1: (ABS & MIN), (ABS & MAX & MAX & ADD & SUB), (>>3), (>>1): kost 140 tors, winst 94 tors

⇨ **Implementatie** met gebruik van oplossing 1 :



⇨ Na het **samenvoegen** van de **bewerkingen** hebben we in de **FU's** (eigenlijk enkel in FU3) een **winst** van **1,5 CLB / 20 poorten / 90 tors**.

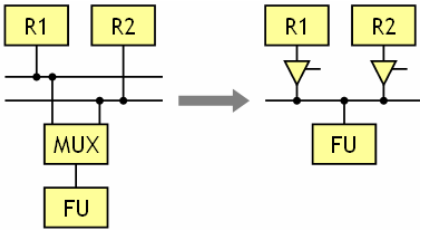
⇨ De **nieuwe kostprijs** van de **registers** en de **MUXen** :

- 2 registers met 3-to-1 MUX : 1 CLB / 8 + 4 poorten / 38 + 18 tors) × 2
- 1 register met 2-to-1 MUX : 0,5 CLB / 8 + 3 poorten / 38 + 12 tors

⇨ De **optimaliseringen beïnvloeden** elkaar: het **samenvoegen** van **FU's** verandert het **aantal registers** en het **aantal verbindingen**

Samen-voegen	CLB			poorten			transistoren			Verbin-dingen
	Reg	FU	Tot	Reg	FU	Tot	Reg	FU	Tot	
Origineel	176	160	336	2816	1504	4320	13376	7488	20864	20
Register	96	160	256	1216	1504	2720	5760	7488	13248	12
FU	80	112	192	1120	832	1952	5184	4544	9728	8
Bus										
Reg.bank										

4) Bus sharing : verbindingen samenvoegen



Bus sharing is het **samenbrengen** van **verbindingen** : we **vervangen twee of meer verbindingen** die **niet tegelijkertijd** gebruikt worden door **één verbinding** die **beurtelings gebruikt** wordt om de vervangen verbindingen te verzorgen.

- ⇨ Dit **vermindert** de **bedrading**, wat tegenwoordig een **belangrijke kost** vertegenwoordigt
- ⇨ We moeten echter wel de kost van **extra 3-state buffers** rekenen die nodig zijn om de verschillende bronnen aan te sluiten of a te sluiten van de bus.
- ⇨ bus sharing **vermindert ook het aantal MUXen** wanneer **één bus** verschillende verbindingen verenigt die **dezelfde FU-ingang aansturen**. Dit verlaagt de kost ook aanzienlijk

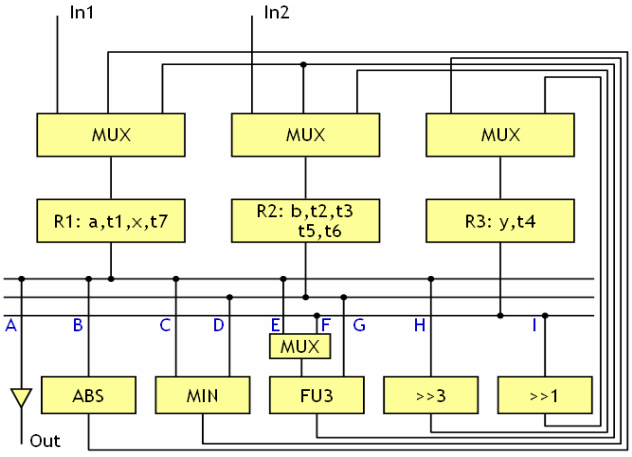
Ook voor het **samenvoegen** van **verbindingen** kunnen we een **Compatibiliteitsgraaf** opstellen :

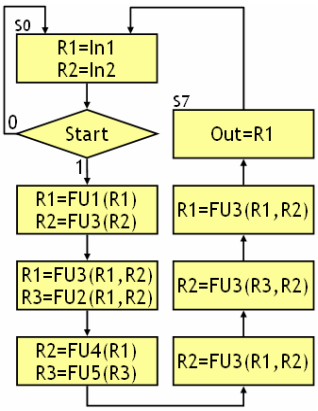
- ⇨ **Knopen** komen overeen met **verbindingen**
- ⇨ **Incompatibiliteitsranden** zijn de randen tussen **onverenigbare verbindingen** : wanneer twee verbindingen in **eenzelfde toestand** gebruikt worden en **verschillende aansturingen** hebben zijn ze **onverenigbaar**
- ⇨ **Prioriteitsranden** tussen twee verbindingen die **éénzelfde aansturing** hebben (besparing # 3-state buffers) of **eenzelfde gebruiker** (besparing # MUX's). Hoe meer bronnen met dezelfde aansturing of gebruiker op éénzelfde bus zitten, hoe beter omdat we dan besparen op buffers of MUXen en dus hoe groter get gewicht van de combinatie.

⇨ Het **samenvoegen** van **verbindingen** moet **tweemaal** gebeuren (voor de twee groepen bussen) : de groep **operanden** voor de verbinding **van registers naar FU's**, en de groep **resultaten** die verbindingen vormen **van de FU's terug naar de registers**. Beide groepen moeten appart behandeld worden en kunnen best niet onderling gemengt worden.

⇨ We zullen het **samenvoegen** van **verbindingen** via de **compatibiliteitsgraaf** tonen voor het **voorbeeld** van de SRA (square root adding) :

We zullen nu **eerst alle verbindingen tussen registers en FU's** proberen samen te nemen, **daarna** doen we **hezelfde** in de **tegengestelde richting**, namelijk van **FU's terug naar registers**. Elke verbindingen van een **register naar een FU** geven we een **letter** en hiermee stellen we de **verbinding** voor in de **graaf**.

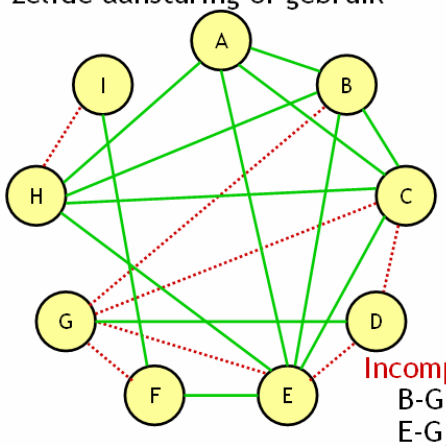




De **onverenigbare** verbindingen zijn die die in **dezelfde toestand** gebruikt worden.

	S0	S1	S2	S3	S4	S5	S6	S7
A : R1→Out								X
B : R1→FU1		X						
C : R1→FU2.1			X					
D : R2→FU2.2			X					
E : R1→FU3.1			X		X		X	
F : R3→FU3.1						X		
G : R2→FU3.2		X	X		X	X	X	
H : R1→FU4				X				
I : R3→FU5				X				

Prioriteitsranden:
zelfde aansturing of gebruik

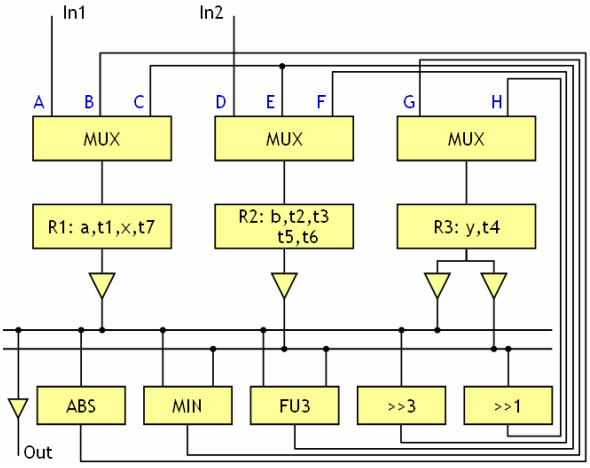


A	R1→Out
B	R1→FU1
C	R1→FU2.1
D	R2→FU2.2
E	R1→FU3.1
F	R3→FU3.1
G	R2→FU3.2
H	R1→FU4
I	R3→FU5

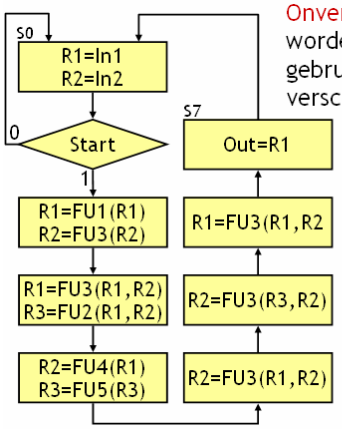
Incompatibiliteitsranden:
B-G, C-D, C-G, D-E,
E-G, H-I, F-G

Oplossing :

- ⇨ **Operandbus** 1 : A, B, C, E, F, H
- ⇨ **Operandbus** 2 : D, G, I



⇨ Voor de **verbindingen** van de **FU's** naar de **registers** doen we het **hele proces nog ee**ns over. De **letters** stellen nu **andere verbindingen** voor als in de vorige graaf !



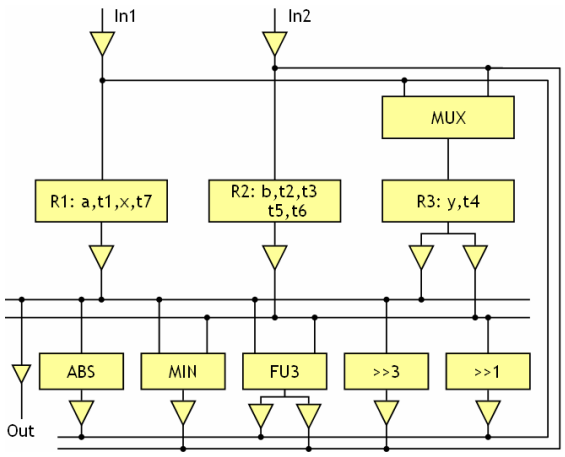
Onverenigbare verbindingen worden in dezelfde toestand gebruikt en komen uit een verschillende FU: A-D B-E C-G F-H

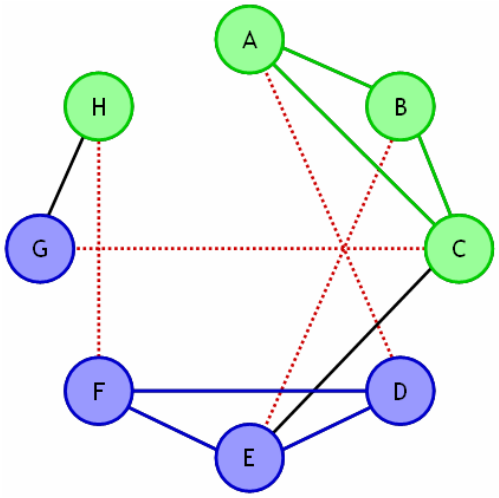
	S0	S1	S2	S3	S4	S5	S6	S7
A : In1→R1	X							
B : FU1→R1		X						
C : FU3→R1			X				X	
D : In2→R2	X							
E : FU3→R2		X			X	X		
F : FU4→R2				X				
G : FU2→R3			X					
H : FU5→R3				X				

Oplossing :

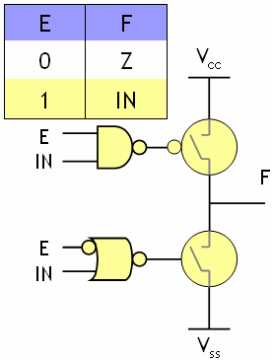
- ⇨ **Resultaatbus 1** : A, B, C, H
- ⇨ **Resultaatbus 2** : D, E, F, G

Uiteindelijke **implementatie** :





De **kostprijs** van een **3-state-buffer bepaald** mede de **winst** voor het **samenvoegen** van **bussen**. Hoe **minder** een **3-state-buffer kost,** hoe **groter** het **voordeel** bij het samenvoegen van bussen.



⇨ **FPGA** : in een FPGA heeft **elke CLB heeft een gratis 3-state buffer** aan een horizontale lange lijn. Er zijn echter een **beperkt aantal lange lijnen** en dus zal het hier extra veel aandacht besteed worden aan zo weinig mogelijk verbindingen

⇨ **Gate array** & **CMOS** : hier kost een 3-state-buffer **3 poorten / 10 tors**. De **implementatie** is in de figuur beschreven : een ingang E (enable) die al dan niet impedant schakelt en een ingang IN die 1 of 0 geeft. De uitgang F is dus afhankelijk van E impedant of gelijk aan de ingang.

Wanneer we de **winst uitrekenen** bekomen we het volgende :

⇨ Qua transport **FU's naar registers** :
- **winnen** we 2 × de verwijdering van 3-to-1 MUX voor registers (= 1 CLB / 8 poorten / 36 tors)
- **betalen** we 6 × een extra 3-state buffers (= 0 CLB / 18 poorten / 60 tors)

⇨ Qua transport **registers naar FU's** :
- **betalen** we 6 × 3-state buffers (= 0 CLB / 18 poorten / 60 tors)
- **winnen** we de verwijdering 2-to-1 MUX uit FU3 (= 0 CLB / 2 poorten / 6 tors)

Samen-voegen	CLB			poorten			transistoren			Verbin-dingen
	Reg	FU	Tot	Reg	FU	Tot	Reg	FU	Tot	
Origineel	176	160	336	2816	1504	4320	13376	7488	20864	20
Register	96	160	256	1216	1504	2720	5760	7488	13248	12
FU	80	112	192	1120	832	1952	5184	4544	9728	8
Bus	48	112	160	1440	1344	2784	5952	6272	12224	4
Reg.bank										

5) Register port sharing : registers samenvoegen in registerbank

We **voegen** hier **registers samen** tot een **registerbank** om

⇨ het **aantal leespoorten** te **verminderen** en dus minder **ingangs-MUX's** te moeten betalen

⇨ het **aantal schrijfpoorten** te **verminderen** en dus minder **3-state buffers** te moeten betalen

⇨ We doen dit volgens de volgende **methodologie** : Met een **Registertoegangstabel** ('Register Access Table') : deze **bevat alle lees- en schrijfoperaties** van de **registers** voor **elke toestand**. We **optimaliseren** dit samenvoegen van registers via de **traditionele methoden** (bijv. **max-cut** en de compatibiliteitsgraaf) of **gewoon exhaustief** voor **eenvoudige gevallen** (zoals voor SRA-voorbeeld).

⇨ Voor het **voorbeeld** van de **SRA** vertrekken we van de **tabellen** met **operandverbindingen** en **resutaatverbindingen**. Vandaaruit stellen we een **tabel** op met daarin de **acties** die moeten gebeuren op de **verschillende registers** in elke **toestand.**

	S0	S1	S2	S3	S4	S5	S6	S7
A : In1→R1	X							
B : FU1→R1		X						
C : FU3→R1			X				X	
D : In2→R2	X							
E : FU3→R2		X			X	X		
F : FU4→R2				X				
G : FU2→R3			X					
H : FU5→R3				X				

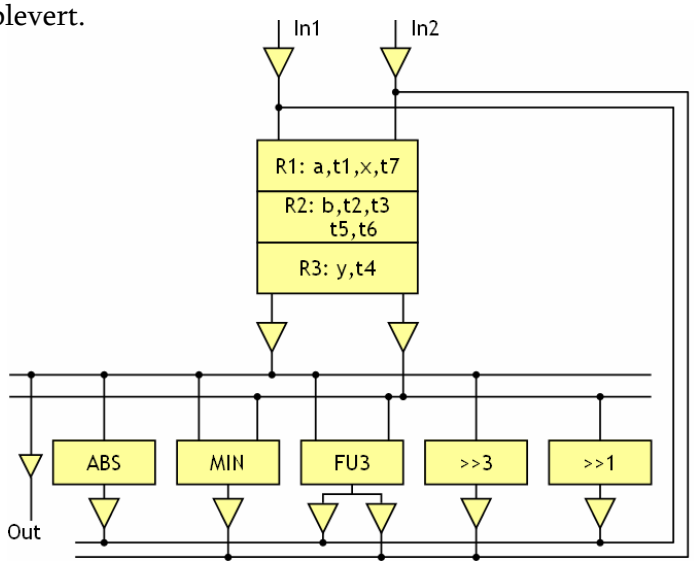
	S0	S1	S2	S3	S4	S5	S6	S7
A : R1→Out								X
B : R1→FU1		X						
C : R1→FU2.1			X					
D : R2→FU2.2			X					
E : R1→FU3.1			X		X		X	
F : R3→FU3.1						X		
G : R2→FU3.2		X	X		X	X	X	
H : R1→FU4				X				
I : R3→FU5				X				

	S0	S1	S2	S3	S4	S5	S6	S7
R1	W	R W	R W	R	R		R W	R
R2	W	R W	R	W	R W	R W	R	
R3			W	R W		R		

⇨ Voor **exhaustieve minimalisering** van het **aantal lees-en schrijfpoorten** van de registers zullen we gewoon **alle opties overlopen**. We gaan alle mogelijke combinatie van registers in banken af en rekenen **telkens uit hoeveel lees-en schrijfpoorten** we **nodig** hebben en **hoeveel poorten dat telkens kost**. We merken dat één bank waar ze alledrie inzitten en die 2 poorten heeft een minimaal aantal poorten oplevert.

registerbanken	# lees/schrijfpoorten	# poorten
(R1), (R2), (R3)	(1;1) + (1;1) + (1;1)	6
(R1,R2), (R3)	(2;2) + (1;1)	6
(R1,R3), (R2)	(2;2) + (1;1)	6
(R1), (R2,R3)	(1;1) + (2;2)	6
(R1,R2,R3)	(2;2)	4

⇨ De **implementatie** wordt dan :



⇨ We zullen weer de **kostenbesparing** in vullen in onze **tabel**. We zien dat dit dit **invoeren** van een registerbank **invloed** heeft op onze **vorige minimalisaties**. We zullen op **gebied** van andere **minimalisaties** terug meer **kost** hebben om deze **reductie** te kunnen **waarmaken**.

Samen-voegen	CLB			poorten			transistoren			Verbin-dingen
	Reg	FU	Tot	Reg	FU	Tot	Reg	FU	Tot	
Origineel	176	160	336	2816	1504	4320	13376	7488	20864	20
Register	96	160	256	1216	1504	2720	5760	7488	13248	12
FU	80	112	192	1120	832	1952	5184	4544	9728	8
Bus	48	112	160	1440	1344	2784	5952	6272	12224	4
Reg.bank	48	112	160	1152	1344	2496	4928	6272	11200	4

Wijzigingen t.g.v. gebruik registerbank
- Ervoor: 1 × 2-to-1 MUX en 6 × 3-state buffers
- Erna: 4 × 3-state buffers

Winst
- 0 CLB's (MUX zat in zelfde CLB als register)
- 9 poort/MUXbit × 32 bit = 288 poorten
- 32 tor/MUXbit × 32 bit = 1024 tors

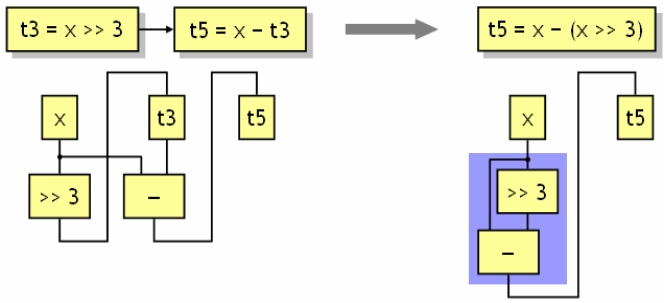
6) Chaining : meerdere bewerkingen in 1 klokcyclus

Chaining is het **uitvoeren** van **meerdere bewerkingen** in 1 **klokcyclus** waarbij het **tussenresultaat** niet in een register wordt **opgeslagen**. We kunnen dit doen wanneer :

- ⇨ De **levensduur** van het **tussenresultaat 1 is**, dus als het tussenresulataat niet buiten die toestand gebruikt moet worden
- ⇨ De **vertraging** van **FU 1** + de **vertraging** van **FU 2** moet **samen kleiner** zijn dan de **klokperiode**, anders passen de 2 bewerkingen niet samen in de klokperiode
- ⇨ De **specificaties** van het **tijdsgedrag** moeten in **orde** zijn

De **voordelen** van het **chainen** van bewerkingen is :

- ⇨ **minder toestanden**
- ⇨ **eventueel minder registers**
- ⇨ **minder klokcycli** voor het hele algoritme (dus het geheel gaat sneller)

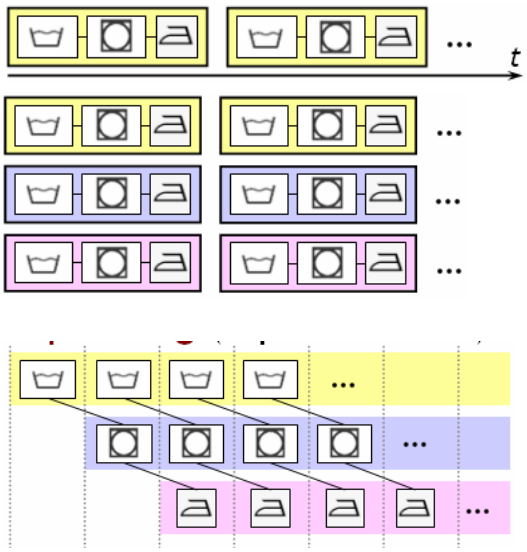


7) Pipelining & multicycling : meerdere klokcycli voor 1 bewerking

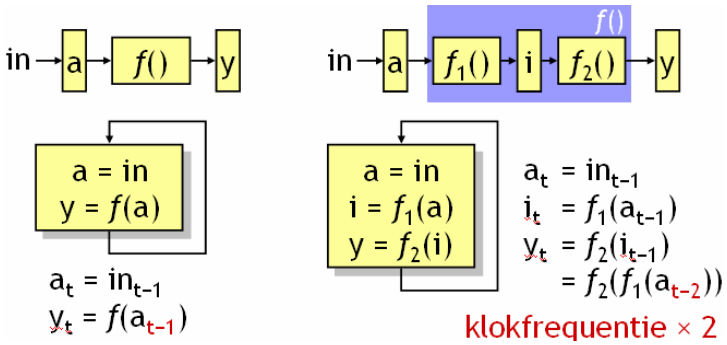
Multicylcing

Bij **piplining** en **multicycling** geven we een **FU meerdere klokcycli** de tijd om een **bewerking** af te ronden. We doen dit wanneer er **meer dan 1 klokcyclus voorbijgaat** tussen de **generatie** en het **gebruik** van het **resultaat** van een bewerking. De **voordelen** zijn **goedkopere** FU's (want trager) of **hogere klokfrequentie** mogelijk. (wanneer de FU meer tijd krijgt voor zijn bewerking kunnen we sneller overschakelen tussen toestanden of kunnen we hem minder nauwkeurig maken kwa tijdsreactie). De **Levensduur** van een **variabele start** als de **bewerking** begint, en er zijn eventueel **extra toestand(**en) nodig. Het idee is dat de schakeling dus gewoon verder zijn ding doet, terwijl die ene FU zijn bewerking afwerkt. Het voordeel is dat de rest van het proces in het datapad niet moet wachten tot de FU klaar is. Voor lange bewerkingen kunnen we zo de FU rustig laten doen (geen extra kost nodig om hem sneller te maken) zonder dat de schakeling daarom trager wordt.

Voorbeeld : een proces dat bestaat uit wassen, drogen en strijken. Wanneer we dit proces sequëntieel uitvoeren en de productie willen opvoeren, dan zouden we dat kunnen doen door verschillende processen parallel uit te voeren. Daar hebben we echter meer hardware voor nodig. Wanneer we het proces chainen kunnen we met 1 keer de hardware 3 keer sneller of 3 keer meer productie uitvoeren. We zetten eigenlijk gewoon elke component heel de tijd aan het werk in plaats van sequentieel af te wisselen

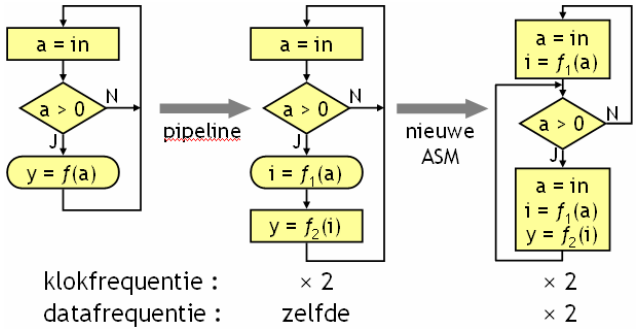


Pipeling

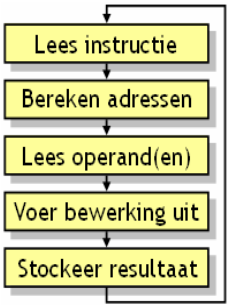


Pipelining is het **opsplitsing** van **bewerking** in ***n* onderdelen**, die **ieder slechts 1/*n*** van de **tijd** gebruiken. We kunnen dan de **klokfrequentie ongeveer *n* keer verdubbelen** en daarmee meestal ≈ de **datafrequentie** ook **n keer groter** maken (n keer grotere performantie). De **berekening** van **1 resultaat** ('latency time') blijft wel ± zelfde. We moeten dan **nog altijd even lang** wachten op een resultaat, maar we **verwerken** wel **n keer meer data** op een **bepaalde tijd** dan normaal.

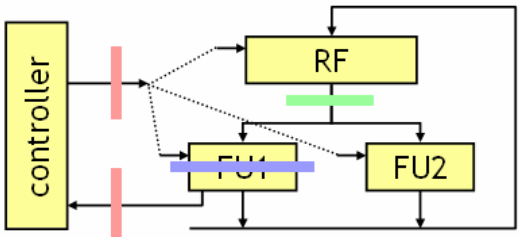
⇨ Normaal hebben we dan ook **n extra toestanden** nodig waar we pipelining doen. We **verliezen daar echter geen performatie mee** omdat de klokfrequentie ook × *n* . We kunnen de **performantie** dikwijls **zelfs verbeteren** door de **ASM** te **herschrijven**.



We zien **verschillende soorten van piplinen afhankelijk** van de **plaats** waar ze voorkomen :

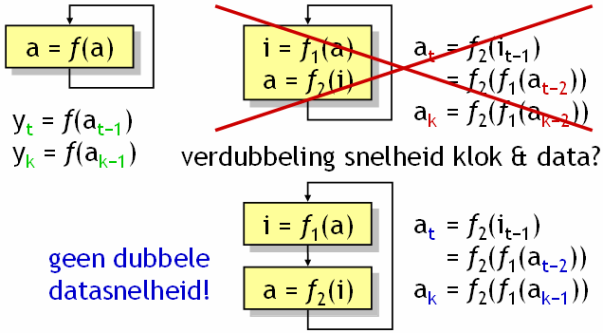


- ⇨ van **FU** (kritisch pad korter maken)
- ⇨ van **datapad** (als trage registerbank)
- ⇨ van **controle** (kritisch pad van processoren meestal door controller)
- ⇨ van **ASM-kaart** (splits ASM)
  - ieder deel krijgt aparte hardware
    ⇒ parallellisme
  - eventueel ontdubbeling hardware



Nadelen van pipelinen :

- ⇨ **Extra hardware** (registers)
- ⇨ **Verlies performantiewinst** bij **terugkoppeling** (bewerking gebruikt resultaat van de vorige)
  ⇒ "stop de pipeline!" (geen nieuwe data)



Besluit Synthese

Het is **zéér moeilijk** om een **optimale implementatie** te vinden! **Eenvoudige optimaliseringstechnieken** (inclusief max-cut) zijn **niet geschikt** voor **sommige minimaliseringen** (bijv. samenvoegen FU's). De **beste keuze** is **sterk afhankelijk** van **technologie**. **Algoritmes herschrijven** geeft veel mogelijkheden, maar dat **ligt niet voor de hand** en dan is er ook wel **herminimalisering** nodig. De **optimaliseringen beïnvloeden** elkaar veel, vandaar dat **globale optimalisering moeilijk is**.

Hfdst 6) Programmeerbare processoren: microprocessoren

Een programmeerbare processor is een FSMD met een programmeerbare controller. De controller bevat nu een geheugen waar we een bepaald algoritme in opslaan en de controller voert het algoritme uit. Een generische controller leest acties uit uit het programmageheugen. We hebben dus voor elke toepassing van de chip een ander programma met een ander algoritme, en een vast design voor de chip zelf. (Elke chip is gelijk, het verschil zit in het programma dat ingeladen is en dat de controller uitvoert op het datapad)

Het programma

1) Instructies

⇨ Het **Programma** is een **opeenvolging** van **instructies** die de **controller moet uitvoeren** op het **datapad**. Deze instructies bevatten **dezelfde informatie** als een **FSM**, en zijn eigenlijk een beschrijving van een controller van een FSMD.

⇨ Elke **instructie** komt overeen een **toestand** van het **FSM** dat de **controller** eigenlijk is.
De instructies - **duiden** aan wat de **volgende instructie** is,
- **zorgen** voor het **aantsturen** van het **datapad** (registers, FU's, MUX's, …)
- **zorgen** voor het **aansturen** van de **data-uitwisseling** (laden/stockeren) met een (**extern) geheugen**

Elke instructie is een **stap** (**actie**) in het programma en **specifiëert** wat er in die stap **moet gebeuren**.
We kunnen **elke actie beschrijven** met verschillende **notaties**:
- **mnemonisch** (zoals in assembleertaal): **Add A, B, C** (≡ a=b+c)
- **als acties** (zoals bij talen voor hardware-specificatie): **Mem[A] ← Mem[B] + Mem[C]**

⇨ **Instructies** zijn in essentie een reeks bits. Om deze bits te gebruiken worden ze **onderverdeeld** in **velden**. **Velden** zijn **groep van bits** waaraan een **betekenis/subtaak** wordt **toegekend**. Een veld kan **allerlei betekenis** hebben voor het programma. De meest voorkomende betekenis :

- een **opcode** ('operation code') is een veld dat aanduid **welke bewerking** moet uitgevoerd worden op de data bv. optelling, …
- een **adres** is een veld dat de **locatie** specifieert waar de **operanden** moeten opgehaald worden & waar de **resultaten** naartoe moeten. Dat kan **zowel** in een **register**(**bank**) als in een **geheugen** zijn, dus in voorlopig geheugen in de processor of vast geheugen om data op te slaan.
  bv. "Load R2, A" (RF[2] ← Mem[A])
- een **instructietype** veld is een stuk instructie dat een **actie doet op opgeslagen variabelen** die **geen bewerking** is :
  -> **registerbewerkingen** : dingen doe met registerplaatsen (verwisselen ofzo)
  -> **sprong** : kiezen tussen verschillende mogelijke volgende instructies of springen van de ene instructie naar een andere ergens anders in de sequentie van instructies (vaak vergelijkingen enzo om keuzes te maken en resultaten te evalueren)
  -> **geheugenoperatie** : operandi verplaatsen van register naar geheugen en omgekeert
- een **adresseermode** duid aan hoe we een **effectieve adres** moeten **berekenen** uitgaande van **adresveld**(**en**). Bv. wanneer het adres van een getal zelf opgeslagen zit in het geheugen
- een **constante** bv. "Add R2, R3, 1"

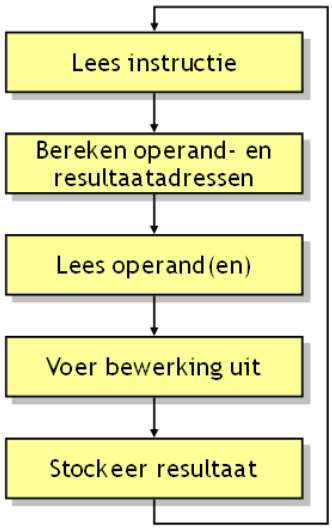
Zowel het **aantal** als de **grootte** van de **velden** in een instructie kan **variabel** zijn. Sommige **soorten velden** kunnen **meerdere keren** voorkomen in 1 instructie.

Meestal **ziet** een instructie **eruit** als **volgt** :

-> eerst een **opcode** die specifieert **welke bewerking** er moet gedaan worden, of een instructietype wanneer er bv data verplaatst moet worden

-> daarachter **meerdere adressen** om de verschillende **operandi** te kunnen ophalen of de geheugenplaats te specifiëren waar het **resultaat** naartoe moet.

⇨ De **generische instructiecyclus** is een **eindeloos herhaalde cyclus** die **uitgevoerd** wordt bij de **werking** van een **programmeerbare processor**. Bestaat uit het **inlezen** van **instructies**, daaruit **opmaken** wat waarop moet **gebeuren**, de **bewerkingen** uitvoeren en het resultaat **opslaan**.



Lees instructie uit het geheugen in het Instructieregister (IR); verhoog Programmateller (PC)

Gebruik adresseermodi en adresvelden om operand- en resultaatadressen te berekenen

Lees operanden uit het geheugen (tijdelijk) in registers

Voer bewerking uit en stockeer resultaat (tijdelijk) in een register

Stockeer resultaat in het geheugen

⇨ De **uitvoeringssnelheid** van de **instructies** hangt af van :

- de **snelheid** van het **datapad** : kan **verhogen** worden met **extra hardware**
- het **aantal keer toegangen** tot het **extern geheugen** : hoe **meer we data moeten** gaan **opzoeken**, hoe **meer tijd** het programma in beslag neemt. Het totaal aantal adresvelden van alle instructies samen is belangrijk. We gebruiken nu best **zo kort mogelijke instructies.**

De **lengte** van een **instructie** is hoofdzakelijk bepaald door het **aantal adresvelden** per instructie.
Korte instructies zijn nu instructies met **weinig adressen** in en dus zijn dus **snelle instructies.**
Nu **hoe korter** de instructies hoe **meer instructies** we moeten hebben per taak
We moeten dus **afwegen** of we **veel korte** instructies of **weinig langere instructies** zullen gebruiken.
We zullen nu in het **volgende voorbeeld** proberen duidelijk te maken dat **veel korte instructies beter zijn dan weinig lange**, door telkens de **instructies** voor het **voorbeeld** uit te werken en het **aantal geheugentoegangen** te **tellen**.

Voorbeeld : bereken **c = (a + b) × (a – b)**

⇨ Stel we gebruiken **instructies** met 32-bits **instructiedata** en eveneens 32-bit **adressen**.

Een instructie bestaat dan uit 1 woord met alle velden zonder de adressen : voor de **bewerking**
1 woord per **geheugenadresveld**.

Dat zijn dus voor n adressen **n+1 woorden**.

!!! Het aantal keer dat we dan **toegang** maken naar het **geheugen** is dan :

- **n+1 keer** om de n **geheugenadresvelden** en de instructies op te halen
  (dit is enkel het ophalen van bewerking en de locatie van de geheugenplaatsen
- **1** extra veld **per schrijfcommando of leescommande** om data **te transporteren**
  (hier maken we toegang om effectief waarden op te halen in de gespecifieerde locaties)

bv. Het commando **Mem[X] ← Mem[A] + Mem[B]** vraagt dus **3+1+3 = 7** geheugentoegangen :
3 om adressen op te halen, 1 om de + op te halen en 3 keer om de Mem[] op te halen.

⇨ **Invloed** van het **aantal adresvelden** (= de lengte) van een instructie op de performaniet van de bewerking **c = (a + b) × (a – b)** :

⇨ Instructies met **3 adressen** : voor **elke bewerking** worden dan in de **instructies 3 geheugenadressen** gespecifiëert. (2 voor operandi en 1 voor het resultaat) Hierdoor vragen de **instructies veel verwerking** en **opslagruimte**, maar het **aantal** instructies is **klein**.

```
Add X,A,B   Mem[X] ← Mem[A] + Mem[B]
Sub C,A,B   Mem[C] ← Mem[A] – Mem[B]
Mul C,X,C   Mem[C] ← Mem[X] × Mem[C]
```

We hebben dan **4 + 3 geheugentoegangen** per **instructie**

⇒ programma heeft **21 toegangen** nodig (3 keer 4+3)

⇨ Instructies met **2 adressen** : het **resultaat overschrijft** telkens de **eerste operand** bij de **dyadische** operaties. Hierdoor moeten we **slechts 2 geheugenadressen** specifiëren (voor de 2 operandi). Omdat we de **opslagplaats** van het **resultaat niet meer kunnen kiezen**, hebben we **verplaatsinstructies** ('move') nodig om de getallen soms van **geheugen te verplaatsen**. Elke instructie bevat dus nu **nog maar 2 adressen**, maar er zijn wel **meer instructies** :

```
Move X,A    Mem[X] ← Mem[A]
Add X,B     Mem[X] ← Mem[X] + Mem[B]
Move C,A    Mem[C] ← Mem[A]
Sub C,B     Mem[C] ← Mem[C] – Mem[B]
Mul C,X     Mem[C] ← Mem[C] × Mem[X]
```

We hebben dan **(3 + (2 of 3)) geheugentoegangen** per **instructie**. Dit is dus **2 keer** de **locatie** van de **variabelen** opvragen (adres), **1 keer** de **bewerking** opvragen, en dan **2 of 3 keer toegang maken** om effectief de **waarde van de variabelen te gaan ophalen** op het daarvoor opgehaalde adres. Bij een **move-operaties** is dit laatste **2** omdat we daar maar **1 operand** nodig hebben en die gewoon ergens andere moeten **wegschrijven**. Voor de andere moeten we **2 keer waarden ophalen** en **1 keer wegschrijven**.

⇒ programma heeft **28 toegangen** nodig (2 keer 5 en 3 keer 6)

- we hebben hier 1**,67 meer instructies** maar instructies zijn **sneller** (**slechts 5,6 geheugentoegangen per /instructie**)

⇨ Instructies met **1 adres:** we gebruiken een **speciaal register (ACC of Accumulator)** dat we altijd gebruiken. Om een **bewerking** met **2 operandi** uit te voeren moeten we dus **maar 1 locatie** van een operandus **specifiëren**, omdat de **andere** gewoon **zowiezo de ACC is**.

```
Load A      ACC ← Mem[A]
Add B       ACC ← ACC + Mem[B]
Store X     Mem[X] ← ACC
Load A      ACC ← Mem[A]
Sub B       ACC ← ACC – Mem[B]
Mul X       ACC ← ACC × Mem[X]
Store C     Mem[C] ← ACC
```

We hebben dan **(2 + 1) geheugentoegangen** per **instructie**. Adres en instructie ophalen, en resultaat effectief wegschrijven.

⇒ programma heeft **21 toegangen nodig** (7 keer 3)

⇨ Instructies **zonder adres** : we gebruiken een **LIFO-stapel** voor **data. Beide operanden** staan dan **bovenaan** stapel, en na de **bewerking worden ze vervangen** door het **resultaat**. Voor **verplaatsinstructies** gebruiken we **relatieve adressering.** Op deze manier moeten we nooit specifiëren waar variabelen vandaan moeten komen, omdat we weten dat het gewoon de bovenste zijn van de LIFO.

```
Load   OffsetA   Push ← Mem[base+OffsetA]
Load   OffsetB   Push ← Mem[base+OffsetB]
Add              Push ← Pop + Pop
Load   OffsetA   Push ← Mem[base+OffsetA]
Load   OffsetB   Push ← Mem[base+OffsetB]
Sub              Push ← Pop – Pop
Mul              Push ← Pop × Pop
Store  OffsetC   Mem[base+OffsetC] ← Pop
```

We hebben dan **(1 + (0 of 1)) geheugentoegangen** per **instructie.** We moeten **enkel** de **instructies** weten, en **eventueel iest wegschrijven** of **ophalen** wanneer er Mem[] staat. We zullen dan **data gaan halen** op **adressen** die **reeds in de LIFO staan** en we dus niet meer moeten opzoeken.

⇒ programma heeft **13 toegangen nodig**

⇨ Instructies met **3 adressen**, **waarvan 2** van een **registerbank** : we doen dit omdat de paar adresbits van een registerbank meestal nog wel passen in het opcode-woord zodat geen extra adreswoord nodig is. Een adres voro een register is meestal veel korter dan dat voor een extern geheugen.

```
Load R1,A       RF[1] ← Mem[A]
Add R2,R1,B     RF[2] ← RF[1] + Mem[B]
Sub R1,R1,B     RF[1] ← RF[1] – Mem[B]
Mul C,R1,R2     Mem[C] ← RF[1] × RF[2]
```

We hebben dan **(2 + 1) geheugentoegangen** per **instructie** nodig. Dus 1 keer om een echt adres op te halen, 1 keer voor instrucies en 1 keer om effectief data op te halen of weg te schrijven om het gespecifiëerde adres.

⇒ programma heeft **12 toegangen** nodig

Er zijn **verschillende combinaties** van **register- en geheugenadresseringen mogelijk**, o.a. …

⇨ Instructies met **3 registeradressen of** met **2 adressen**, **waarvan 1** van een **registerbank** : het systeem is hier dat we **telkens eerst** alle **nodige variabelen** voor een bewerking **in een registerbank** steken met **zo weinig mogelijk** (1 echte en 1 register) **geheugentoegang**, en dan de **bewerkingen** enkel uitvoeren op **registeradressen** (3 registeradressen).

```
Load R1,A       RF[1] ← Mem[A]
Load R2,B       RF[2] ← Mem[B]
Add R3,R1,R2    RF[3] ← RF[1] + RF[2]
Sub R4,R1,R2    RF[4] ← RF[1] – RF[2]
Mul R5,R3,R4    RF[5] ← RF[3] × RF[4]
Store R5,C      Mem[C] ← RF[5]
```

We hebben dan **(2 + 1) of (1 + 0) geheugentoegangen** per **instructie**. Ofwel verplaatsen we **data tussen register en ge**heugen (1 adres, 1 actie en 1 keer effectief lezen of schrijven), ofwel doen we een **bewerking** en hebben we enkel een **opcode** nodig.

⇒ programma heeft **12 toegangen** nodig

⇨ Deze **laatste manieren** van werken betekend een **tijdswinst** wanneer de meeste **data** of **tussenresultaten verschillende** malen **gebruikt** wordt, omdat **registertoegangen veel sneller** zijn dan geheugentoegangen. We gebruiken dit dan ook **in alle moderne processoren**.

2) AdresseerModi

Soms **coderen** we op bepaalde manier het **adres** van een **geheugenveld**, omdat dat dikwijls de **grootte** van het adresveld **reduceert**. De **adresseermodi** zijn verschillende manieren om een geheugenadres te specifiëren in een instructiewoord. Dit is **analoog** aan **hogere programmeertalen** (cfr. matrices).

Verschillende adresseermodi :

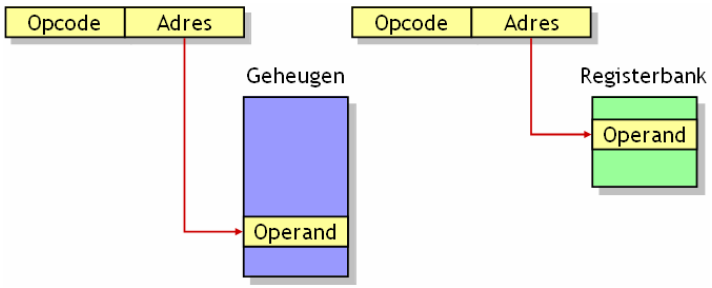
⇨ ***Impliciete adressering*** : het **adres zit impliciet** in de **opcode** gecodeerd. Een instructie bestaat dan zuiver uit opcode (zodat we geen appart veld nodig hebben voor het geheugenadres).

Bv - "ADD" betekent bij een 0-adres machine
"Vervang de twee waarden bovenaan de stapel door hun som"
- "CLRA" betekent bij een 1-adres machine "Reset de accumulator"

⇨ ***Onmiddelijke adressering*** : het **adresveld bevat niet het adres zelf** maar **direct de waarde** van de **operand** zelf. Dit wordt gebruikt voor **constanten**.

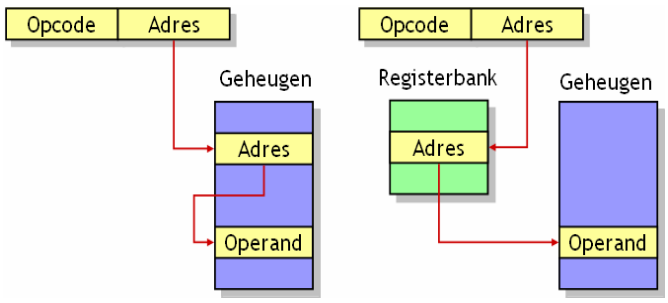
Een instructie bestaat dan uit opcode en een operand, zonder dat we eerst die operand nog moeten opzoeken met zijn adres.

⇨ ***Directe adressering*** : het **adresveld** bevat het **adres** van de **operand**. Het adres van een **geheugen** (MEM) is dikwijls **32 bit**, het adres van een **registerbank** (RF) is typisch **3 tot 8 bits**. We zetten dit verschil ook in onze instructies :
operand = **MEM[adres]** of **RF[adres]**. Wanneer we dus met registerbanken werken in plaats van geheugen kunnen we dus met kortere adressen werken. De instructie uitvoeren bestaat dan uit het ophalen van de opcode, het adres, en de effectieve waarde op het adres.



⇨ ***Indirecte adressering*** : het adresveld verwijst naar de plaats waar het eigenlijke adres van de operand zich bevindt. We moeten dan eerst het indirecte adres gaan ophalen, daarmee het directe adres gaan zoeken en dan daarmee de waarde gaan halen. We kunnen het adres ook in een registerbank steken omdat dat sneller te bereiken is :

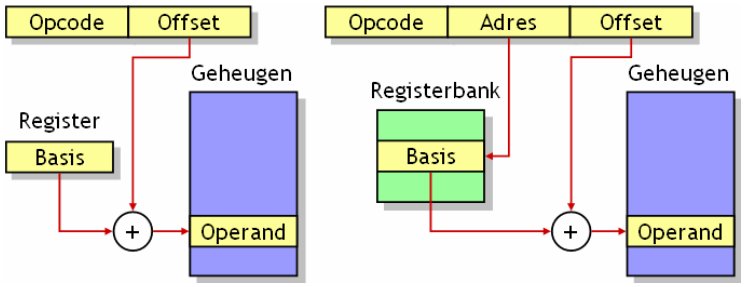
-> ***indirect***: eigenlijke adres in geheugen
⇒ operand = MEM[MEM[adres]]
-> ***register-indirect***: eigenlijke adres in register
⇒ operand = MEM[RF[adres]]



⇨ ***Relatieve adressering*** : het **effectieve adres** van de **gezochte waarde** bekomen we als de **som** van een **basisadres** + een (kleine) **offset**. Voordeel is dat de **kleine offset minder geheugen** inpakt als telkens het volledige adres te vermelden. De **code** voor de instructie wordt dan **operand = MEM[basis + offset].** Kan ook met **meerdere basissen in een register**, waarbij we dan een basis-adres voor het basis-register nodig hebben en een offset.

-> ***relatief***: het **basisadres** zit in een **impliciet register**. Het **adresveld** dat in de instructie staat is de **offset**. We gebruiken dit voor bv voor **sprongadressen** (basis = PC)

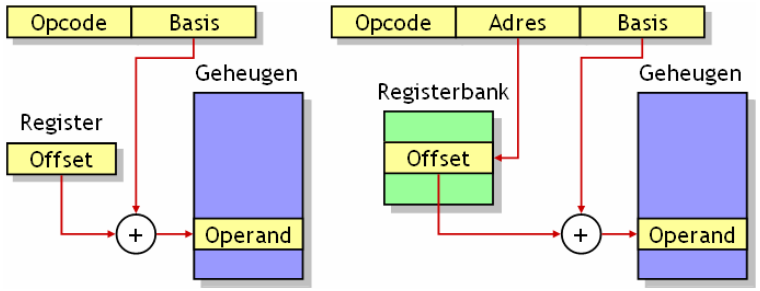
-> ***register-relatief***: in een **expliciet register** worden de **basissen gestockeerd** en in het **instructiewoord** gespecifieerd met een **adres**. (**basis = RF[adres]**). Gebruikt voor bv voor **opzoektabel** ('Look-Up Table')



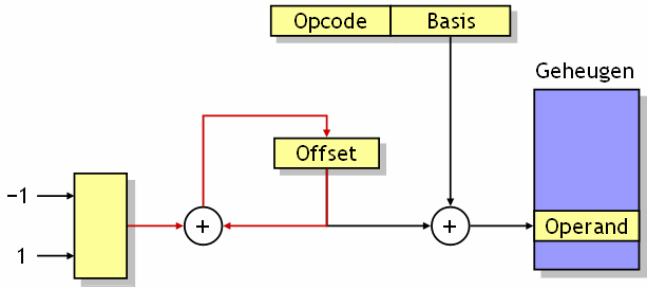
⇨ ***Geïndexeerde adressering*** : het **effectieve adres** van de gezochte waarde is de **som** van een in de **instructie gespecifieerd basisadres** + een **vaste (kleine) offset**. De **basis** is dus het **beginadres** van een matrix, stapel, wachtrij, … en de **offset** is de **index**. Dit is het omgekeerde van het vorige.

-> ***geïndexeerd***: de **offset** zit **vast opgeslagen** in **impliciet register** en het adresveld in de isntructie bevat de basis

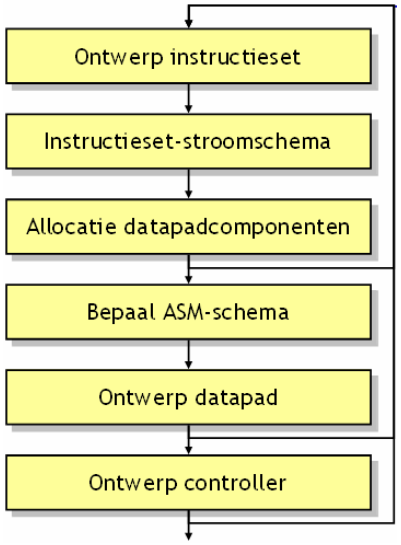
-> ***register-geïndexeerd***: de **offset** is niet vast **maar moet gekozen** worden uit een **expliciete registerbank** op basis van een **adres** in de **instructiecode** (offset = RF[adres]). Ook de **basis** zit (samen met de offset) in het **adresveld**.



⇨ ***Geïndexeerde adressering met autoincrement/autodecrement*** : geïndexeerde adressering waarbij de offset automatisch verhoogd (of verlaagd) wordt. Meestal ±1 maar soms ook ±2, ±4, …



ProcessorOntwerp



Wanneer we een **programeerbare processor ontwerpen** volgen we een vaste **volgorde** van **ontwerpstappen**. We ontwerpen eerst de **instructieset** waarmee we het algoritme zullen beschrijven. Dan beschrijven we alle **operaties** die in die **instructies** kunnen gebeuren en zullen we daar **componenten** voor **voorzien** op het **datapad**. Dan maken we een **ASM**-schema waar we de **instructies** in **cycli** verdelen en de **registertransfer/cyclus** bepalen. Op basis daarvan ontwerpen we dan eerst het **datapad** en dan de **controller**. Het **ontwerp** van het **datapad** hangt daarbij nauw samen met de **instructieset** en **omgekeert**. (alles wat de instructieset beschrijft moet de controller kunnen uitvoeren) De **controller** doen we pas op het **einde** aangezien deze het **geheel** moet **besturen**. (besturing kunnen we pas maken als we al hebben wat bestuurd moet worden).

⇨ Er zijn **2 soorten programmeerbare processoren** : In essentie moeten een compromis zoeken tussen de volgende uitersten :

-> **Ofwel** nemen we een **duur complex datapad** dat vele verschillende FU's heeft met veel geheugen enzo, waardoor we een **klein compact programma met weinig complexe instructies** kunnen maken omdat het datapad zoveel opties heeft.

-> **Ofwel** nemen we een **eenvoudig goedkoop datapad** dat niet veel functionaliteiten heeft, maar dan moeten we een **lang programma schrijven met veel simpele instructies** zodat het datapad dat aankan.

Deze 2 opties leiden tot het ontwerp van 2 soorten programmeerbare processoren die elk één van beide opties belichamen :

⇨ **CISC**: Complex Instruction Set Computer : deze voert **vele complexe (trage) instructies** uit
- meerdere operaties, instructietypes, adresseermodi en adresvelden
- **complex datapad** met veel FU's, veel registers en complexe verbindingen dat resulteert automatisch in een **lage klokfrequentie**
- **kort simpel programma**, maar elke **instructie** verbruikt **veel klokcycli**

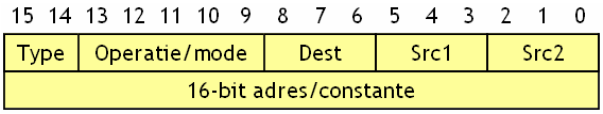
⇨ **RISC**: Reduced Instruction Set Comp. : voert slechts **enkele snelle (eenvoudige**) **instructies** uit
- weinig operaties en instructietypes; 0 of 1 adresvelden; weinig adresmodi
- **eenvoudig datapad** waardoor we een **hoge klokfrequentie** kunnen nemen
- **lang programma**, maar elke instructie verbruikt maar een **paar klokcycli**

⇨ Op gebied van **uitvoertijd** zijn **beide opties ongeveer gelijk**. In **complex datapad** met een **kort programma** met complexe instructies duurt het meestal **veel langer om 1 instructie** uit te voeren dan in het **simpele datapad** waar de **korte instructies snel na elkaar** uitgevoerd worden. Beide voeren hetzelfde programma uit in **ongeveer dezelfde tijd**.

⇨ Een ander **belangrijk aspect** voor de **snelheid** van een programma is de **mogelijkheid tot pipelining**. Een programma met **eenvoudig datapad** heeft **vele simpele instructies** en die kunnen **veel makkelijker gepipelined w**orden dan complexe instructies. Dus op gebied van **pipelining is de 2de optie beter**.

1) CISC-processoren : Complex Instruction Set Computer

Ontwerp CISC-instructieset



De **instructieset** van een **CISC-processor** bestaat uit **vele complexe (trage) instructies**. We zullen hier een CISC bespreken met **volgende specificaties** :

⇨ De **instructieset** bestaat uit **maximaal twee 16-bit woorden** :

1) het **eerste woord** is altijd gebruikt en dat bevat de **instructie-code**. Deze specifieert de **bewerking** die er gedaan moet worden en **eventuele register-adressen**. De **16 bits** van het eerste woord worden als volgt verdeeld :

⇨ **2 bits (Type**) zijn voor het **type van instructie**. We definiëren hier wat de instructie in essentie doet. Aangezien we **2 bits** hebben hebben we **4 mogelijke types** instructies :

- ⇨ **00 registerinstructies** : registeradressen waar we de bewerking mee uitvoeren
- ⇨ **01 verplaatsinstructies** : om waarden van geheugen / registerplaats te verplaatsen
- ⇨ **10 spronginstructies** : om door de isntructieset te springen
- ⇨ **11 andere instructies**, zoals NOP

⇨ **5 bits** (**Operatie/mode**) stellen we beschikbaar voor het **specifiëren** van de **bewerking** en de **adresseermodes**.

⇨ **9 bits** (**Dest, Src1, Src2**) zijn de **registeradressen** van de te gebruiken **operandi** en de locatie voor het **resultaat**. Hoewel **niet elke instructies evenveel registeradressen** nodig heeft, voorzien we toch **zowiezo 3 registeradressen van 3 bits**.

2) het **tweede woord** is een **geheugenadres** en wordt **enkel gebruikt** wanneer we **data willen importeren van het geheugen**. We kiezen hier namelijk voor een **systeem** waarbij we telkens **eerst data uit het geheugen laden/stockeren in registers** met behulp van het **16-bit adresveld**, en voor de **bewerkingen enkel registeroperaties** zullen gebruiken zodat we niet telkens in het geheugen moeten. Dit veld kan eventueel ook gewoon een **constante** bevatten in plaats van een geheugenadres.

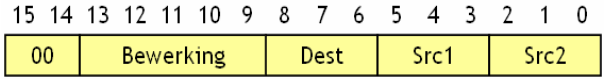
We zullen nu de 4 mogelijk types van instructies verder bespreken :

00) Registerinstructies

Een **registerinstructie** is een **bewerking** (operatie OP) met **waarden** die in het **register** opgslagen zijn en waarbij het **resultaat** van de bewerking **opnieuw opgeslagen** wordt in het **register**. We kunnen operaties uitvoegen **op zowel 1 als op 2 operandi**.

⇨ OP **Dest,Src1,Src2** : RF[Dest] ← RF[Src1] OP RF[Src2]

⇨ OP **Dest,Src1** : RF[Dest] ← RF[Src1] OP

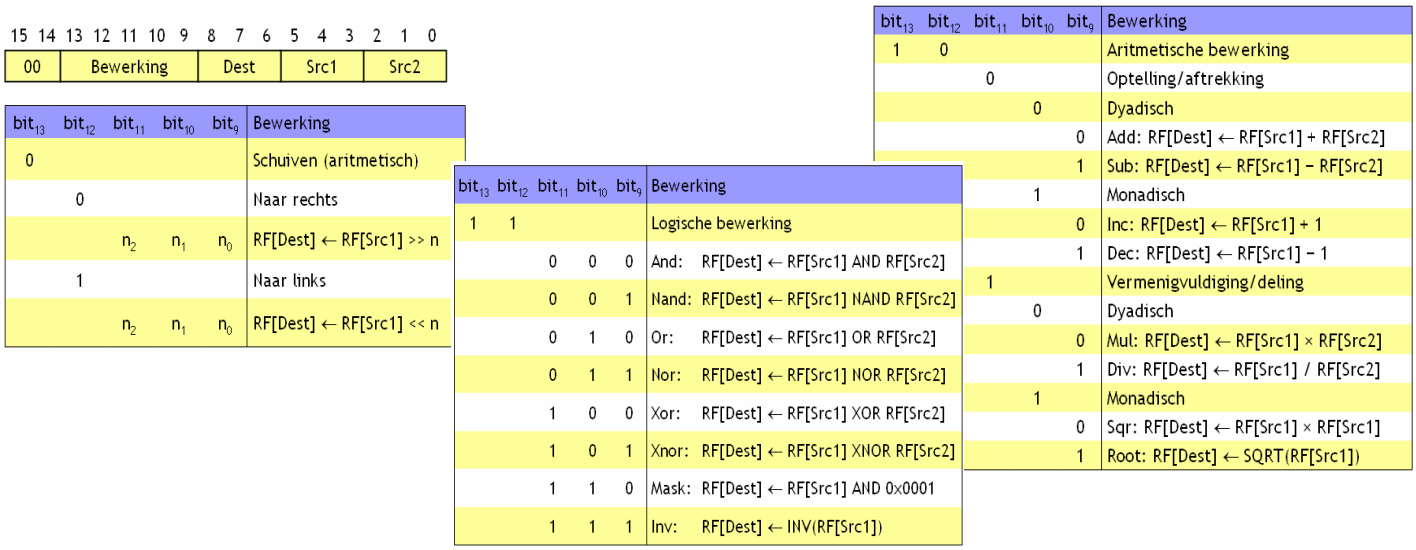


De **bewerking OP** kunnen volgende dingen zijn :

- **aritmetische** operaties (optellen, wortels, …)
- **logische** operaties (allerlei poorten als AND, XOR …)
- **schuifoperaties** (shiften naar links en rechts, …)

De **keuze** van de bewerkingen hangt af van de **frequentie** van hun **gebruik** in **typische toepassingen**.

⇨ Met de **2 eerste bits** van het **instructiewoord** bepalen we dat we een **registerinstructie** doen, met de **5 bits** van **bewerking specifiëren** we welke bewerking we willen doen :



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00			Bewerking				Dest			Src1			Src2		

bit_{13}	bit_{12}	bit_{11}	bit_{10}	bit_9	Bewerking
0					Schuiven (aritmetisch)
	0				Naar rechts
		n_2	n_1	n_0	RF[Dest] ← RF[Src1] >> n
	1				Naar links
		n_2	n_1	n_0	RF[Dest] ← RF[Src1] << n

bit_{13}	bit_{12}	bit_{11}	bit_{10}	bit_9	Bewerking
1	1				Logische bewerking
		0	0	0	And: RF[Dest] ← RF[Src1] AND RF[Src2]
		0	0	1	Nand: RF[Dest] ← RF[Src1] NAND RF[Src2]
		0	1	0	Or: RF[Dest] ← RF[Src1] OR RF[Src2]
		0	1	1	Nor: RF[Dest] ← RF[Src1] NOR RF[Src2]
		1	0	0	Xor: RF[Dest] ← RF[Src1] XOR RF[Src2]
		1	0	1	Xnor: RF[Dest] ← RF[Src1] XNOR RF[Src2]
		1	1	0	Mask: RF[Dest] ← RF[Src1] AND 0x0001
		1	1	1	Inv: RF[Dest] ← INV(RF[Src1])

bit_{13}	bit_{12}	bit_{11}	bit_{10}	bit_9	Bewerking
1	0				Aritmetische bewerking
		0			Optelling/aftrekking
			0		Dyadisch
				0	Add: RF[Dest] ← RF[Src1] + RF[Src2]
				1	Sub: RF[Dest] ← RF[Src1] – RF[Src2]
			1		Monadisch
				0	Inc: RF[Dest] ← RF[Src1] + 1
				1	Dec: RF[Dest] ← RF[Src1] – 1
		1			Vermenigvuldiging/deling
			0		Dyadisch
				0	Mul: RF[Dest] ← RF[Src1] × RF[Src2]
				1	Div: RF[Dest] ← RF[Src1] / RF[Src2]
			1		Monadisch
				0	Sqr: RF[Dest] ← RF[Src1] × RF[Src1]
				1	Root: RF[Dest] ← SQRT(RF[Src1])

01) Verplaatsinstructies

De **verplaatsingsinstructies** zorgen voor de **uitwisseling** van **gegevens** tussen **geheugen** en **registerbank**. Telkens wanneer we een **bewerking** willen uitvoeren zullen we eerst de **variabelen** in het **register laden** en na de bewerking het **resultaat terug naar het geheugen schrijven** als we het niet meer nodig hebben.

⇨ We hebben hier **slechts 2 moglijke operaties**. We kunnen de uit te voeren operatie dan ook specifiëren met **1 bit OP** :

-> **Load** (Op = 0) : verplaaten waarden van **geheugen → naar register**

-> **Store** (Op = 1) : verplaatsen waarden van **register → naar geheugen**

⇨ De **overblijvende 4 bits** van '**Operatie/mode**' worden gebruikt voor de **adresseermode** van het geheugen. **Adressmode** :

-> De **keuze** van adresseermode hangt af van de **frequentie** van hun **gebruik** in typische toepassingen

-> **Src2** wordt niet gebruikt en kan dus **eventueel gebruikt worden als offset**

-> De **adresseermode** zal ook bepalen of **tweede woord** (adres/constante) **al dan niet gebruikt** wordt. Merk op dat we het 2de instructiewoord nooit gebruikt hebben bij de registerinstructies

Verschillende **Load**-mogelijkheden **(OP=0) :** (met het aantal nodige woorden voor de instructie)

⇨ Het 2de woord is een **constante** en we laden dat direct in het geheugen (=**onmiddellijk adres**)
(**2** woorden): RF[Dest] ← Constante

⇨ Het 2de woord is een **direct adres** van een **geheugenplek** waarvan we de waarde inlagen
(**2** woorden):RF[Dest] ← Mem[Adres]

⇨ Het 2de woord is een **indirect adres** van een **geheugenplek** waar we het echte adres staat
(**2** woorden):RF[Dest] ← Mem[Mem[Adres]]

⇨ **Register-indirect** adres : het **nodige geheugenadres** staat reeds in het **register**
(**1** woord):RF[Dest] ← Mem[RF[Src1]]

⇨ **Register-relatief** adres : het **nodige adres** bouwen we uit een **basis** in het **register** en een **offset**
(**1** woord):RF[Dest] ← Mem[RF[Src1] + Src2] =RF[Dest] ← Mem[Basis + Offset]

⇨ **Register-geïndexeerd** adres : nodige adres bouwen uit een **vaste basis** en **offset in register**
(**2** woorden):RF[Dest] ← Mem[Adres + RF[Src1]] =RF[Dest] ← Mem[Basis + Offset]

⇨ We **copiëren** de **inhoud** van een **register** naar een ander **register** :
(1 woord):RF[Dest] ← RF[Src1]

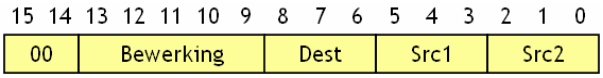
Verschillende **Store**-mogelijkheden (OP=1) : (met het aantal nodige woorden voor de instructie)

- ⇨ Een **onmiddellijk adres** gebruiken is **zinloos**! (= bij load is dit gewoon een cosntante in de programmacode inladen, maar we niet een constante wegschrijven in de programmacode)
- ⇨ Al de rest is hetzelfde maar dan in omgekeerde richting

Toewijzing van de **bewerking-bits** : De **eerste** bit (12) slaat op **Load/Store**, die **daarop** (13) kiest of er een **tweede woord** gebruikt wordt in de instructie, en de overige (11, 10, 9) de **adresmode** :

- ⇨ **Bit 13** : 0 = **Load** / 1 = **Store**
- ⇨ **Bit 12** : 0 = **twee woorden** (er wordt een adres/constante gebruikt) / 1 = **één woord**

Bits 11,10,9 : 000 = **onmiddellijk adres** (enkel bij Load)
001 = **direct adres**
010 = **indirect adres**
011 = **relatief adres**
100 = **geïndexeerd adres**
101 = **kopieer register** (dan ook bit 13 = 0)



10) Spronginstructies

De **spronginstructies** dienen om de **uitvoering** in het programma te **verplaatsen**. (we verplaatsen ons gecontroleerd **van de ene instructie naar een andere** die **ergens anders** in de instructieset voorkomt) Ook hier hebben we **4 mogelijke instructies** voor **sprongen**, die we specifiëren met 2 bits, namelijk 12 en 13. De 4 mogelijkheden :

⇨ ***00) JMP*** : een **onconditionele sprong** (2 woorden) waar we een **adres** specifiëren waar we **direct naartoe** gaan. PC ← Adres

⇨ ***01) CJMP*** : **conditionele sprong** waarbij we als we aan een **voorwaarde** voldoen gewoon het **programma volgen**, en **anders** overgaan naar een **gespecifiëerd adres**. (2 woorden)

IF Status=0 THEN (PC ← PC + 1) ELSE (PC ← Adres)

⇨ ***10) JSR*** : **spring naar subroutine** : **we slaan in het register op waar we zaten** met de programma-uitvoer (terugkeeradres), we gaan naar een **subroutine**. De **terugkeeradressen** worden opgelsagen in een **LIFO** waarvan we dus het **schrijfadres verhogen**. (2 woorden)

Mem[RF[Src1]] ← PC + 1 **terugkeeradres** op LIFO met
RF[Src1] het LIFO-**schrijfadres**
en (RF[Src1] – 1) het **leesadres**
RF[Src1] ← RF[Src1] + 1 **verhoog LIFO-schrijfadres**
PC ← Adres **spring naar subroutine**

⇨ ***11) RTS*** : **keer terug uit subroutine** : we halen het terugkeeradres uit de LIFO, veranderen het schrijfadres terug en keren terug naar het adres opgelsagen in de LIFO(1 woord)

RF[Src1] ← RF[Src1] – 1 **verminder LIFO-schrijfadres**;
RF[Src1] bevat nu ook het LIFO-adres van het terugkeeradres
PC ← Mem[RF[Src1]] **lees terugkeeradres** uit LIFO

De **3 overige van de 5 bits** worden **gebruikt** om de **bewerking** verder te **definiëren**

bit_{13}	bit_{12}	bit_{11}	bit_{10}	bit_9	Bewerking
0	0	X	X	X	NOP (doe niets)
0	1				Zet op 0
		0	X	X	CLR: RF[Dest] ← 0
		1	X	X	ClrS: Status ← 0
1	0				Vergelijk RF[Src1] en RF[Src2] IF ... THEN Status ← 1 ELSE Status ← 0
		0	0	0	GT: RF[Src1] > RF[Src2]
		0	0	1	GE: RF[Src1] ≥ RF[Src2]
		0	1	0	LT: RF[Src1] < RF[Src2]
		0	1	1	LE: RF[Src1] ≤ RF[Src2]
		1	0	0	EQ: RF[Src1] = RF[Src2]
		1	0	1	NE: RF[Src1] ≠ RF[Src2]
		1	1	X	Niet gebruikt ⇒ ongeldige instructie
1	1	X	X	X	SetS: Status ← 1

Voorstelling van instructies :

Een **instructiewoord** is dus een **binaire reeks** van **16 bits** die de machine inleest, **analyseert** en **uitvoert**. Meestal schrijven we voor mensen elk woord **hexadecimaal** om schrijffouten te vermijden.

bijv. 0010000111110000 = 0x21F0

De **hardware** heeft echter de **bits appart nodig** en "**interpreteert**" deze als een binaire instructie. Met **interpreteren** bedoelen we dat de **controller** eingelijk **op basis van deze sequenties** van bits bepaalde **commando's** zal geven aan het datapad. :

0010000111110000 : registerbewerking
00**10**000111110000 : aritmetisch
0010**000**111110000 : optelling
0010000**111**110000 : resultaat in RF[7]
0010000111**110**000 : operand 1 = RF[6]
0010000111110**000** : operand 2 = RF[0]

Wanneer we een **algoritme schrijven** doen we dat in een **assembleertaal**. Dat is een **mnemonische voorstelling** van de **acties** die moeten **uitgevoerd** worden. We schrijven dit in een **redelijk menselijke taal**. Er bestaat dan een **1-naar-1 relatie** met **machinetaal**, waarbij we elke beschreven actie **vertalen** naar een voorstelling in bit's voor de machine die op die manier weet wat hij moet doen.

bijv. 0x21F0 ≡ ADD 7,6,0

We noemen het **compileren** van een **mnemonische assembleertaal** naar **machinecode Assembleren**.

- ⇨ De **vertaling** van instructies kan gebeuren **door elke instructie** via een eenvoudige **opzoektabel** te **vertalen** naar **machine-code**.
- ⇨ We kunnen **variabelen** een **betekenisvolle naam** geven (makkelijker om te gebruiken)
  De **assembler** kent dan **zelf een adres toe** aan de naam die je kiest
  De adressen worden **relatief t.o.v. het begin** van het **datageheugen** uitgedeeld
- ⇨ We kunnen ook een **betekenisvolle naam** (**label**) geven aan de **adressen** van de **geheugenplaatsen** die we gebruiken.
  De **Assembler** kent dan **zelf een adres toe** door er een **springadres** bij te tellen (relatief t.o.v. begin programmageheugen). Dit springadres is het verschil tussen de namen die wij geven en de namen die effectief moeten gebruikt worden

Nu volgen enkele **eenvoudige vertalingstabellen** tussen **assembleertaal** en **machine-code** :

⇨ Voor de **registeringsinstructies** :

Assembleertaal	Opcode	D	S1	S2	A	Actie
ASR n	$0000n_2n_1n_0$	√	√	X	–	RF[D] ← RF[S1] >> n
ASL n	$0001n_2n_1n_0$	√	√	X	–	RF[D] ← RF[S1] << n
ADD D,S1,S2	0010000	√	√	√	–	RF[D] ← RF[S1] + RF[S2]
SUB D,S1,S2	0010001	√	√	√	–	RF[D] ← RF[S1] – RF[S2]
INC D,S1	0010010	√	√	X	–	RF[D] ← RF[S1] + 1
DEC D,S1	0010011	√	√	X	–	RF[D] ← RF[S1] – 1
MUL D,S1,S2	0010100	√	√	√	–	RF[D] ← RF[S1] × RF[S2]
DIV D,S1,S2	0010101	√	√	√	–	RF[D] ← RF[S1] / RF[S2]
SQR D,S1	0010110	√	√	X	–	RF[D] ← RF[S1] × RF[S1]
ROOT D,S1	0010111	√	√	X	–	RF[D] ← Root(RF[S1])
AND D,S1,S2	0011000	√	√	√	–	RF[D] ← RF[S1] AND RF[S2]
NAND D,S1,S2	0011001	√	√	√	–	RF[D] ← RF[S1] NAND RF[S2]
OR D,S1,S2	0011010	√	√	√	–	RF[D] ← RF[S1] OR RF[S2]
NOR D,S1,S2	0011011	√	√	√	–	RF[D] ← RF[S1] NOR RF[S2]
XOR D,S1,S2	0011100	√	√	√	–	RF[D] ← RF[S1] XOR RF[S2]
XNOR D,S1,S2	0011101	√	√	√	–	RF[D] ← RF[S1] XNOR RF[S2]
MASK D,S1	0011110	√	√	X	–	RF[D] ← RF[S1] AND 0x0001
INV D,S1	0011111	√	√	X	–	RF[D] ← INV RF[S1]

⇨ Voor de **verplaatsingsinstructies** :

Assembleertaal	Opcode	D	S1	S2	A	Actie
LOAD D,#waarde	0100000	√	X	X	√	RF[D] ← waarde
LOAD D,adres	0100001	√	X	X	√	RF[D] ← Mem[adres]
LOAD D,(adres)	0100010	√	X	X	√	RF[D] ← Mem[Mem[adres]]
LOAD D,(S1)	0101010	√	√	X	–	RF[D] ← Mem[RF[S1]]
LOAD D,(S1)+S2	0101011	√	√	√	–	RF[D] ← Mem[RF[S1]+S2]
LOAD D,adres+(S1)	0101100	√	√	X	√	RF[D] ← Mem[adres+RF[S1]]
STOR adres,S1	0110001	X	√	X	√	Mem[adres] ← RF[S1]
STOR (adres),S1	0110010	X	√	X	√	Mem[Mem[adres]] ← RF[S1]
STOR (D),S1	0111010	√	√	X	–	Mem[RF[D]] ← RF[S1]
STOR (D)+S2,S1	0111011	√	√	√	–	Mem[RF[D]+S2] ← RF[S1]
STOR adres+(D),S1	0111100	√	√	X	√	Mem[adres+RF[D]] ← RF[S1]
COPY D,S1	01XX101	√	√	X	–	RF[D] ← RF[S1]

⇨ Voor de **spronginstructies** :

Assembleertaal	Opcode	D	S1	S2	A	Actie
JMP adres	1000XXX	X	X	X	√	PC ← adres
CJMP adres	1001XXX	X	X	X	√	IF (Status=0) THEN (PC ← PC+1) ELSE (PC ← adres)
JSR adres,(S1)	1010XXX	X	√	X	√	Mem[RF[S1]] ← PC+1; RF[S1] ← RF[S1]+1; PC ← adres
RTS (S1)	1011XXX	X	√	X	–	RF[S1] ← RF[S1]−1; PC ← Mem[RF[S1]]

⇨ Voor de **andere instructies** :

Assembleertaal	Opcode	D	S1	S2	A	Action
NOP	1100XXX	X	X	X	–	–
CLR D	11010XX	√	X	X	–	RF[D] ← 0
ClrS	11011XX	X	X	X	–	Status ← 0
SetS	1111XXX	X	X	X	–	Status ← 1
GT S1,S2	1110000	X	√	√	–	IF (RF[S1] > RF[S2]) THEN (Status ← 1) ELSE (Status ← 0)
GE S1,S2	1110001	X	√	√	–	IF (RF[S1] >= RF[S2]) THEN (Status ← 1) ELSE (Status ← 0)
LT S1,S2	1110010	X	√	√	–	IF (RF[S1] < RF[S2]) THEN (Status ← 1) ELSE (Status ← 0)
LE S1,S2	1110011	X	√	√	–	IF (RF[S1] <= RF[S2]) THEN (Status ← 1) ELSE (Status ← 0)
EQ S1,S2	1110100	X	√	√	–	IF (RF[S1] = RF[S2]) THEN (Status ← 1) ELSE (Status ← 0)
NE S1,S2	1110101	X	√	√	–	IF (RF[S1] <> RF[S2]) THEN (Status ← 1) ELSE (Status ← 0)

Programmavoorbeeld

We **bespreken** als **voorbeeld** het volgende **algoritme** : Bepaal de **som**, het **minimum** en het **maximum** van **1024 getallen**, opgeslagen in een **reeks A[i].**

⇨ **Beschrijving** van het programma in een **hoog-niveau taal**:

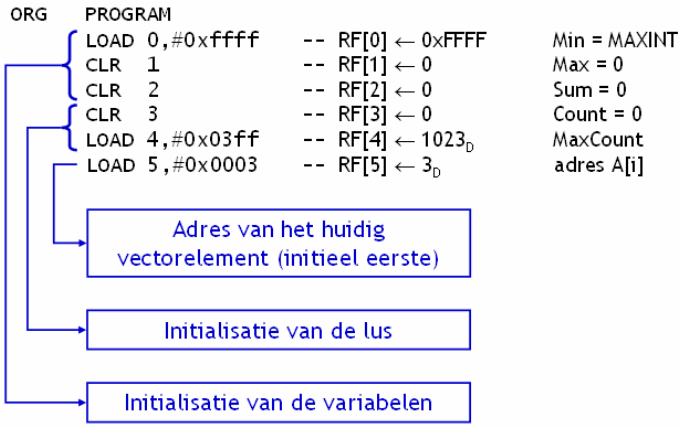
```
Min = MAXINT
Max = 0
Sum = 0
FOR Count: 0..1023
    Sum = Sum + A[i]
    IF A[i] > Max THEN Max = A[i]
    IF A[i] < Min THEN Min = A[i]
END FOR
```

⇨ **Beschrijving** van het programma in een **assembleertaal** :

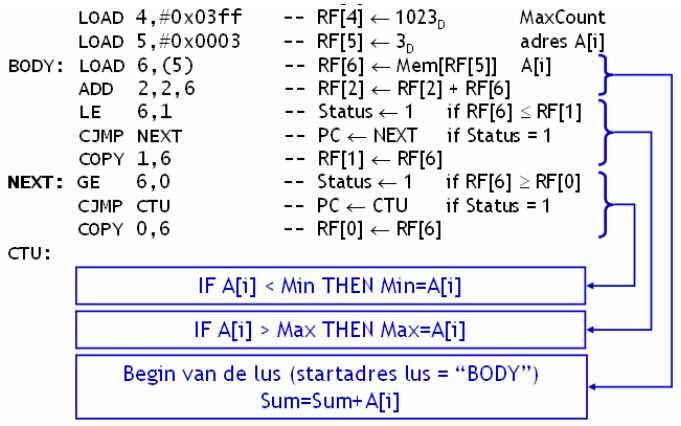
-> Het **algoritme begint** met het **creëren** van **variabelen**, waarvoor we **geheugenplaatsen** moeten **voorzien**. In tekst start dit met **ORG DATA**, met daarna de **declaraties** van de **variabelen**. We gebruiken een **datageheugen** met **adressen** met notatie die begint bij **0x0000**. We noemen **eerst de naam** van de variabele en daarachter **wat voor variabele** hij is.

-> Daarna volgt **ORG PROGRAM** : Dit geeft het **begin** van het **programma** aan. Het programma **start** op de **geheugenplaats** met **startadres (0x0403)**. Dit adres **verandert elke keer** wanneer een bijkomende **variabele gedeclareerd** wordt, waardoor alle **sprongadressen** gewijzigd moeten worden!

-> het programma **begint** met het **intialiseren** van de **variabelen** (waarde toekannen). Het komt erop neer dat we een aantal **registerplaatsen inladen** vanuit het **geheugen**. We initialiseren **min** als een **groot getal**, **max** en **sum** worden op **0** gezet. We intialiseren **lus-variabelen**, namelijk een **teller** (op 0 zetten) en een **maximumwaarde** (1023 hier). Verder hebben we ons **eerste element** van de 1024 nodig.

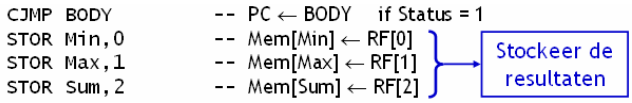


-> Daarna komt de **BODY** van het **programma**. We beginnen de **lus** met het **optellen** van het **niewe getal** bij de **som**, daarna **kijken** we of hij **groter** is dan **max**. Is hij **groter**, dan is hij zeker **niet kleiner** dan de **kleinste**. We zetten hem in **max** en gaan naar **CTU**. Is hij **niet groter** dan max dan is hij **mss wel kleiner** dan de **kleinste** en dan **zetten** we hem als **kleinste**. We gaan zowiezo naar **CTU** voor het einde van de lus



-> **CTU** is het **deel** dat de **lus beëindigd** en **opnieuw start**. Eerst **verhogen** we de **lusteller**, we **kijken** of we **nog niet aan het maximum** zitten, en we **halen** het **adres** van **BODY** op en gaan naar daar. **PC** is het **adres** waar de **instructiewoorden** van het **moment in geladen** worden (denk ik)

-> op het **einde** besluiten we met het **opslaan** van de **nuttige variabelen min, mas en sum** in hun daarvoor voorziene **geheugenplaatsen**



A[i] wordt 1 maal geladen in RF[6] en wordt 5 maal gebruikt: dit bespaart vermogen en verhoogt de snelheid

⇨ In **machine-code** zal dit **programma** er zo **uitzien** (slechts stuk getoont)

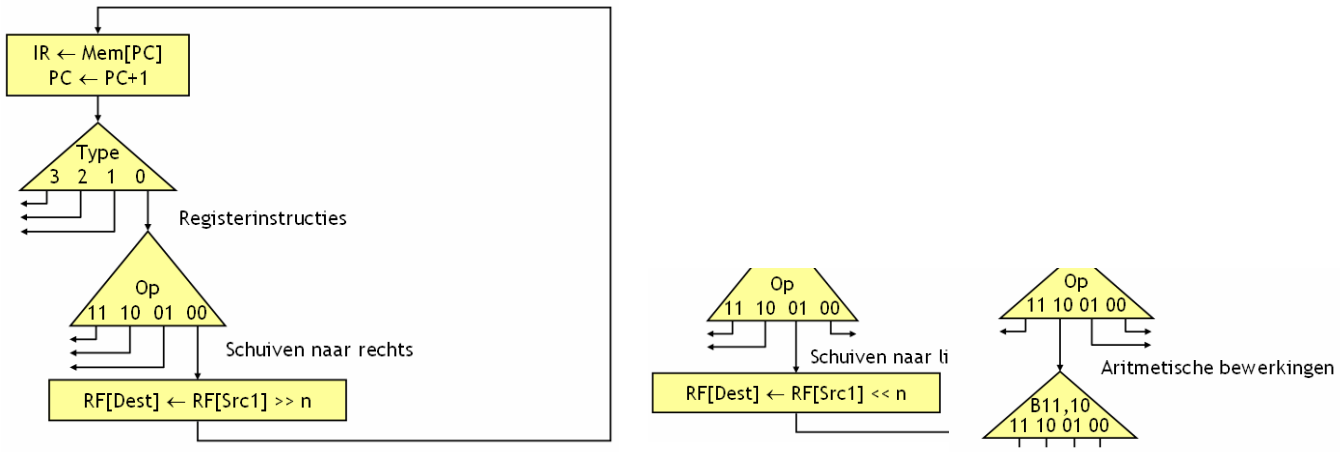
```
4003  0100000000XXXXXX   -- LOAD 0,#0xFFFF
4004  1111111111111111
4005  11010XX001XXXXXX   -- CLR 1
4006  11010XX010XXXXXX   -- CLR 2
4007  11010XX011XXXXXX   -- CLR 3
4008  0100000100XXXXXX   -- LOAD 4,#0x03FF
4009  0000001111111111
400A  0100000101XXXXXX   -- LOAD 5,#0x0003
400B  0000000000000011
400C  0101010110101XXX   -- LOAD 6,(5)          BODY
400D  0010000010010110   -- ADD 2,2,6
400E  1110011XXX110001   -- LE 6,1
400F  1001XXXXXXXXXXXX   -- CJMP NEXT
4010  0100000000010010
```

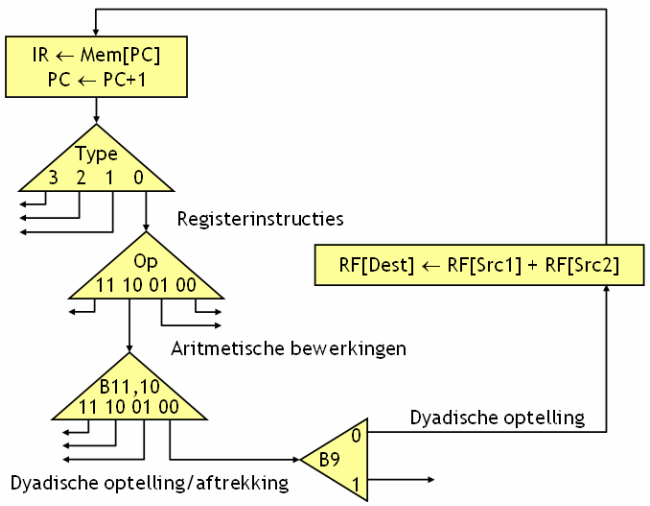
Instructieset-stroomschema ('Instruction Set Flowchart')

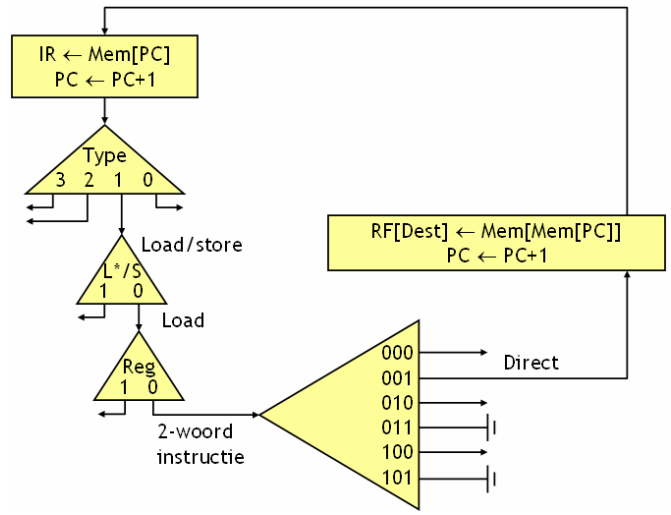
Een **instructieset-stroomschema** is een **visuele voorstelling** van **de set instructies** (i.p.v. in tabelvorm). Dit is een **duidelijkere manier** wanneer we dit doen op een **groot genoeg stuk papier**. Het is een **handige manier** om de **instructiedecoder** te ontwerpen. Wat we doen is **elke mogelijke instructie** die het **datapad** kan **verwezelijken** voorstellen in een **stroomschema** volgens de **voorstelling** ervan in het **instructiewoord**. De **instructie-decoder** (controller) zal op basis van het schema **makkelijk ontworpen** kunnen worden.

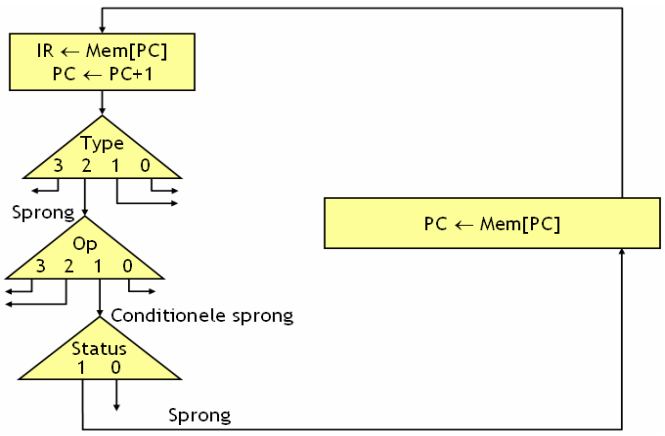
⇨ Voor het **ontwerpen** van de **instructie-decoders** hebben we nog **veel vrijheid**, aangezien de **enige architecturale beslissing** die we tot hiertoe genomen hebben is de **veronderstelling** dat we **beschikken over** een - geheugen (Mem)
- registerbank (RF)
- programmateller (PC)
- instructieregister (IR)
- statusvlag (Status)

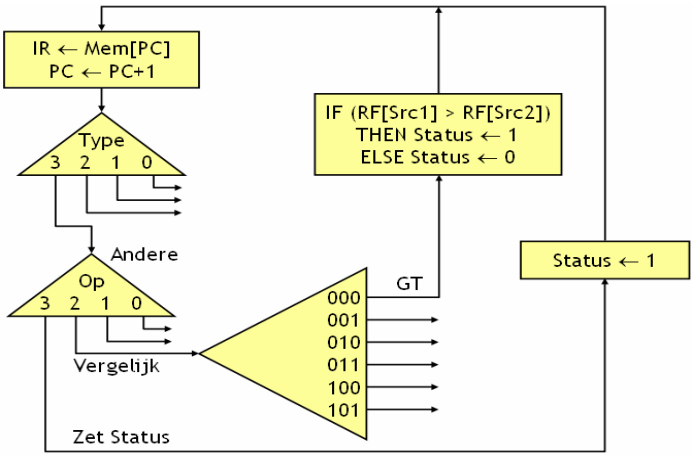
⇨ We zullen nu de **tabellen** van **daarstraks** vertalen in een **schema** : we geven enkel voorbeelden
-> **Type** kiest het **soort** instructie
-> **Op** kiest de **exacte** instructie
Het **schema begint** telkens **bovenaan** waar we op **basis** van de **PC** (program counter) de **volgende instructies inlezen** in het **IR** (instructieregsiter). We **verhogen** de **PC** om **volgende keer** de volgende **instructie** te kunnen ophalen. Op basis van de **inhoud** van de **IR** bepaald de **controller** dan de **juiste actie voor het datapad.**











Allocatie datapadcomponenten

We moeten nu het **datapad ontwerpen** zodat het **in staat is alle bewerkingen die in het algoritme** staan **uit te voeren** op commando van de **controller**. We hebben tot nu toe nodig :

⇨ **Registers** : **Extern geheugen** (1 lees/schrijfpoort; 64K×16), een **Registerbank** (2 lees- & 1 schrijfpoort; 8×16) en een **Statusvlag** (1 bit)

⇨ **FU's**: en **ALU** (aritmetische logische eenheid), **Comparator**, 16-bit **links/rechts schuifoperatie**

Om voor **sommige toepassingen** een **hogere snelheid** te halen kunnen **bijkomende registers en/of FU's** toegevoegd worden, maar let wel op dat de **instructieset** moet **herzien** worden bij elke **wijziging** van het **datapad** !!

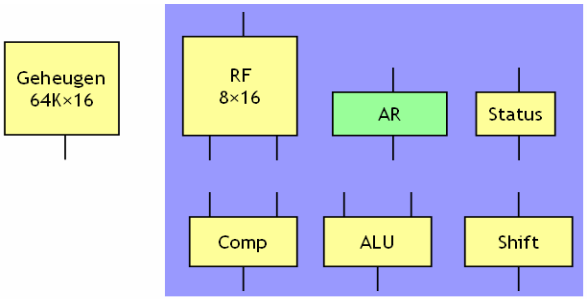
We beslissen hier om **één extra toevoeging** te voorzien omwille van de snelheid. We willen de **traagste instructie** van de set **versnellen**, namelijk het **indirect laden** van gegevens uit het geheugen.
RF[Dest] ← Mem[Mem[adres]] & adres ← Mem[PC]
Deze instructie leest **3 × uit extern geheugen** en duurt bv. 150 ns (als we 50 ns per toegang rekenen).

⇨ De **instructie beperkt** hiermee de **maximum klokfrequentie** tot **6,7 MHz** (= 1s delen door 150 ns) omdat **elke registerinstructies** dan **ook 150 ns** zal duren door de klok, zelfs als hun

combinatorische vertraging bv maar 10 ns is. (we moeten onze **klokfrequentie** voorzien op het **kritische pad**, dus op de duur van de langste operatie).

Als **extra aanpassing** zullen we nu **zorgen** dat **elke externe geheugentoegang** in een **aparte klokcyclus** uitgevoert wordt, waardoor we de **klokcycli tot 50 ns kunnen beperken**. (dit is **pipelining**; het **opsplitsen** van **bewerkingen** zodat ze **individueel minder lang duren** en we dus een **snellere klok** kunnen gebruiken). De **maximum klokfrequentie** wordt nu **20 MHz**, wat **aanzienlijk sneller** is. De **traagste instructie** duurt nu **3 klokcycli** van **50 ns,**dus **nog steeds 150ns**, maar een **registerinstructie** die op de klok wacht (elke instructie wacht o de klok) **duurt nu nog maar 50 ns** (en vroeger 150).

⇨ We zullen dus nu een **bijkomend *Adresregister*** AR **toevoegen** om een **adres**, dat uit **geheugen gelezen** wordt, te bewaren voor gebruik in de volgende klokcyclus. We zullen dus bij **indirecte adressering** eerst het **adres opzoeken**, **opslaan** in een **register** en dan in de **volgende clokcyclus** data op dat gana ophalen.

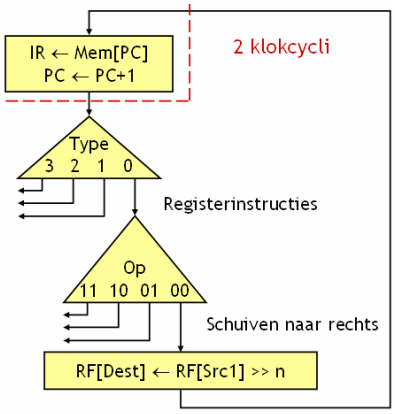


ASM-schema

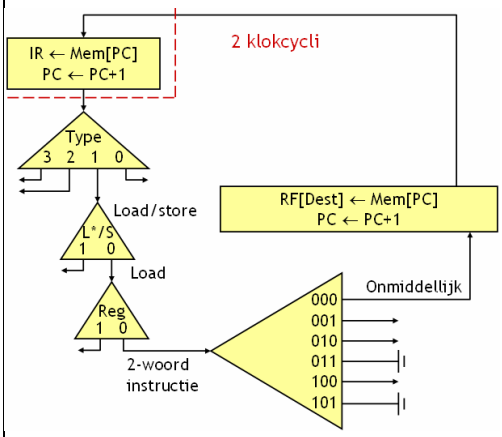
Het **ASM-stroomschema** is een **schema** dat aangeeft **voor elke klokcylcus** aan **welke registertransfers** erin gebeuren. Aangezien het **IS-stroomschema** aangeeft **welke registertransfers** er in een **instructie** gebeuren, moeten we hierin enkel uitzoeken **welke stukken van de instructies samen in één klokcyclus** gebeuren. We **splitsen** dus elke **instructie** op in (zo weinig mogelijk) verschillende **klokcycli**. We doen dat opdat er **geen hardware conflicten** zouden ontstaan tussen de **operatie** en de **datatransfers** in **eenzelfde klokcyclus**.

⇨ We zullen nu **verschillende IS-schema's** verdelen in **verschillende klokcycli**. De stippellijnen bakenen de stukken af die in 1 klokcyclus gebeuren :

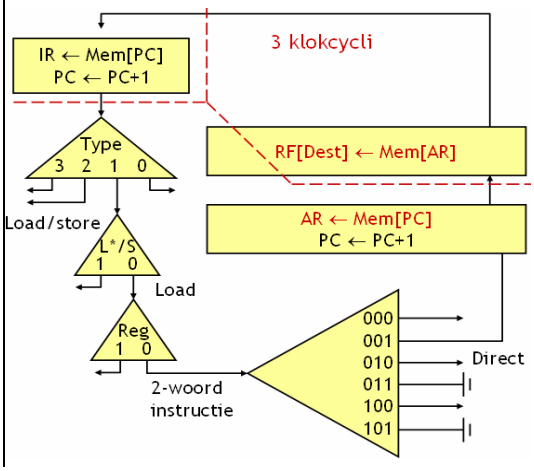
-> Aritmetisch schuiven



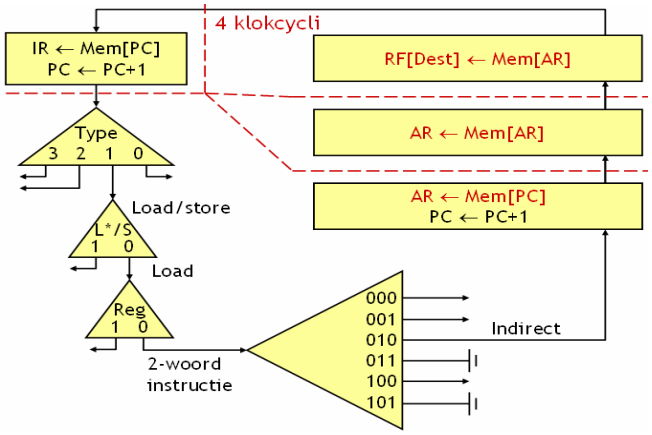
-> Een constante inladen



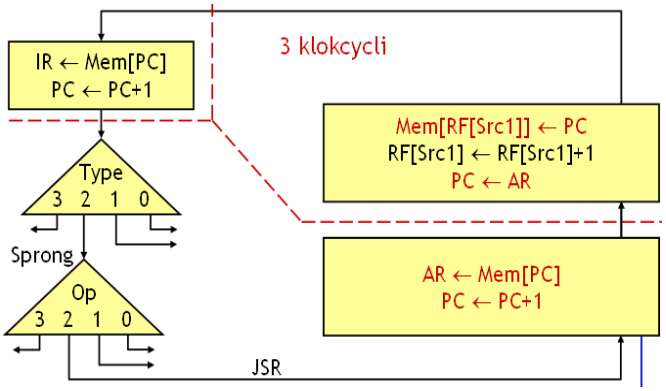
-> Direct laden uit geheugen



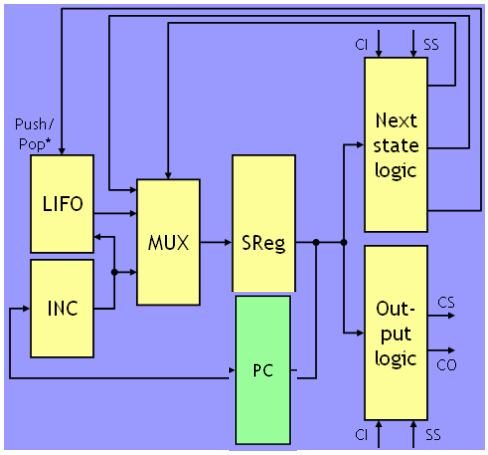
-> Indirect laden



-> Sprong naar subroutine



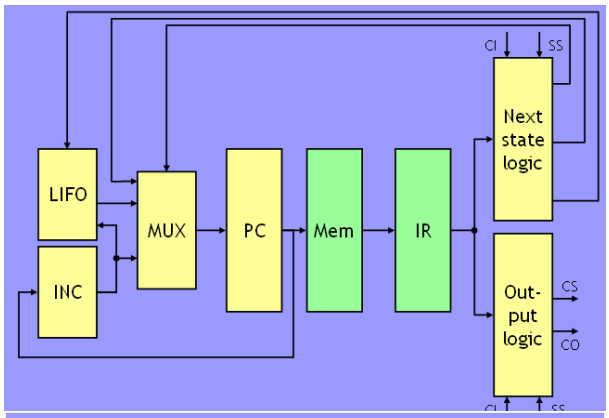
Ontwerp Controller



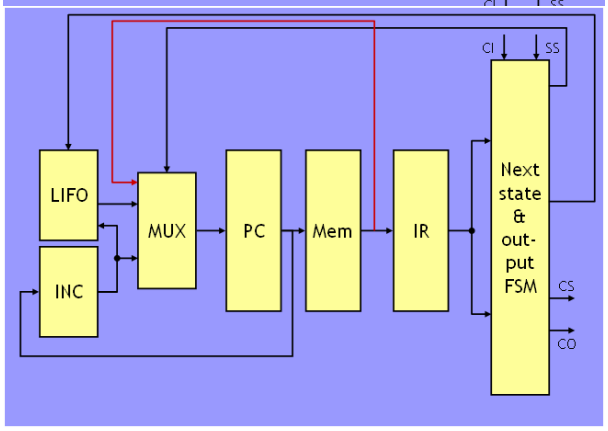
Om een **CISC-controller** te **ontwerpen** zullen we vertrekken van het **basisontwerp** van de controller van **FSMD**. We zullen deze zodanig aanpassen dat we er een **CISC** mee kunnen controlleren :

⇨ Het **Sreg register** dat bijhoudt in **welke toestand** we zijn, vervangen we door de **PC** (**program counter**), de programmateller die **bijhoudt waar we** in het programma **zitten**

⇨ Omdat het **algoritme niet vast** is maar **ingeladen** in een geheugen, is ook de **next-state-logic** en **output-logic** niet meer **vast**. We hebben dus een **bijkomende vertaalstap** nodig die **via programmageheugen** en een **IR** (instructie register) de **next-state** en de **outputs** bepaald.

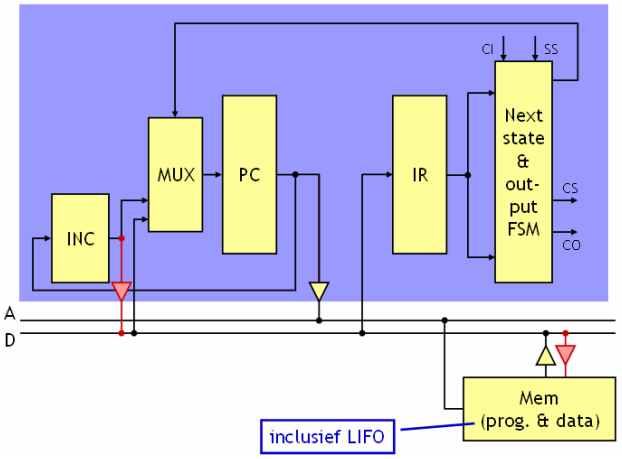


⇨ een **instructie** is een **sequentie** van een **paar kleine (micro-)instructies** waarvan er **1 per klokcyclus** wordt uitegvoerd. De **next-state** & **output** zijn een **FSM** i.p.v. combinatorisch, omdat ze **voor één instructie verschillende uitgangen** en dus **verschillende toestanden** moeten hebben.



⇨ De **sprongadressen** die we soms moeten gebruiken om in het programma te bewegen, komen **niet uit de next-state-FSM** maar uit het **programmageheugen**. Daarom kan deze **output- en next-state-logic** een **complete FSM-controller** zijn met zijn eigen µPC, µIR en µprogramma-ROM

⇨ **Gewoonlijk** wordt het **geheugen** dat tussen de PC en de IR staat gebruikt voor **zowel programma-opslag** als **data-opslag**

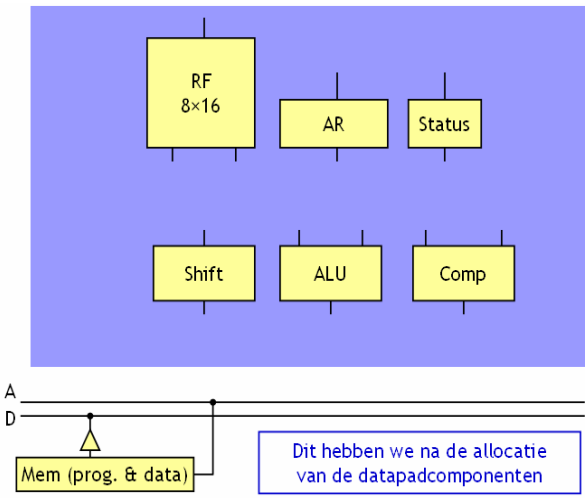


⇨ Meestal wordt de **LIFO** in het **begin** van de controller ook **mee ingebouwd** in het geheugen.

Ontwerp Datapad

De **componenten** van het datapad (= registers & FU's) zijn **reeds vroeger vastgelegd** bij de **allocatie** van de **datapadcomponenten**. We moeten nu enkel nog de **verbindingen** tussen de componenten **voorzien** (inclusief 3-state buffers). Deze zijn **af te leiden uit ASM-schema**.
Als er **conflicten** zijn tussen het **ASM-schema** en de **componenten of verbindingen** dan kunnen we :

- ⇨ **extra componenten** invoeren
  ⇒ herbegin vanaf de allocatie van de componenten
- ⇨ **extra klokcycli** invoeren
  ⇒ herbegin vanaf de bepaling van het ASM-schema

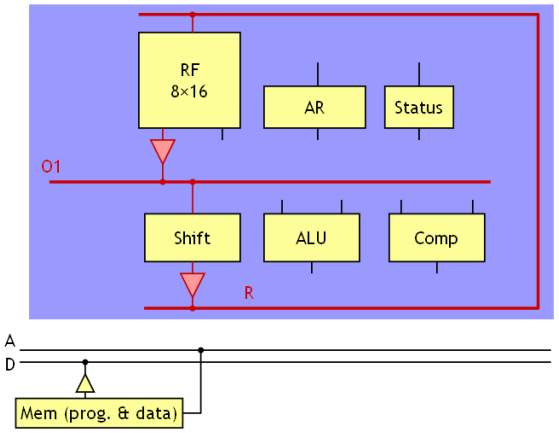


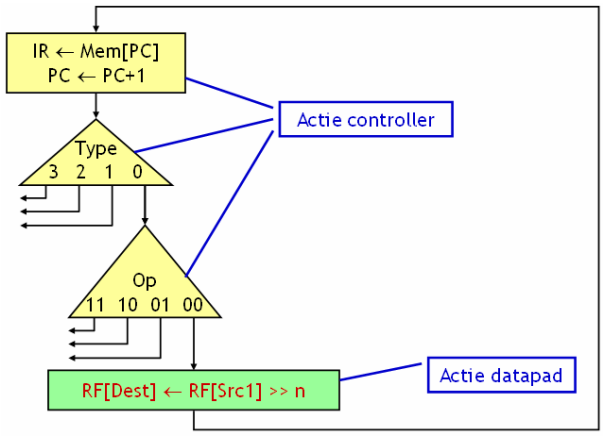
De **componenten** die we al hebben :

- ⇨ **RF** : de registerbank om variabelen tijdelijk te slaan
- ⇨ **AR** : de adres-registers voor indirecte adressen
- ⇨ **Status** : de statusvlag
- ⇨ **Shift** : schuifoperaties
- ⇨ **ALU** : aritmetische logische eenheid
- ⇨ **Comp** : de comparator die waarden vergelijkt
- ⇨ **Mem** : geheugen met programmacode en data

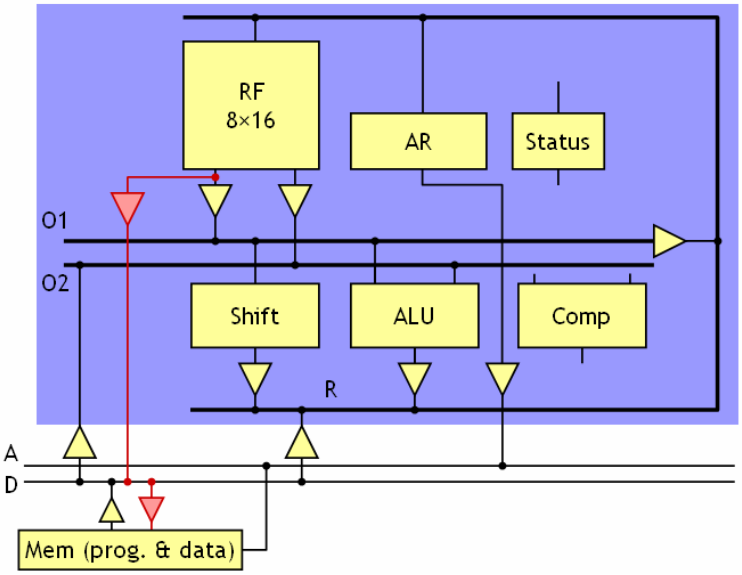
We zullen nu weer voor een **aantal acties** hun **impelementatie** op het **datapad** bespreken. Het enige dat we eigenlijk moeten doen is de **bedrading tussen de bovenstaande vaste componenten leggen** :

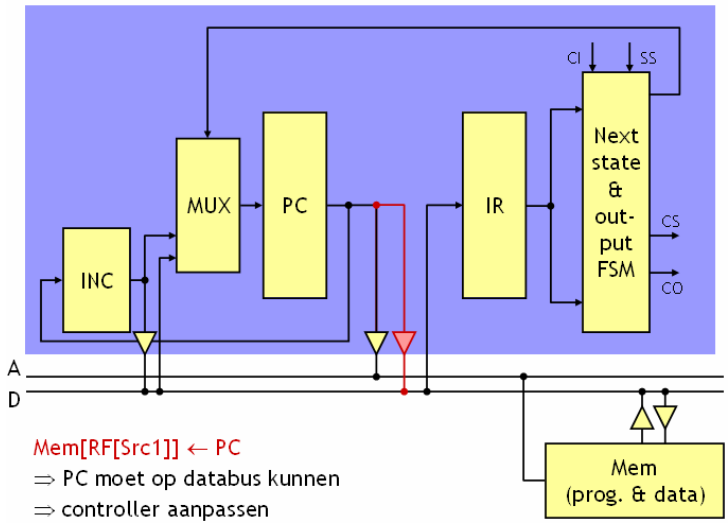
⇨ **verschuiven naar rechts** : deze **actie** wordt **ontleed** door de **controller** vanuit de **IR**. In het **datapad** moeten we dan in de **registerbank** data **ophalen**, dat door de **shift duwen** en dan waar in de **registerbank steken.** Enkel het kadertje met de effectieve bewerking is voor het datapad. Wat we dus moeten **voorzien** is **bus** met **buffer** die de **registerbank** verbind met de **shift**, en van de **shift** terug naar de **registerbank**.





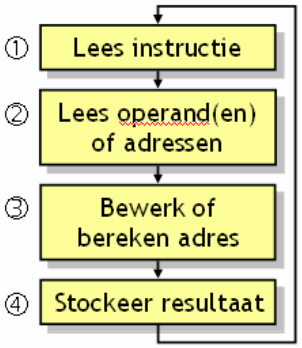
⇨ **Diadische optelling** : buffers van RF naar de 2 operand-bussen en van daar naar de ALU, die dan ook met een bus op de bus terug naar de RF staat (RF=registerbank)
⇨ **Constante laden uit geheugen** : kan met een eenvoudige bufferverbinding van de data-bus van het geheugen naar de bus die naar de RF gaat
⇨ **Direct laden uit geheugen** : wanneer we de geheugenbus A verbinden via de AR naar de resultatenbus voor de RF kunnen we rechtstreeks waarden in het register laden, of resultaten naar het geheugen schrijven.
⇨ **geïndexeerd laden** : kan door de D-bus van het geheugen te verbinden met de 2de operand-bus. We kunnen dan waarden uit de RF combineren met dingen in het geheugen.
⇨ **Register copiëren** : wanneer we de waarde van een registerplaats willen overbrengen naar een andere registerplaats, zorgen we voor een buffer die de 1ste operandbus rechtstreeks met de resultatenbus voor de RF verbind.
⇨ **Direct stockeren** : wanneer we een waarde van het register rechtstreeks willen overschrijven naar het geheugen, kunnen we dit doen door een rechtstreekse verbinding van een uitgang van het register
te verbinden met de data-lijnen van het geheugen.





⇨ **Sprong naar subroutine** : Wannneer we willen springen naar een subroutine, moeten we ook de controller aanpassen omdat de PC (teller) dan op de databus moet kunnen gezet worden omdat we de huidige plaats waar we zitten in ons programma moeten onthouden.

2) RISC-processoren : Reduced Instruction Set Computer



Ongeveer **analoog** aan de **CISC**, maar met **veel meer en veel eenvoudigere instructies** zodat de **kolkfrequentie** veel **hoger** ligt. Het wordt hier belangrijk zo veel mogelijk te **pipelinen** (**instructies parallel** uit te voeren (volgende instructie starten voor de vorige klaar is))

⇨ **Pipelining** van het **uitvoeren** van **instructies** : we zullen proberen om zo **eenvoudig mogelijke** en **altijd ongeveer even lange instructies** te maken
⇨ **Pipelining** van **controller en datapad**, waardoor het **stoppen** van de **pipeline mogelijk** wordt. Dat is nodig, om bv een **sprong** te kunnen maken in het programme, want dan moet **eerst de nieuwe PC berekend** worden voor we kunnen doorgaan met pipelinen.

①	I_1	I_2	I_3	I_4	I_5
②		I_1	I_2	I_3	I_4
③			I_1	I_2	I_3
④				I_1	I_2

Hfdst 7) Hardware-beschrijvingstalen: VHDL

Inleiding

VHDL (**V**HSIC **H**ardware **D**escription **L**anguage, **VHSIC** = **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit) : is een **programmeertaal** om het **gedrag** (ontwerp) van **digitale systemen** te beschrijven. We **beschrijven** volgens een **vaste syntax** wat een bepaalde chip moet doen (zijn **gedrag**) en de **VHDL-compiler** zet dat **ontwerp** dan om naar een **hardware-ontwerp**. We kunnen VHDL **gebruiken** om :

- ⇨ **eenduidige specificaties** voor een ontwerp op gedrags- & RTL-niveau ingeven
- ⇨ **simulatie** van een ontwerp op logische werking en tijdsgedrag
- ⇨ **synthese** (vereenvoudigen) van een **ontwerp** (goed bruikbaar voor RTL-niveau)
- ⇨ **documentatie** van de werking van de chip

De **standaardisatie** van VHDL is **IEEE 1076**. De 1e versie is VHDL-87, tweede VHDL-93 en derde versie VHDL-2001.

VHDL **Analog and Mixed Signal** : is een **uitbreiding** van (zuiver digitale) VHDL die het werken met **analoge** signalen toelaat. We hebben dan ook **continue signalen** in plaats van enkel discrete.

Bv. VHDL-**AMS** (IEEE standaard 1076.1-1999) is een **superset** van VHDL-93 (digitaal ontwerp) die **continue-tijd-modellen** van ontwerpen kan maken en een **set differentiële & algebraïsche vergelijkingen** kan opstellen en oplossen. VHDL-ASM is echter **complex en weinig gebruikt**

Nadelen van **VHDL** t.o.v. het gebruik van **schema's** :

- ⇨ VHDL is **eenvoudig te leren** maar **moeilijk volledig te beheersen**
  - **Conceptueel** is VHDL **verschillend** van elke **software-taal** omdat het om een **hardware beschrijvende taal** is.
  - VHDL schijnt een **moeilijke syntax** te hebben ⇒ daarom gebruiken we meestal **taalgevoelige editoren** met **sjablonen**
- ⇨ Nogal '**langdradige' codes** : er is veel code nodig voor eenvoudige dingen
- ⇨ Een **lijst instructies** is veel **minder overzichtelijk** dan een blokschema voor een mens
- ⇨ VHDL bevat **meer mogelijkheden** dan **strikt noodzakelijk** voor hardware synthese (bijv. specificatie tijdsgedrag voor simulatie)

Voordelen van **VHDL** t.o.v. **schema's** :

- ⇨ **VHDL-code** is algemene **standaart** voor **hardware-beschrijvinge** en is **makkelijk overdraagbaar** over verschillende programma's voor simulatie, synthese, analyse, verificatie, … van verschillende fabrikanten
- ⇨ Het maakt het **gemakkelijker om complexe schakelingen** te beschrijven omdat we kunnen werken op een **hoger abstractieniveau** met **automatische synthese**
  - Je kan bv '**add**' gebruiken zonder een specifiek type van **opteller** te kiezen: het is de taak van het syntheseprogramma om het beste type te kiezen, rekening houdend met randvoorwaarden zoals tijdsgedrag, vermogen en kostprijs
  - VHDL is makkelijk te **parametriseren** (woordlengte, stapeldiepte, …)
  - VHDL is makkelijk om **repetitieve structuren** te beschrijven

Beperkingen van VDHL :

- ⇨ Slechts een **subset** van VHDL kan **automatisch gesynthetiseerd** worden en **elke fabrikant** supporteert een **verschillende subset**. We kunnen dus **meestal niet onze volledige code automatisch laten omzetten** in implementatie.
- ⇨ De **standaard beschrijft** enkel de **syntax** en **betekenis** van zijn code, **niet hoe men de code moet schrijven** (codeerstijl)

  Eenzelfde gedragscomponent (bijv. MUX) kan op heel wat **verschillende manieren beschreven** worden,die ieder tot een **totaal andere implementatie** kunnen leiden (bijv. selector of 3-state bus). De **implementatie** hangt zelfs ook nog af van het **syntheseprogramma**. Je moet al heel wat **ervaring** opdoen alvorens je aanvoelt hoe je de code moet schrijven zodat deze tot de meest efficiënte hardware gesynthetiseerd wordt door een bepaald programma.

Er zijn ook **nog andere 'Hardware Description Languages'** zoals VHDL :

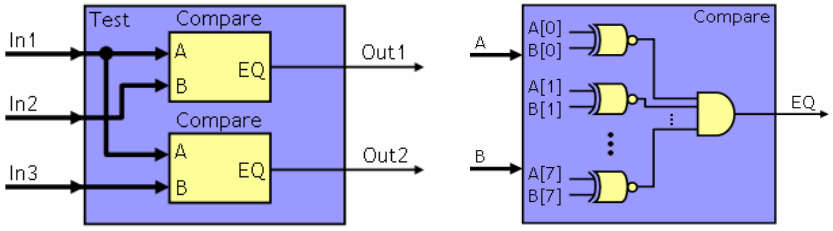
- ⇨ **Verilog** (IEEE 1364) : deze harware-beschrijvende taal is meer verspreid in **USA** dan in Europa. De **syntax** is **verwant met C**, waar VHDL meer verwant is met Ada

  -> **Beter dan VHDL ?** "Both languages are easy to learn and hard to master. And once you have learned one of these languages, you will have no trouble transitioning to the other."
- ⇨ **PLD-talen** zoals ABEL, PALASM, … : gebruikt op **poortniveau** voor een **specifieke technologie**

VHDL in vogelvlucht

Om **VHDL** te **leren kennen** zullen we een **eenvoudig voorbeeldje** bespreken en uitwerken:
Ontwerp een **schakeling 'Test'** met **drie 8-bit ingangen** (In1, In2, In3) en **twee 1-bit uitgangen** (Out1 en Out2) waarvoor geld dat **Out1 = 1 als In1 ≡ In2** en dat **Out2 = 1** als **In1 ≡ In3**.
We kunnen nu beginnen met het **ontwerpen** van **schema's**:

⇨ een schema op **topniveau**, gebruik makend van **componenten** 'Compare'. Dit dient om de **algemene werking** te illusteren. Daarna zullen we de **componenten uitwerken**, bv component compare.



⇨ De **code-declaratie** van de **entiteit comparator** : We zullen nu de '**compare'** definieren als een op **zichzelf bestaande comonent** die we dan **later kunnen gebruiken** in het algemene ontwerp zonder hem nog eens opnieuw te moeten beschrijven. (een soort **zwarte doos** met va**st aantal in- en uitgangen** waarvan de **werking appart beschreven** is) Deze beschrijving specificeert het gedrag op **RTL-niveau**: een **synthese-programma** zal dit omzetten naar **verbindingen** op **poortniveau**

- ⇨ **Entity** : specifiëert dat **compare** een **apparte compentent** is. Een **entiteit** kan meerdere **architecturen** hebben (**verschillende implementaties** van **hetzelfde gedrag**)
- ⇨ De **in- of uitgangssignalen** worden gedeclareerd als '**port**'. Elke 'port' heeft een **expliciete richting** en is een **bit(vector)**

```
-- 8-bit comparator
--
entity Compare is
  port(A,B: in bit_vector(0 to 7);
       EQ: out bit);
end entity Compare;

architecture Behav1 of Compare is
begin
  EQ <= '1' when (A=B) else '0';
end architecture Behav1;
```

'entity' specificeert de interface van de schakeling (zwarte doos in een schema)

'port' specificeert een ingangs- of uitgangssignaal

'architecture' beschrijft het gedrag en/of de structuur van een entiteit (het binnenste van de doos)

⇨ De **code** van het **volledige programma** ('Test'): We kunnen nu **gebruik maken van de component 'compare'** die we daarvoor geïnitialiseerd hebben. Deze architectuur van **'Test'** beschrijft de **structuur van decomponent**, nl. hoe deze **entiteit opgebouwd** is als **verbonden componenten** van **lager niveau**. We laten dus de **werking** van de **lagere-niveau-comonten weg.** We zien ook dat we **twee componenten 'Comparator' tegelijkertijd** kunnen gebruiken!

```
-- Component Test met 2 comparatoren
--
entity Test is
  port(In1,In2,In3: in bit_vector(0 to 7);
       Out1,Out2: out bit);
end entity Test;

architecture Struct1 of Test is
component Comparator is
  port(X,Y: in bit_vector(0 to 7);
       Z: out bit);
end component Comparator;
begin
  Compare1: component Comparator port map (In1,In2,Out1);
  Compare2: component Comparator port map (In1,In3,Out2);
end architecture Struct1;
```

Virtuele component: laat onafhankelijke ontwikkeling van alle hiërarchische niveaus toe. 'Comparator' kan later aan 'Compare' gekoppeld worden

Twee instantiaties van dezelfde component met zijn signaalbindingen

We kunnen ook gebuik maken **virtuele componenten** : deze laten een **onafhankelijke ontwikkeling** van alle **hiërarchische niveau's** toe. We kunnen zo later onze **entiteit compare** aan **comparator koppelen**.

⇨ **Configuratie** : We moeten nu nog **aangeven** welke van de **mogelijke architecturen** van een **entiteit gebruikt** moet worden en we moeten de **componenten aan de entiteiten koppelen**. We moeten met andere woorden voor de **entiteit Compare** één van de **mogelijke architecturen** kiezen en dan de **componente Comparator** koppelen aan **Compare**.

```
-- Configuratie: definieer koppeling component met een
--               bepaalde architectuur van een entiteit
configuration Build1 of Test is
  for Struct1
    for Compare1: Comparator use entity Compare(Behav1)
      port map (A => X, B => Y, EQ => Z);
    end for;
    for Compare2: Comparator use entity Compare(Behav1)
      port map (A => X, B => Y, EQ => Z);
    end for;
  end for;
end configuration Build1;
```

⇨ We proberen **VHDL** te **vergelijken** met een **traditionele programmeertaal** zoals C++ of Java: Een **entiteit** als **Compare** komt overeen met een **klasse** in Java. De **contructor-argumenten** zijn altijd de **ingangen** en de **resultaten** van de methodes zijn de **uitgangen**.
-> In **java** is er **slechts 1 gedragsbeschrijving** per functie (klasse) mogelijk, in VDHL meerdere
-> De **twee 'Compare'-functies** zouden in java **sequentieel** uitgevoerd worden, in VHDL **parallel**

```
// 8-bit comparator
boolean Compare(int A, int B) {
  return (A == B);
}

// Topniveau Test
main() {
  int In1, In2, In3;
  boolean Out1, Out2;

  cin >> In1 >> In2 >> In3;
  Out1 = Compare(In1, In2);
  Out2 = Compare(In1, In3);
  cout << Out1 << Out2;
}
```

Functie-interface: argumenten = ingangen resultaat = uitgang

Gedragsbeschrijving van de functie

2 oproepen van de functie 'Compare' met de gekoppelde argumenten

-> de '**main**' wordt **éénmaal uitgevoerd** en **stopt** dan in java, **hardware blijft altijd actief** (stopt nooit)
Verdere verschillen met traditionele talen :

- ⇨ **Datatypes**: in VHDL hebben we **nood** aan **typische hardware-data-types**: bitvectoren, getallen met een arbitraire grootte, getallen met vaste komma. In java hebben we strings enzo.
- ⇨ **Gelijktijdigheid** ('concurrency'): in VHDL werken **alle hardwarecomponenten in parallel**
- ⇨ **Tijdsconcept**: Alle componenten **werken continu**: hardware stopt nooit!
  En voor **simulatie** is een **koppeling** met het **reële tijdsgedrag** van componenten nodig : waar in computerprogramma's **alle regels code mooi achtereen** uitgevoerd worden wanneer de vorige klaar is, is er in hardware alleen een **hoop onsamenhangende delen**.

Elementen van de VHDL-taal

1) Lexicale elementen (woordenschat)

Wanneer we een VHDL-code schrijven doen we dat met een specifieke syntax en woordenschat. We bespreken kort de verschillende lexicale elementen :

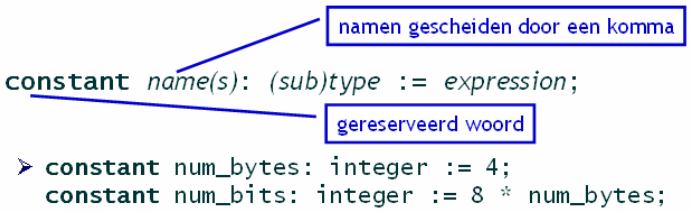
- ⇨ **Commentaar**: wanneer we in de code **eigen commentaar** willen schrijven zonder dat het wordt uitgevoert (**programma negeert die lijnen** dan), schrijven we voor de tekst **'--'**
- ⇨ De '**Identifier'** of **naam** van een **variabele of een instantie** wordt geschreven als een reeks van **alphanumerische karakters of niet-opeenvolgende '_'**, die start met een letter & niet eindigt met '_'. Bv. "Next_value_0". De VDHL-code is **niet Case-sensitive** : geen verschil hoofdletters en kleine letters
- ⇨ Een **getal** declareren : een getal kan een '**integer literal'**, dus een **geheel getal** zijn bv"1480" of een **'real literal'**, dus een **fractioneel getal** zijn bv "1480.0".
  Beide kunnen **exponentieel** worden gemaakt bv"148E1"
  Voor het **leesgemak** kan men **'_'** in de getallen schrijven : deze worden **genegeerd** bv "1_480"
  Wanneer we een getal in een **ander talstelsel** dan het 10-tallige willen gebruiken, kunnen we dat doen via volgense syntax : **base#literal#exp** bv 253.5 = 16#FD.8# / 2#1#E10 = 16#4#E2
- ⇨ Een **karakter** ('character literal') wordt **genoteert** tussen enkele **aanhalingstekens** bv 'a' 'A'
- ⇨ Een **reeks karakters** ('string') wordt genoteerd tussen **dubbele aanhalingstekens** bv "A string". Om een Quote"teken in een string te krijgen schrijven we dubbele aanhalingstekens bv """Quote it"", she said."
- ⇨ Een **bitreeks** ('bit string') is een **reeks bits**, voorgesteld door een **reeks cijfers** voorafgegaan door een **basisspecificatie** ('B', 'O', of 'X') bv O"12" = b"001_010" ⇔ X"a" = B"1010"

2) Data-objecten en Data-types

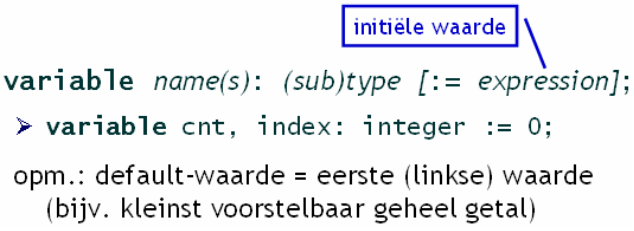
VDHL-objecten

Een VHDL-object is een variabele die een naam heeft en een waarde van een specifiek type

- ⇨ **Constanten** : deze maken het **programma meer verstaanbaar**
  Declaratie : we **definieren** eerst dat het constanten zijn met het commando **'constant'**, daarachter komt de **naam**, dan het **type** en dan de **waarde** van de constante :



- ⇨ **Variabelen** : deze bevatten de **tussenresultaten** van de bewerkingen **zonder fysische betekenis**.
  Er is hier **geen verband** met een **ASM-variabele**!
  Declaratie : we beginnen met '**variable'**, daarachter de **naam**, het **type** en eventueel een **initiële waarde**.
  -> Wanneer we een **waarde** willen **toekennen** aan de variabele, doen we dit door **' naam := expressie '** met **naam** de naam van de **variabele** en **expressie** een uitdrukking met de **nieuwe waarde**.



⇨ **Signalen** : deze zijn **draden**, (interne) **verbinding**, of **golfvormen**, zoals zichtbaar tijdens een **simulatie**

```
signal name(s): (sub)type [:= expression];
➢ signal a, b: bit;
```

Declaratie : vooraan '**signal'**, dan **naam**, **type** en eventueel **initiële waarde**
-> Wanneer we een **golfvorm** willen **toekennen** aan een signaal doen we dat met de uitdrukking **' naam <= golfvorm '**. Een **golfvorm** kan een **eenvoudige logische uitdrukking** zijn bv `y <= a and b;` , of een **complexe golfvorm** voor tijdens de simulatie (zie tekst hiernaast).

```
[delay_mechanism] expression [after time]
 [, expression [after time]]...
▪ line_out <= transport line_in after 100 ps;
  pulse <= '1', '0' after T_pw;
```

⇨ (Bestanden)

VDHL-types

Een VHDL-type is een soort van waarde, zoals een enkele bit, een scalair getal, een vector bits, …

⇨ **VHDL-types**
- Scalaire types : set waarden
- Samengestelde types : verzameling sets
- 'Access' types : 'pointers' voor gelinkte lijsten
- Bestandstypes

⇨ **Declaratie type** : **type** name **is** type_definition bv. **type** int_8 **is range** -128 **to** 127;

⇨ **Declaratie subtype** : beperkte set waarden van basistype : **subtype** name **is** scalar_(sub)type [**range** expression (**down**)**to** expression]; bv. **subtype** nat_8 **is** int_8 **range** 0 **to** 127;

Scalaire types

Declaratie van **scalaire types** variabelen :

⇨ **Discrete types** : variabelen **zonder komma**, met **gehele begrensde waarde**.
-> ***'Integer' types*** : de **gewone gehele getallen**. Wanneer we **type integer** definieren, bepalen we eigenlijk de **range van getallen** die we willen kunnen gebruiken, (alle getallen die **voorstelbaar moeten zijn**) omdat het programma aan de hand daarvan een **vast aantal geheugenplaatsen** moet voorzien voor de variabele : (hoe groter de range, hoe meer geheugenplaatsen en adressen voorzien moeten worden). De variabele heeft op elke ogenblik **één waarde** uit de mogelijke **voorgedefiniëerde range**.

range *integer_expression* (**down**)**to** *integer_expression*
type mem_address **is range** 65535 **downto** 0;

⇨ **voorgedefiniëerde Integer types** :
- **type** integer **is range** *implementation_defined*; :
  met een bereik van minstens –231+1 tot +231–1
- **subtype** natural **is** integer met **range** 0 **to** integer'high;
- **subtype** positive **is** integer met **range** 1 **to** integer'high;

-> '***Enumeration' types*** ; Bij dit type variabele bepalen we de **range** zelf als een **eindige verzameling elementen.** Bij het aanmaken van het type **sommen we alle mogelijke waarden** van het type op, en elk object van dat type heeft dan **op elk ogenblik één** van de **vooropgesomde waarden**. Hierbij heeft het **type een naam**, en **alle mogelijke waarden van het type** hebben ook elk hun **naam** : **(*name_or_charliteral [, name_or_charliteral]…*)** (een **integer-type** heeft ook telkens één vaste waarde uit een vaste verzameling, maar het **verschil** is dat daar die verzameling impliciet met zijn grenzen bepaald is i.p.v.expliciet) bv

type FSM_state **is** (reset, wait, input, calculate, output);
type tri_val **is** ('0', '1', 'Z'); *(mogelijke waarden zijn 0, 1 en Z)*

⇨ **voorgedefiniëerde Enumerition types** :
- **type** bit **is** ('0','1');
- **type** boolean **is** (false, true);
- **type** character **is** (*ASCII set*); dit zijn namen voor niet-afdrukbare controlekarakters zoals bv. nul, cr, lf, esc, del
- **type** severity_level **is** (note, warning, error, failure);

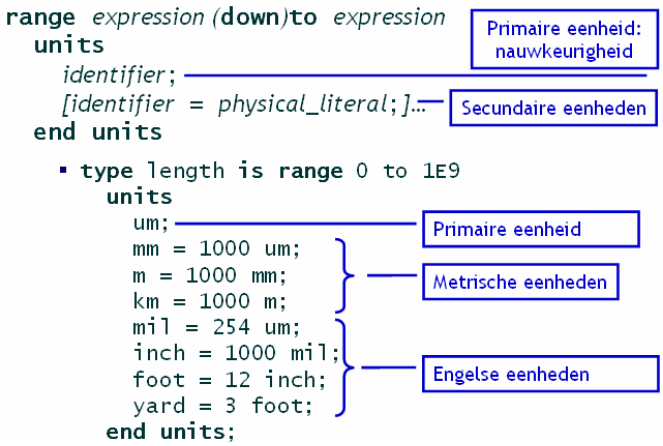
⇨ **Komma-types** : **variabelen met een komma**, **niet gehele getallen**. Hierbij moeten we niet alleen definieren van welke tot welke waarde we willen kunnen toekennen, maar ook in **hoever we decimaal willen gaan** (het **aantal plaatsen na de komma** dat voorzien moet worden in het geheugen) :

range *real_expression* (**down**)**to** *real_expression*
type probability **is range** 0.0 **to** 1.0;

⇨ **voorgedefiniëerde Komma-types** :
- **type** real **is range** *implementation_defined*;
  bereik van minstens IEEE 32-bit enkelvoudige precisie

⇨ **Fysische types** : **analoog** aan het vorige, een **variabele of constante** met een bepaalde **range** (ook na de komma soms), die een bepaalde **fysische betekenis** heeft :

⇨ **voorgedefiniëerde Fysische-types** :
- **type** time **is range** *implementation_defined*
  units
  fs;
  ps = 1000 fs;
  ns = 1000 ps;
  us = 1000 ns;
  ms = 1000 us;
  sec = 1000 ms;
  min = 60 sec;
  hr = 60 min;
  end units;



⇨ **de IEEE 1164 'Standard logic'** : voor logische signalen hebben we in praktijk meer dan '0' en '1' nodig, daarom definieert IEEE standaard 1164 logische signalen met 9 mogelijke waarden. Gebruik altijd deze i.p.v. 'bit' voor echte toepassingen!

```
library IEEE; use IEEE.Std_logic_1164.all;
    type std_ulogic is (
      'U',   -- niet geïnitialiseerd, bijv. bij opstarten
      'X',   -- sterk aangestuurd ongekend, bv. na schending set-up
      '0',   -- sterk aangestuurd logisch 0
      '1',   -- sterk aangestuurd logisch 1
      'Z',   -- hoog-impedant, dus m.a.w. niet aangestuurd
      'W',   -- zwak aangestuurd ongekend
      'L',   -- zwak aangestuurd logisch 0
      'H',   -- zwak aangestuurd logisch 1
      '-'    -- don't care                                    );
    type std_logic is resolved std_ulogic;
  subtype X01 is resolved std_ulogic range 'X' to '1';
```

Samengestelde types

'Array' types : Samengestelde types zijn **verzamelingen van variabelen** (vectoren & matrices)':

⇨ **Begrensd** ('constrained'): **begrensde vaste omvang** met dus een vast aantal indices. We hebben hier een **naam voor de array** en indices voor de verschillende elementen. Elk element van de array is van **hetzelfde (sub)type**.

Array *(range [, range]…)* **of** *(sub)type*

waarbij *[range,]* ofwel een ***discreet subtype*** is ofwel een ***expression*** (**down**)**to** ***expression***

```
type word is array (15 downto 0) of bit;
type next_state is array (FSM_state, bit) of FSM_state;
        (bit zijn hier getallen van 15 tot 0 om de plaatsen in next_state aan te
        duiden. 15 komt overeen met de MSB (meerst beduidende bit), 0 met LSB)
  bv. variable next: next_state; next(calculate, '1') := output;
```

⇨ **Onbegrensd** : **grenzen** van array zijn **onbepaald**, we kunnen **onbegrenst elementen toevoegen**

Array (*(sub)type* **range** <>*[*, *(sub)type* **range** <>*]…*) **of** *(sub)type*

```
type sample is array (natural range <>) of integer;
subtype buf_type is sample(0 to 255);
   bv. variable sample_buf: sample(0 to 63);
```

⇨ **voorgedefiniëerde onbegrensde types** :

-> **Onbegrensde matrices**

```
- type string is array (positive range <>) of character;
     bv   constant Error_message: string := "Unknown error: ask for help";
- type bit_vector is  array (natural range <>) of bit;
     bv   constant State1: bit_vector(4 downto 0)  := "00100";
```

-> **Onbegrensde matrices** in **IEEE 1164**

```
- type std_[u]logic_vector is array (natural range <>) of std_[u]logic;
```

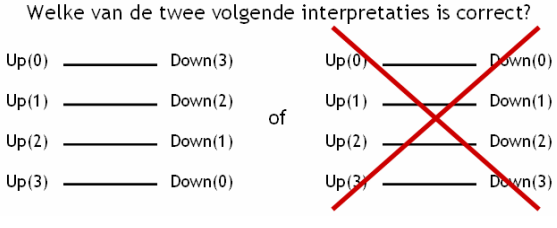
⇨ **Matrices** kunnen aan **elkaar toegekend** worden als ze **dezelfde dimensies** en **grootte** hebben.

De **correspondentie** gebeurt via **positie**, **niet via index**!

bv. zie voorbeeld Up(0) en Down(3) zijn beide de laatste plaats.

```
signal Down: std_logic_vector (3 downto 0);
signal Up: std_logic_vector (0 to 3);
Up <= Down;
```

Welke van de twee volgende interpretaties is correct?



⇨ **Array literal** : Wanneer we een **array niet als een vector verschillende waarden** beschouwen : bv

⇨ (**Bit)string literal** : **literaire variabalen** zijn variabelen die **tekst** voorstellen in plaats van numerieke waarden. We kunnen zo bv met een **reeks bits een woord voorstellen**,

variable w: word := "1010000101111111";
variable w: word := x"A17F";

⇨ **Matrixgeheel** ('array aggregate') : We associëren de waarden in de Array met een bepaald ding

-> **Associatie** van de array met een **positie in een ruimte**. De 3 waarden in de array zijn dan de **positie-coördinaten** van het punt dat de array voorstelt : (*expression [, expression]…*)

type point **is array** (1 **to** 3) **of** integer;
variable p: point := (4, 5, 5);

-> **Associatie** van de array met een **naam** (*choice [| choice]… => expression [, choice [| choice]… => expression]…*)

met ***choice*** ofwel een **uitdrukking**, ofwel een **discreet bereik**, ofwel **others**

variable p: point := (3 => 5, 1 => 4, 2 => 5);
variable p: point := (1 => 4, 2 | 3 => 5);
variable p: point := (1 => 4, 2 **to** 3 => 5);
variable p: point := (1 => 4, **others** => 5);
sample_buf := (**others** => 0);

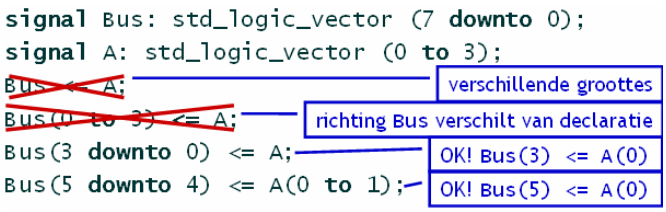
⇨ Een **matrixdeel ('slice')** is een **subset** van **opeenvolgende matrixelementen**. We nemen dus een **aantal opeenvolgende elementen** uit een **bepaalde matrix** en definiëren dat als **slice**. We moeten er dan wel voor zorgen dat de **richting** (**to** of **downto**) **dezelfde** is in de matrix als in de slice !

⇨ we kunnen zo een **matrix begrenzen** met een **subtype-definitie**

subtype halfword **is** bit_vector(0 **to** 15);

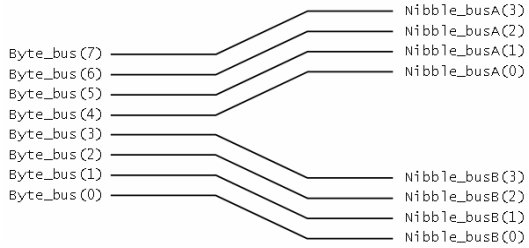
⇨ we kunnen zo **waarden toekennen** aan een **stuk** van een matrix, **zonder de rest te wijzigen**. Dit kan dus alleen als ze **dezelfde grote** hebben en **dezelfde richting** van **indexering** hebben.

signal Bus: std_logic_vector (7 **downto** 0);
signal A: std_logic_vector (0 **to** 3);
Bus <= A;
Bus(0 **to** 3) **<=** A;
Bus(3 **downto** 0) <= A;
Bus(5 **downto** 4) <= A(0 **to** 1);



⇨ **Concatenatie van (ééndimensionale) matrices** : is het samenvoegen van matrices door een aantal draden te bundelen

signal Byte_bus: bit_vector(7 **downto** 0);
signal Nibble_busA, Nibble_busB: bit_vector(3 **downto** 0);
Byte_bus **<=** Nibble_busA **&** Nibble_busB;



Attributen

⇨ Een **attribuut** is **informatie** over een **object** of een **types**. Voorbeelden van **voorgedefinieerde** attributen :

- ⇨ van **scalaire** types
  - T'left: eerste waarde van T
  - T'low : kleinste waarde van T
  - T'pos(*x*) : positie van *x* in T
- ⇨ van **matrixtypes & -objecten**
  - A'range*[(n)]* : indexbereik van dimensie *n*
  - A'length*[(n)]* : grootte van indexbereik
  - A'left*[(n)]* : eerste waarde in indexbereik
- ⇨ van **signalen**
  - S'event : true als er een 'event' was op S in de huidige simulatiecyclus
  - S'last_event : tijd sinds laatste 'event' op S

⇨ Een **atrribuut** toegevoegd door de **gebruiker** is het toevoegen van **beperkingen** en **informatie buiten** de **structuur** en het **gedrag** van de schakeling

- ⇨ **Declaratie** van een attribuut :
  attribute *name* : *sub(type)*;
  attribute pin_number : positive;
  attribute encoding : bit_vector;
- ⇨ **Specificatie** van een attribuut
  attribute *name* of *name(s)* : *class* is *expression*;
  attribute pin_number **of** EN_1, EN_2: **signal is** 14;
  attribute encoding **of** state1: **literal is** b"0000";

3) Bewerkingen: logisch, relationeel, aritmetisch & schuiven

⇨ De **logische bewerkingen** die gebruikt kunnen worden zijn: **not**, **and**, **or**, **xor**, **nand**, **nor, (xnor)**
De **Prioriteit** van de bewerkingen :
'**not**' heeft de hoogste prioriteit, alle andere hebben gelijke prioriteit lager dan '**not**'
De **logische bewerkingen** zijn **enkel gedefinieerd** voor de **datatypes**:
bit*[_vector]*, boolean, en de std_*[*u*]*logic*[_vector]*
De logische bewerkingen kunnen **toegepast** worden op **matrices** van **dezelfde grootte**:
de bewerkingen gebeuren dan telkens op elementen met overeenkomende posities.
Het **resultaat** van een **logische operatie** is een **boolean** (of een matrix booleans)

⇨ De **vergelijkingen** (relationele operatoren) die **gebruikt kunnen worden** zijn : **<, <=, =>, >, =, /=**
We kunnen **alleen operanden** vergelijken als ze **hetzelfde type** hebben
Het **resultaat** is altijd een **boolean**
Een **vergelijking** kan **uitgevoerd** worden op **matrices**, **zelfs van verschillende grootte**
->Dit werkt dan als volgt : we **aligneren** de matrices op hen **linkerelementen** en we **vergelijken element** per **element**, van **links naar rechts**. We **vergelijken maximaal** zoveel **elementen** als er in de **kleinste matrix** aanwezig zijn.
-> Daarom zijn de **volgende vergelijkingen waar:** "1110" > "10111" en "1110" = "11101"
-> Dit werkt dus op bitvectoren van **gelijke lengte** alsof het **positieve getallen** waren

⇨ De aritmetische bewerkingen die we kunnen toepassen zijn :

+, –, *, /, ** (exponent), **abs** (absolute waarde), **mod** (modulus), **rem** (rest)

De bewerkingen zijn **enkel gedefinieerd** voor de datatypes **integer** en **real** (behalve mod en rem), maar **niet op bitvectoren**. Voor bit(vectoren) gebruiken we 'overloading' (niet standaard!). Voor **fysische datatypes** zijn enkel +en – gedefiniëerd.

De operanden moeten van **hetzelfde type** zijn, maar een **verschillend bereik** is **toegelaten**

Een variabele van het **fysische type** (bijv. time) kan ook **vermenigvuldigd** worden met (of gedeeld worden door) een **integer of een real**. Het resultaat blijft echter een fysische type.

⇨ De **schuifoperaties** die toegepast kunnen worden (niet in VHDL-87):

sll ('shift-left logical'), **srl**, **sla** ('shift-left arithmetic'), **sra**, **rol** ('rotate left'), **ror**

De **eerste operand** is een **vector** van **bits** of van **booleans**

De **tweede operand** is een **integer**. Als deze negatief is, schuiven we in de tegengestelde richting

Het **resultaat** is van **hetzelfde type** als de **eerste operand**.

Voorbeelden: B"10001010" sll 3 = B"01010000"
B"10001010" sll -2 = B"00100010"
B"10001010" sra 3 = B"11110001"
B"10001010" ror 3 = B"01010001"

4) Controle-uitdrukkingen: conditionele uitdrukkingen & lussen

Conditionele uitdrukkingen zijn uitdrukkingen waar we een **bepaalde expressie evalueren** en op **basis** van de **uitkomst** bepalen wat er **als volgende stap** gebeurt.

⇨ Uitdrukking **" if "** :

met het **if-statement** zullen we een **expressie evalueren** die als **resultaat** een **boolean** geeft. Is dit resultaat **waar** dan zullen we de **statements uitvoeren** die **na de if** komen. Wanneer het **onwaar** is zullen we de **statements na het else-statement** uitvoeren.

Ingebouwde prioriteit: de eerste voorwaarde die waar is bepaalt welke *statement(s)* uitgevoerd worden.

Structuur : **if** *boolean_expression* **then**
statement(s)
[**elsif** *boolean_expression* **then**
statement(s)]…
[**else**
statement(s)]
end if;

⇨ Uitdrukking **" case"** :

met het case-statement zullen we een expressie evalueren die meerdere verschillende oplossingen kan hebben. We zullen alle mogelijke oplossingen overlopen en telkens statements voorzien om uit te voeren wanneer die oplossing zich voordoen. Na 'case' komt telkens de te evalueren uitdrukking, alle mogelijke oplossingen worden ingeleid met 'when' :

case *expression* **is**
when *choice(s)* **=>** *statement(s)*
*[***when** *choice(s)* => *statement(s)]…*
end case;

De **voorwaarden** voor het **correct gebruik** van 'case' :

-> **Alle mogelijke oplossing** van de expressie moeten **exact eenmaal gespecificeerd** worden
-> **Alle opgesomde oplossingen** moeten van **hetzelfde type** zijn als de **uitkomst** van expression
-> **Alle waarden** blijven **constant** en zijn **gekend** op het moment van ontwerp

Voorbeeld :

- **case** x **is**
  when 0 **to** 4 **=>** y <= 'Z';
  when 5 => y <= '1';
  when 7 | 9 **=>** y <= '0';
  when others => **null**;
  end case;

<u>lussen</u>

⇨ de **Oneindige lus**

een oneindige lus specifiëren we met 'loop' in het begin en met 'end loop' op het einde.

loop
statement(s)
end loop;

Oneindige lussen zijn typisch voor hardware omdat hardware nooit stopt, er is altijd een volgende toestand. We bespreken het voorbeeld van een 4-bit teller met uitgang "count" :

```
val := 0;
loop
  count <= val;
  wait until clk = '1' or reset = '1';
  exit when reset = '1';
  val := (val + 1) mod 16;
end loop;
```

wacht tot clk '1' wordt

We kunnen de **oneindige lus onderbreken** met een **"exit"-uitdrukking**. Achter 'exit' komt een uitdrukking met **booleaanse oplossing**. Wanneer die **waar** is verlaten we de lus.

exit *[***when** *boolean_expression]*;

We kunnen **overgaan naar een andere oneindige lus** met de "**next**"-**uitdrukking**. Ook achter 'next' komt een **booleaanse uitdrukken**. Wanneer die **waar** is gaan we naar next.

next *[***when** *boolean_expression]*;

⇨ de **" while " lus**

is een voorwaardelijke lus. We blijven de lus uitvoeren tot de booleaanse uitdrukking die na 'while' komt, niet meer waar is.

while *boolean_expression* **loop**
statement(s)
end loop;

⇨ de **“ for ”-lus**

we **specifiëren** hier een **variabele** met een **bepaalde naam** en een **bepaalde range**. We zullen de lus nu **telkens uitvoeren** met een **andere waarde** van ‘name’ in de **vooropgestelde range**. We beginnen met de **eerste waarde** in range en **stoppen de lus** wanneer we de **laatste gehad** hebben.
De lusvariabele ‘name’ moet moet niet expliciet gedeclareerd worden en kan alleen binnen de lus gebruikt worden.

for *name* **in** *range* **loop**
statement(s)
end loop;

Enkele voorbeelden van *range* :

for i **in** 0 **to** 3 **loop** … / **for** i **in** an_array'range **loop** … / **for** state **in** FSM_state **loop …**

5) Subprogramma’s: procedures & functies

Een **subprogrammas** is een **verzameling** van **sequentiële uitdrukkingen** die (een aantal keer) uitgevoerd kunnen worden als onderdeel van een groter programma en die appart gedeclareerd zijn.

subprogram_specification **is**
[constant, variable, (sub)type declaration]…
[subprogram]…
begin
statement(s)
end;

Mogelijke *subprogram_specification*:

⇨ **Procedure**: equivalent met een entity : voert gewoon een aantal regels code uit
(entity als onderdeel van een hoofd-entity)
procedure *name [*(*interface_list*)*]*
waarbij *interface_list* een lijst is van *[***signal***] param_name(s)* : *[mode] (sub)type. mode* is, gescheiden door ‘;’, en waarbij mode één van **in**, **out**, **inout** of **buffer** is

⇨ **Functie**: (onderdeel van) een expression : berekent effectief iets en geeft waarde(s) terug
function *name [*(*interface_list*)*]* **return** *(sub)type*
waarbij minstens één van de statement(s) een **return** expression is.

We **gebruik** subprogramma’s via **associatie** met hun **declaratie** :

⇨ **Associatie** van **argumenten** volgens **positie** : *name*(*expression [*, *expression]…*)
We mappen hier de in- en outs van de instantie van de component op die van de declaratie door ze in dezelfde volgorde door te geven als in de declaratie staat

⇨ **Associatie** van **argumenten** volgens **naam** : In plaats van de ins en outs in de volgorde van de declaratie door te geven, zullen gewoon zeggen welke param we waarop mappen
name (*param_name* => *expression [*, *param_name* => *expression]…*)

Voorbeeld van een functie :

```
function or_bv ( bv : in bit_vector )
        return bit is   variable result : bit := '0';
begin
  for index in bv'range loop
    result := result or bv(index);
  end loop;   return result;
end;
```

⇨ **Overloading** : is het **toelaten** van **verschillende subprogramma's** met **dezelfde naam** maar met een **verschillende *interface_list***. Ze hebben dus dezelfde naam maar **doen verschillende dingen**. De **context** bepaalt dan **welke interface-list** gebruikt wordt. We kunnen bijvoorbeeld een functie 2 maal definiëren met verschillende types van ingangen. Wanneer we dan de functie oproepen op bepaalde ingangen, hangt het dus van hun type af welke interface wordt uitgevoerd.

procedure incr(a : **inout** integer) is …
procedure incr(a : **inout** bit_vector) is …

Om operatoren te '**overloaden' plaatsen** we ze tussen **aanhalingstekens**

function "+" (a,b: **in** bit_vector) : optellen van 2 integers **return** bit_vector **is** …
function "+" (a: **in** bit_vector, b: **in** integer) : optellen van een integer en een bitvector
return bit_vector **is** …

6) Bibliotheken

Evolutionair ontwerpen: dikwijls kan tot **95% van een ontwerp hergebruikt** worden om en volgend ontwerp te maken. Het is dus **nuttig** om **stukken ontwerp** op te slaan om ze gewoon **over te kunnen nemen** in een **nieuw ontwerp**.
⇨ Een '**Package'** groepeert **definities** van constanten, componentdeclaraties, datatypes en subprogramma's
⇨ Een '**Library'** is de **plaats** waar de **binaire code** van **analyse/compilatie gestockeerd** wordt (folder, databank, ...). Default noemt deze work (dat is dan de huidige werkfolder).

De **VHDL**-bibliotheek

- ⇨ **Declaratie**: interface. Voor de declaratie van variabelen kunnen we gebruik maken van een aantal packages waarin reeds een aantal types van variabelen zijn gedefiniëert.
  package *name* **is**
  [constant, signal, component, (sub)type,
  attribute, subprogram declaration]…
  end *[***package***] [name]*;
- ⇨ '**body': subprogramma's** : wanneer we de body van ons programma schrijven moeten kunnen we bepaalde specifieke taken laten uitvoeren door reeds vooraf geprogrammeerde code in een package.
  package body *name* **is**
  [constant, (sub)type declaration]…
  [subprogram]…
  end *[***package body***] [name]*;
- ⇨ **Gebruik**
  library *library_name*;
  use *library_name.package_name*.**all**;
  bv **library** ieee; **use** ieee.std_logic_1164.**all**;

Standaart IEEE-bibliotheken :

- ⇨ std_logic_1164 : bewerkingen met std_*[*u*]*logic*[*_vector*]*
- ⇨ numeric_bit : bewerkingen met *[*un*]*signed : **type** *[*un*]*signed **is array** (natural **range <>**) **of** bit;
- ⇨ numeric_std : idem maar op std_logic i.p.v. bit
- ⇨ math_real : bewerkingen op real
- ⇨ math_complex : bewerkingen op complexe getallen

Hardware-beschrijving met VHDL

1) Gedragsbeschrijving van componenten

VHDL **modulebeschrijving** :

⇨ Interface van een module is de **declaratie** van een **entiteit**. We geven de entiteit een **naam**, een **lijst** van **generische parameters** (expressies) *name(s)* : *(sub)type [:= expression]* die verschillend zijn voor alle instanties die van de entiteit gemaakt worden en een **lijst** met **poorten** voor bepaalde signalen *signal_name(s)* : *[mode] (sub)type* die de interface vormen naar de entiteit en afwezig zijn op het hoogste niveau.
In de lijsten worden de elementen gescheiden met een ';'.

entity *name* **is**
*[***generic** (*generic_list*);*]*
*[***port** (*port_list*);*]*
end *[***entity***] [name]*;

Voorbeeld :

```
entity reg is
  generic (n : positive;
        T_pd : time := 5 ns);
  port (D : in bit_vector(0 to n-1);
      Q : out bit_vector(0 to n-1);
      clk : in bit);
end entity reg;
```

-> Generische constanten laten toe om zowel het **gedrag** als de **grootte** van **verbindingen** te **parametriseren** voor de entiteit. Daardoor wordt het **hergebruiken** van **entiteiten** in **verschillende omstandigheden** mogelijk. Hierdoor is **VHDL krachtiger** dan **schema's**. De waarden van de generische parameters moet **gekend** zijn op ogenblik van de **synthese**!

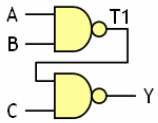
⇨ Implementatie van een module is de **declaratie** van **één of meerdere architecturen**, die **alternatieve implementaties** beschrijven van de module

architecture *name* **of** *entity_name* **is**
[constant, variable, signal declaration]…
[(sub)type, attribute declaration]…
[component declaration]…
[subprogram]…
begin
{[label :] concurrent_statement}…
end *[***architecture***] [name]*;

[label] : we benoemen de uitdrukkingen en componenten.
Dit is nuttig voor debugging, simulatie & configuratie.
We zullen zien dat alle alle concurrent_statements parallel worden uitgevoerd en elk bestaan uit een aantal sequentiële uitdrikkingen. De labels van de concurrent_statements komen overeen met die van de processes (zie verder).

⇨ **Paralelle uitdrukkingen** : de **uitdrukkingen** (*concurrent_statement*s) worden **allemaal tegelijkertijd uitgevoerd**. Dit is zeer **normaal** van **hardware**. De **volgorde** van parallelle uitdrukkingen is **onbelangrijk** aangezien ze toch allemaal gelijkertijd uitgevoerd worden.

```
entity Concurrent is
  port (A,B,C: in std_logic;
        Y: out std_logic);
end entity Concurrent;
architecture Struct of Concurrent is
  signal T1: std_logic;
begin
  NAND2: entity NAND2 port map (T1,C,Y);
  NAND1: entity NAND2 port map (A,B,T1);
end architecture Struct;
```



⇨ De **basis** van **parallelle uitdrukkingen** wordt gevormt door een "**process**". Dit is een **programma** van **sequentiële uitdrukkingen** die samen **één parallelle uitdrukking** vormen. Ze komen voor in de architecture van een component worden parallel uitgevoerd met andere processes. Elk proces heeft **label**, dat overeenkomt met het label van zijn uitrukking in algemene module-architectuur.

process *[**is**]*
  [constant, variable, (sub)type declaration]…
  [subprogram]…
begin
  sequential_statement(s)
end process *[label]*;

Het **process herhaalt** zijn ***sequential_statement(s)*** **eindeloos** zoals een **oneindige lus**. Daarom moet er **minstens één sequentiële uitdrukking 'wait'** voorkomen in het process.

```
clock_gen: process is
  variable val: std_logic := '0';
begin
  clk <= val;
  val := not val;
  wait for T_pw;
end process clock_gen;
```

globaal signaal
locale variabele
globale constante

Deze **sequentiële "wait"bepaal** de **reactie** van het proces op **signalen** van het globale programma

⇨ **wait** *[**on** signal_name(s)] :* hier is het process **gevoelig voor signalen**:
het proces **hervat** als **één van de *signal_name(s)* verandert** van waarde
*[**until** boolean_expression]* : hier is het programma **gevoelig voor een expressie**
het proces **hervat als *boolean_expression* waar** is of waar wordt als er geen gevoeligheidsvoorwaarde is
*[**for** time_expression]*; : het proces is **gevoelig voor de tijd** :
het proces **hervat na *timeout*** : we wachten op het tijdspunt *time_expression*

⇨ **wait**; -- wait forever

⇨ **wait until** clk = '1' : wacht tot clk 1 wordt (**hervatten op elke klokflank**)

⇨ **wait on** clk **until** reset = '0' **for** 1 ms : **wacht tot reset 0** is op een verandering van clk, maar **niet langer** dan een **simulatietijd van 1 ms**

⇨ **Proces met gevoeligheidslijst** : Een process kan een gevoeligheidslijst *(signal_name(s))* hebben als **parameter**. Het komt erop neer dat we de **sequentiële wait uit de sequentiële uitdrukkingen** halen en **algemeen** bepalen voor het process. De volgende 2 blokken code voor een proces doen dus hetzelfde, maar in de eerste blok hebben we de sequentiële wait eruit gehaald en bovenaan als parameter ingevoerd. We wachten dan gewoon telkens op de *(signal_name(s))* om het proces opnieuw uit te voeren :

```
process (signal_name(s)) [is]
  [declarations and subprograms]…
begin
  sequential_statement(s)
end process [label];
```

```
process [is]
  [declarations and subprograms]…
begin
  sequential_statement(s)
  wait on signal_name(s);
end process [label];
```

Het **verschil** tussen **variabelen** en **signalen** :

- ⇨ Een **variabele** kan **enkel** in een **subprogramma** of een **proces** gebruikt worden; als de **waarde** ervan **buiten** een proces beschikbaar moet zijn, **moet ze aan een signaal toegekend** worden. Variabelen zijn eigenlijk gewoon tussenresultaten, verbindingen tussen poorten in een bepaalde schakeling. Ze hebben geen betekenis buiten de schakeling.

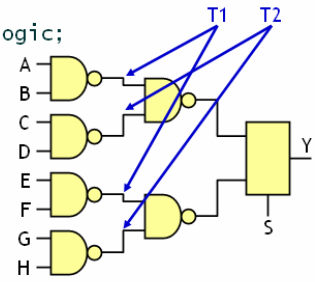
- ⇨ Een **variabele** wordt **dadelijk aangepast** wanneer zijn waarde veranderd (direct), een **signaal** wordt **aangepast** door de **eerstvolgende "wait"-uitdrukking** en men ziet een signaal dus **buiten pas veranderen** wanneer een **bepaald signaal de wait heeft geäctiveerd**. (Variabelen ziet met buiten niet). We zien dit in het **volgende stukje code** waar v een variabele is en s een signaal : v veranderd ogenblikkelijk, s gehoudt zijn waarde 0 tot na wait, ook al is zijn volgende waarde 1 reeds gedefinieert.

```
v := '1';
if v = '0' then
  -- gebeurt nooit
end if;
```

```
s <= '1';
if s = '0' then
  -- s was 0 ervoor
end if;
```

- ⇨ **Signalen** hebben **gewoonlijk** een **fysische betekenis**, **variabelen niet** noodzakelijk : in de figuur hebben variabelen T1 en T2 geen fysische betekenis omdat ze gewoon elk naar 2 verschillende verbindingen refereren waar verder niets mee gebeurt.

```
process is
  variable T1,T2: std_logic;
begin
  if (S='1') then
    T1 := A nand B;
    T2 := C nand D;
  else
    T1 := E nand F;
    T2 := G nand H;
  end if;
  Y <= T1 nand T2;
  wait on A,B,C,D,E,F,G,H,S;
end process;
```



Combinatorische logica

Parallelle signaaltoekenning is een **verkorte manier** voor **functionele modellering**. Het komt erop neer dat we een **waarde (***waveform***) willen toekennen** aan een **signaal**, en dat de **waarde** die we toekennen **afhankelijk** is van een **expressie**. We stellen dus een **uitdrukking voorop** en **afhankelijk van de uitkomst** van die uittdrukking kennen we een **bepaalde waarde** toe aan het signaal. Dit is ter **vervanging** van een **volledig process** om de **nieuwe waarde** van het signaal te **bepalen**.

- ⇨ Toekenning van conditionele signalen : de **nieuwe waarde** van het signaal is **afhankelijk** van het resultaat van een **conditionele uitdrukking** die dus als **oplossing een boolean** geeft.
  - -> Met **parallelle signaaltoekenning** is de code *:*
    name <= *[waveform1* **when** *boolean_expr* **else** *waveform2]*;
  - -> Op de **normale manier** zou dit een **hele process-beschrijving** vragen :
    process(*alle_signalen_behalve_name*)
    begin
    *[***if** *boolean_expr* **then***]* *name* <= *waveform1*;
    *[***else** *name* <= *waveform2*;*]*
    *[***end if**;*]*
    end process;
    voorbeeld : we kunnen schrijven
    y <= d1 **when** s = '1' **else** d0 **when** s = '0' **else** 'X';
    ter vervanging van de blok code links

```
process(d0,d1,s) begin
  if s = '1' then
    y <= d1;
  elsif s = '0' then
    y <= d0;
  else y <= 'X';
  end if;
end process;
```

⇨ Toekenning van geselecteerde signalen : de **nieuwe waarde** voor het signaal hangt af van de oplossing van een uitdrukking. Voor elke mogelijke oplossing van de uitdrukking zullen we een niewe waarde voor het signaal voorzien.

-> met paralelle signaaltoekenning wordt de uitdrukking :

with *expression* **select** *name* <= *[waveform1* **when** *choice(s),* w*aveform2* **when** *choice(s)]*;

-> Op de normale manier zou dit in processvorm moeten op volgende manier :

process(*alle_signalen_behalve_name*)
begin
case *expression* **is**
*[***when** *choice(s)* => *name* <= *waveform;]…*
when *choice(s)* => *name* <= *waveform*;
end case;
end process;

voorbeeld : we kunnen nu schrijven

with op **select** y <= a+b **when** addop,a-b **when** minnop;

waar dat vroeger gedaan moest worden met het proces hiernaast :

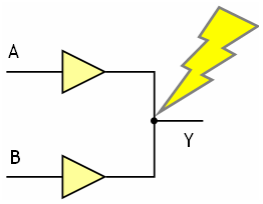
```
process(op,a,b) begin
  case op is
    when addop =>
      y <= a+b;
    when minop =>
      y <= a-b;
  end case;
end process;
```

⇨ Wat gebeurt er wanneer we **parallel meerdere verschillende waarden toekennen** aan **éénzelfde signaal**, dus m.a.w. wat is het is het effect van deze uitdrukkingen?

Y <= A;
Y <= B;

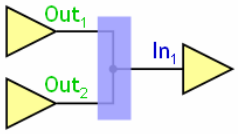
⇨ In een **process** wordt de **eerste uitdrukking genegeerd** omdat daar alle uitdrukkingen **sequentieel** uitgevoerd worden en dus de 2de waarde B de eerste overschrijft.

⇨ In de **architectuur** van een programma is deze **code ongeldig** (**compilatiefout**) vermits VHDL **slechts enkelvoudige toekenningen** toelaat. Dit is omdat we in de architectuur alle uitdrukkingen **parallel** uitvoeren, en een parallelle toekenning van een variabele tot **kortsluiting** zou kunnen leiden : bv als A='0' en B='1', wordt het actieve signaal signaal kortgesloten op het niet-actieve.



⇨ **Oplossing** voor het vorige probleem : **Omgezette ('resolved') signalen**. We voegen aan de **signaaldefinitie** een **resolutiefunctie** toe die de eigenlijke **waarde berekent** uit **alle aangelegde waarden**, om zo deze **kortsluitingen** te **vermijden**. We installeren eigenlijk een extra instantie (resolutiefunctie) die eerst alle aangelegde signalen controleert en op basis daarvan het signaal definiëert. Bv :

function resolved (s: std_ulogic_vector)
return std_ulogic;
type std_logic **is** resolved std_ulogic;



We gebruiken dus voor de **in- en outpoorten** en **argumenten** telkens een **resolutiefunctie** in plaats van het signaal zelf.

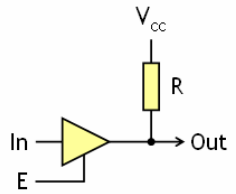
⇨ De **bibliotheek** van IEEE '**Std_logic_1164**' heeft zulke **resolutiefuncties** : Deze functie krijgt een **variabele s** (vector van signalen) **binnen** en bepaalt aan de hand van de **waarden in s één waarde std_ulogic** die de **logische combinatie** is van **alle signalen** van s is.

-> Wanneer **s maar 1 signaal** bevat is dit uiteraard het resultaat.

-> Wanneer **s meerdere signalen** bevat zullen we een variabele result definiëren. We zullen dan alle **signalen in s doorlopen** en voor elk signaal in de **resulution_table** gaan kijken. We zullen uit de tabel met de vorige waarde result en het signaal uit s dat we aan het bekijken zijn een nieuwe waarde voor result bepalen. We zullen telkens **kijken of result en het volgende signaal in conflict** zijn. Wanneer dat zo is, zal result 'X' worden (niet bepaald), anders wordt result het resultaat dat het meest logisch volgt uit de combinatie van de 2.

```
constant resolution_table :
 array (std_ulogic, std_ulogic) of std_ulogic :=
 -- 'U','X','0','1','Z','W','L','H','-'
  (('U','U','U','U','U','U','U','U','U'),  -- 'U'
   ('U','X','X','X','X','X','X','X','X'),  -- 'X'
   ('U','X','0','X','0','0','0','0','X'),  -- '0'
   ('U','X','X','1','1','1','1','1','X'),  -- '1'
   ('U','X','0','1','Z','W','L','H','X'),  -- 'Z'
   ('U','X','0','1','W','W','W','W','X'),  -- 'W'
   ('U','X','0','1','L','W','L','W','X'),  -- 'L'
   ('U','X','0','1','H','W','W','H','X'),  -- 'H'
   ('U','X','X','X','X','X','X','X','X')); -- '-'
function resolved(s : std_ulogic_vector)
         return std_ulogic is
  variable result : std_ulogic := 'Z';
begin
  if s'length = 1 then return s(s'low); end if;
  for i in s'range loop
    result := resolution_table(result, s(i));
  end loop;
  return result;
end function resolved;
```

Voorbeeld: de afgesloten bus




```
entity Pull_buf is
    port (In, E: in std_logic;
          Out: out std_logic);
  end entity Pull_buf;

entity Pullup is
    port (Out: out std_logic);
  end entity Pullup;

architecture RTL of Pullup is begin
    Out <= 'H';
  end architecture RTL;

architecture RTL of Pull_buf is
    component Driver
      port (I,E: in std_logic;
            O: out std_logic);
    end component Driver;
  begin
    component Pullup port map (Out);
    component Driver port map (In, E, Out);
  end architecture RTL;
```

resistieve driver

actieve driver

Sequentiële logica

Flip-flops in **VHDL** : VHDL heeft **geen speciale uitdrukkingen** voor **flip-flops**! Ze zijn **impliciet aanwezig** als een **signaal of variabele zijn waarde behoudt** gedurende een tijd. Dit gebeurt typisch bij een **onvolledige if of case** uitdrukking.

⇨ We kunnen nu wel een **latch beschrijven** in **VHDL** : we hebben dan een **ingang D** en een **klok Clk** nodig, en een **uitgang Q**. We moeten er voor zorgen dat wanneer Clk=1 is de waarde van Q die van D volgt. Dus **telkens de waarde van D of Clk veranderd** zullen we **checken of Clk 1** is. En wanneer dat het geval is zullen we de **waarde van D in Q stoppen**. Mogelijkheden :

-> Clk-event & Clk=0 : er gebeurt niets
-> Clk-event & Clk=1 : D wordt gekopieerd naar Q
-> D-event & Clk=1 : D wordt gekopieerd naar Q

```
process (D,Clk) is
begin
  if (Clk='1') then
    Q <= D;
  end if;
end process;
```

⇨ We kunnen ook een **latch maken die zijn uitgang Q reset op 0** telkens **de klok 0 wordt** en dus **enkel 1** kan zijn wanneer de **klok en D beide 1** zijn.

```
Mux: process(D,Clk)
begin
  if Clk = '1' then
    Q <= D;
  else Q <= '0';
  end if;
end process Mux;
```

We kunnen ook **een D-flipflop** maken die enkel op een **stijgende klokflank schakelt** (en niet zoals de vorige latch die ook schakelt wanneer D veranderd buiten de stijgende klokflank wanneer Clk=1). We kunnen dit op **2 manieren** :

⇨ Met een "**wait until**" uitdrukking. Hier **wachten** we dus telkens **tot Clk 1** wordt en steken dan D in Q.

```
DFF: process is
begin
  wait until Clk='1';
  Q <= D;
end process DFF;
```

⇨ Met een **"event"-attribuut** : we **wachten** tot er iets **gebeurt met de klok Clk** en **testen dan of de Clk 1** is. We moeten hier de **klok als ingang opgeven**. In de **bibliotheken** komt dit **event-attribuut** voor als **'rising_edge(clk)'**. Het voordeel hiervan is dat het rekening houdt met 'H', hetgeen beter is voor std_/u/logic.

```
DFF: process (Clk) is
begin
  if (Clk'event and Clk='1') then
    Q <= D;
  end if;
end process DFF;
```

We zien in de methode met het "event"-attribuut *rising_edge(clk)* een duidelijk voordeel wanneer we flip-flops met asynchrone reset willen gebruiken. We kunnen de asynchrone reset namelijk niet bouwen met het *wait-until* commando. Bij ff's met synchrone reset kan dit eventueel wel. We kunnen hen gebruiken bij het maken van registers met een combinatorische schakeling aan de ingang.

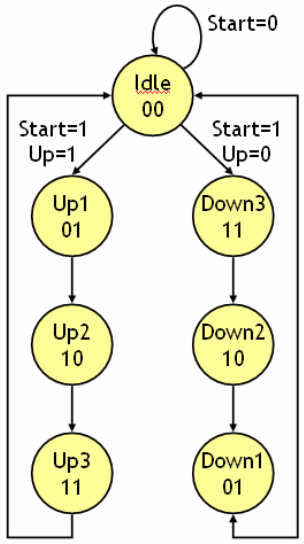
Synchrone reset

```
process(D,Clk,Rst)
begin
  if
   rising_edge(Clk)
   then
    if Rst='1' then
      Q <= '0';
    else
      Q <= D;
    end if;
  end if;
end process;
```

Asynchrone reset

```
process(D,Clk,Rst)
begin
  if Rst = '1' then
    Q <= '0';
  elsif
   rising_edge(Clk)
   then
    Q <= D;
  end if;
end process;
```

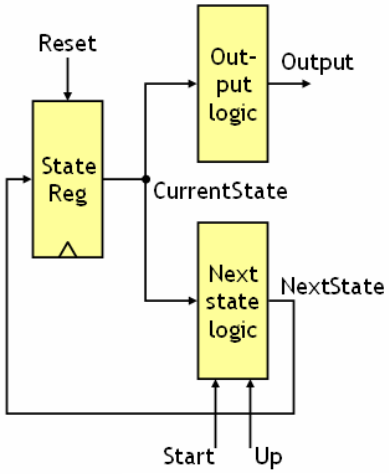
We zullen nu een **voorbeeld** van een **FSM bespreken**. We ontwikkelen zijn **hardware** in **VHDL-code**.



⇨ De **algemene code** voor de structuur van het totaal :

```
entity FSM is
  port (Start, Up, Reset, Clk: in std_logic;
        Output: out std_logic_vector(0 to 1));
end entity FSM;

architecture Behav of FSM is
  type FSM_States = (Idle,Up1,Up2,
            Up3,Down1,Down2,Down3);
  signal CurrentState, NextState :
            FSM_States;
begin
 StateRegister:
  process(NextState,Clk,Reset)
    …
  end process StateRegister;
 NextStateLogic:
  process(CurrentState,Start,Up)
    …
  end process NextStateLogic;
 OutputLogic:
  process(CurrentState)
    …
  end process OutputLogic;
end architecture Behav;
```



⇨ De code voor het register dat bijhoudt in welke toestand we zijn :

```
StateRegister:
process(NextState,Clk,Reset) is
begin
  if Reset='1' then
    CurrentState <= Idle;
  elsif rising_edge(Clk) then
    CurrentState <= NextState;
  end if;
end process StateRegister;
```

⇨ De code voor de next-state-logica :

```
NextStateLogic:
process(CurrentState,Start,Up) is
begin
  case CurrentState is
    when Idle =>
      if Start = '0' then
        NextState <= Idle;
      elsif Up = '1' then
        NextState <= Up1;
      else
        NextState <= Down3;
      end if;
    when Up1 =>
      NextState <= Up2;
    when Up2 =>
      NextState <= Up3;
    when Up3|Down1 =>
      NextState <= Idle;
    when Down3 =>
      NextState <= Down2;
    when Down2 =>
      NextState <= Down1;
  end case;
end process NextStateLogic;
```

⇨ de code voor de output-logica :

```
OutputLogic:
process(CurrentState) is
begin
  case CurrentState is
    when Idle =>
      Output <= "00";
    when Up1|Down1 =>
      Output <= "01";
    when Up2|Down2 =>
      Output <= "10";
    when Up3|Down3 =>
      Output <= "11";
  end case;
end process OutputLogic;
```

⇨ **Veilige toestanden** : Stel dat we een **toestandsmachine** hebben met **3 toestanden**, **gecodeerd in 2 bits**. We zouden dan in de **problemen** kunnen komen wanneer we door bv. ruis of bij het opstarten in de **4de (ongedefiniëerde) toestand terecht komen**.
We moeten daarom **voorzorgen** nemen in de VHDL-code : we definiëren altijd '**others**' wanneer we met **case** werken. Stel dat we een **andere** dan de **vooropgestelde oplossingen** hebben, dan is deze '**others**' en zal de **schakeling uitvoeren** wat voor **others gedefiniëert** is. We zullen zo voorkomen dat de schakeling in problemen komt, omdat we altijd een **oplossing** heeft die **we zelf geprogrammeert hebben.** Wanneer we **geen 'others'** definiëren is het **gedrag** van de schakelin **onvoorspelbaar**.

```
NextStateLogic: process(CurrentState) is
begin
  case CurrentState is
    when Idle   =>  NextState <= S1;
    when S1     =>  NextState <= S2;
    when S2     =>  NextState <= Idle;
    when others =>  NextState <= Idle;
  end case;
end process NextStateLogic;
```

2) Structurele beschrijving

Een structurele beschrijving beschrijft de hiërarchie van de componenten, als verbindingen tussen subsystemen.

Instantiatie van componenten : we bepalen het gebruik van de entiteiten en componenten. We moeten met andere woorden vanuit een beschrijving van een component effectieve instanties van die component maken door o.a. de te gebuiken variabelen in te vullen. We kunnen dit aanmaken van instanties op 2 manieren doen :

⇨ **Directe instantiatie** (niet in VHDL-87) : We **maken direct instanties aan zonder eerst een component te declareren** en daar een instantie van te maken. We kunnen nu de instanties associëren via positie of naam voor generische constanten en poorten (cfr. subprogramma's). Dit is een **impliciet 'bottom-up' ontwerp** (omgekeert ontwerp) en is dus niet geschikt voor grote ontwerpen.

```
entity entity_name [(architecture_name)]
  [generic map (generic_association(s))]
  [port map (port_association(s))];
```

voorbeeld :

```
entity work.reg(struct)
  generic map (n => 4)
  port map (D_in, Q_out, clock);
```

default T_pd

⇨ **Via Component-declaratie** : We declareren nu eerst een component en maken daar een instantie van aan. Een component is eigenlijk een virtueel element waar we effectieve instanties (reële elemanten) van kunnen maken. Dit is een 'top-down'-ontwerp dat grotere ontwerpen toelaat. We kunnen componenten uit bibliotheken gebruiken zodat we ze zelf niet meer moeten maken.

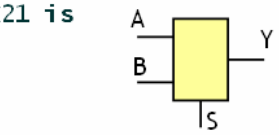
```
component name [is]
  [generic (generic_list);]
  [port (port_list);]
end component [name];
```

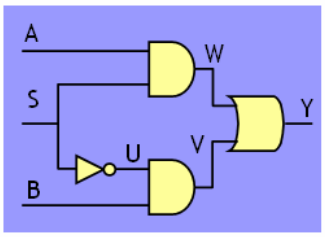
⇨ **Component-instantiatie** : het effectief aanmaken van instanties van een component. Gebeurt met volgende code :

```
[component] name
  [generic map (generic_association(s))]
  [port map (port_association(s))];
```

Voorbeeld van een **2-to-1-MUX** via eerst **declaratie** van componenten en dan **aanmaken** van **instanties** van die componenten :

```
architecture Struct of MUX21 is
  signal U,V,W : bit;
  component AND2 is
    port (X,Y: in bit;
          Z: out bit);
  end component AND2;
  component OR2 is
    port (X,Y: in bit;
          Z: out bit);
  end component OR2;
  component INV is
    port (X: in bit;
          Z: out bit);
  end component INV;
begin
  Gate1: component INV  port map (X=>S,Z=>U);
  Gate2: component AND2 port map (X=>A,Y=>S,Z=>W);
  Gate3: component AND2 port map (X=>U,Y=>B,Z=>V);
  Gate4: component OR2  port map (X=>W,Y=>V,Z=>Y);
end architecture Struct;
```





De **Configuratie** van het **programma** is dan de **koppeling** van **componenten** en **entiteiten** : We **configureren** een **entity** met een **architectuur**. Deze **architectuur** bestaat uit een **aantal componenten** (met elk een label) waar we **entiteiten** van aanmaken. Voor elke component hebben we dan een commando '**use**' dat gaat **bepalen welke entiteit(en)** we zullen gebruiken van de component. De ***use_info*** achter 'use is **ofwel een naam voor de entiteit** (*entity_name [(architecture_name)]*) **ofwel een configuratie** *(configuration_name)* waarbij we de component zelf configureren.

De ***label(s)*** is ofwel **others** ofwel **all** ofwel een door komma's gescheiden lijst van labels van componenten.

Componenten waarvoor **geen koppeling voorzien** is, worden **gekoppeld** aan entiteiten met **dezelfde naam**. We maken van die component dan **één entiteit** aan die dezelfde naam heeft als de naam van de component. **Hiërarchische ontwerpen hergebruiken architectuur-configuratie** als koppeling.

```
configuration name of entity_name is
  for architecture_name
    [for label(s): component_name
      use use_info
       [generic map (generic_association(s))]
       [port map (port_association(s))];
    end for;]...
  end for;
end [configuration] [name];
```

koppeling kan anders zijn voor elke component

Voorbeeld van de MUX21 : om deze te implementeren, willen we de implementaties van AND3, OR3 en INV uit een bibliotheek gebruiken (architectuur "RTL"). Dit doen we met de code :

```
entity AND3 is
  port (A,B,C: in bit; Y: out bit);
end entity AND3;

entity OR3 is
  port (A,B,C: in bit; Y: out bit);
end entity OR3;

entity INV is
  port (A: in bit;     Y: out bit);
end entity INV;
```

De **configuratie** wordt dan :


```
configuration Use3InputGates of MUX21 is
  for Struct
    for Gate1:INV use entity INV(RTL)
      port map (A=>X,Y=>Z);
    end for;
    for all:AND2 use entity AND3(RTL)
      port map (A=>X,B=>Y,C=>'1',Y=>Z);
    end for;
    for Gate4:OR2 use entity OR3(RTL)
      port map (A=>X,B=>Y,C=>'0',Y=>Z);
    end for;
  end for;
end Use3InputGates;
```

3) Beschrijving van repetitieve structuren

De **parallelle uitdrukking "generate"** : Wanneer we **verschillende instanties** willen maken met **dezelfde structuur**, kunnen de deze genereren met een **lus** met het **commando 'generate'**. We kunnen de lus realiseren met een for-lus of met een booleaanse expressie.

- ⇨ Het **genereren iteratieve structuren** door het **herhalen identieke cellen** : We zullen hier een **vast aantal** van dezelfde structuren maken, evenveel als er in de **range passen**. We zullen de **identifier** gebruiken om de **structuren te nummeren** of te benoemen

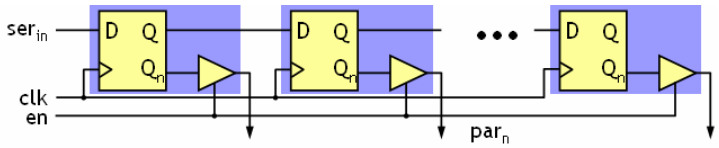
```
for identifier in range generate
  [declaration(s)
begin]
  {[label :] concurrent_statement}...
end generate [this_label];
```

geen **begin** als geen *declaration(s)* !

- ⇨ **Structuren conditioneel genereren** : we **blijven structuren aanmaken** zolang de **booleaanse expressie waar** is. Een ander voordeel is dat we hier sommige **cellen anders kunnen behandelen** dan de anderen

```
if boolean_expression generate
  [declaration(s)
begin]
  {[label :] concurrent_statement}...
end generate [this_label];
```

Voorbeeld : de 3-state-SIPO :
-> Een **sipo** is (denk ik) een soort **stapelgeheugen** dat staat voor **seriëel in, parallel out**. We hebben **1 ingang** die in de eerst ff binnenkomt. Op elke klok **schuiven we de waarde in elke ff door** naar die die er achter komt, en de **eerste krijgt de waarde van de ingang**. **Parallel out** betekent dat we de waarde van **elke ff kunnen raadplegen**.
-> We heben hier een **generische parameter n** die **vastlegt hoeveel ff's** we hier achtereen zetten. Op **basis van n** hebben we dan **ook n uitgangen**, en **3 ingangen** (Clk, enable en Seriële ingang).
-> we zullen nu **in de code n ff's aanmaken**. Hiervoor kunnen we zeer goed gebruiken maken van het **commando generate**, waar we **simpelweg met een for-lus n identieke structuren** zullen aanmaken die als ingang telkens de uitgang van de vorige hebben, hun uitgang doorgeven en hu nandere uitgang als ide uitgang van het systeem stellen met een 3state-buffer. Dit is veel beter dan de code die hieronder staat :




```
library ieee; use ieee.std_logic_1164.all;

entity ser2parinv is
  generic(n: positive);
  port(clk, en, ser_in: in std_logic;
       par_n: out std_logic_vector(1 to n));
end entity ser2parinv;

architecture cells of ser2parinv is
  component DFF is
    port(clk, D: in std_logic;
         Q, Qn: out std_logic);
  end component;
  component tristate is
    port(en, A: in std_logic; Y: out std_logic)
  end component;
  signal state: std_logic_vector(1 to n));

  begin
    cell_array: for index in 1 to n generate
      signal prebuf: std_logic;
    begin
      first_FF: if index = 1 generate
        FF: component DFF
          port map(clk, ser_in,
                   state(index), prebuf);
        end generate first_FF;
      other_FF: if index > 1 generate
        FF: component DFF
          port map(clk, state(index-1),
                   state(index), prebuf);
        end generate other_FF;
      buf: component tristate
        port map(en, prebuf, par_n(index));
   end generate cell_array;
  end architecture cells;
```

```
configuration struct of ser2parinv is
  for cells                  -- architecture
    for cell_array(n)        -- last FF
      for other_FF
        for FF:DFF use entity …; end for;
      end for;
      for buf:tristate use entity …; end for;
    end for;
    for cell_array(1 to n-1) -- other cells
      for first_FF     -- 1st inner generate
        for FF:DFF …; end for; end for;
      for other_FF     -- 2nd inner generate
        for FF:DFF …; end for; end for;
      for buf:tristate use entity …; end for;
    end for;
  end for;
end configuration struct;
```

We zien dat we voor de 1ste ff een iets ander moeten doen dan voor de anderen : conditioneel

Hardware-simulatie met VHDL

1) Gebeurtenisgedreven simulatie

Een **groot voordeel** van VHDL is dat we eens dat we ons **ontwerp hebben ingegeven**, we het **virtueel kunnen testen in de tijd** en zijn **werking kunnen simuleren**. Hierdoor kunnen we **onvoorziene fouten** ontdekken en corrigeren. We zullen hierbij **signalen simuleren** en kijken hoe de **schakeling daarop regeert.**

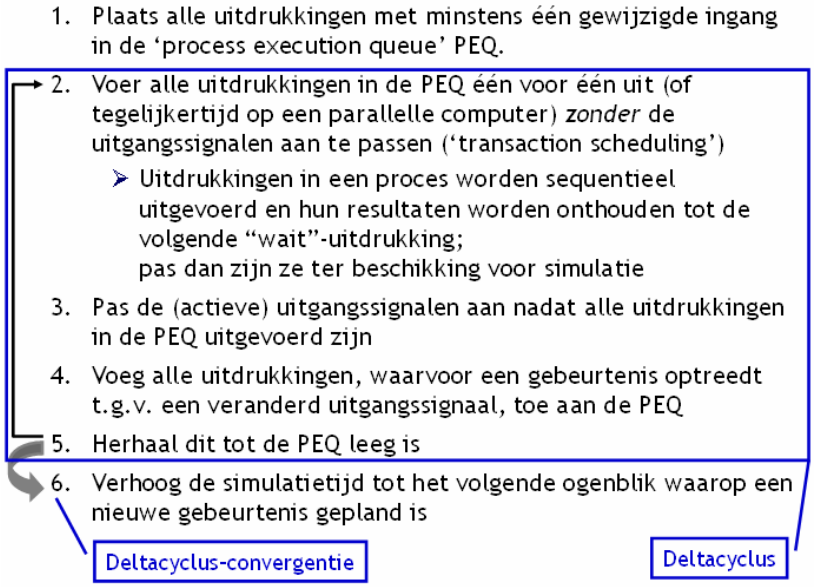
⇨ We zouden dit kunnen doen door **signalen aan te leggen** en **continu de uitgangen te bereken** (bijv. per fs of een andere zeer klein tijdsinterval). Dit is echter **nodeloos veel rekenwerk** aangezien in veel van die intervallen de uitgang niet verandert.

⇨ We kunnen dat oplossen met **'event-driven' simulatie** :

- ⇨ Wanneer we een **signaal een nieuwe waarde toekennen** in de code creëert dit een **transactie** (het signaal zal dan een nieuwe waarde krijgen op het volgende simulatietijdstip)
- ⇨ Wanneer de **simulatietijd** dan **voortgaat** naar het nieuwe simulatietijdstip krijgt **het singaal dan effectief zijn nieuwe waarde** (signaal is actief tijdens deze deltacyclus)
- ⇨ Deze toekenning van de nieuwe waarde zal pas een **gebeurtenis ('event') veroorzaken** wanneer de **nieuwe waarde verschilt van de oude**. (dus als er netto verandering optreed in het circuit)
- ⇨ Alle **uitdrukkingen parallel** met die van het event hebben nu een **gevoeligheidslijst** met bepaalde **variabelen**. Alleen die die dat **signaal op hun lijst hebben** worden **opnieuw berekent** wanneer er zich een event voordoet.

Dit **mechanisme** zorgt er enkel voor dat de **simulatie versnelt zonder het gesimuleerde gedrag te wijzigen**. Dit is dus een manier om de hoeveelheid rekenwerk te verminderen zonder dat het gesimuleerde gedrag daardoor wijzigt.

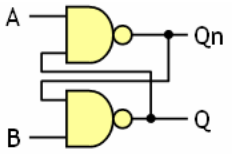
⇨ Implementatie van de simulator : We plannen op voorhand wanneer we welke ingangen zullen veranderen. Telkens we een ingang veranderen zullen we dan het volgende porgramma doorlopen en zo de uitgangen berekenen als reactie op de ingangsverandering. Dit programma noemen we de Delta-cyclus. Wanneer we deze cyclus afgewerkt hebben verhogen we de simulatietijd tot de volgende verandering van de ingangen.Dus per gewijzigde ingang doen we :



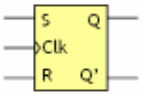
Voorbeeld : we zullen nu de simulatie van een SR-FF (set-reset flip-flop) bespreken.
-> de **set-reset ff** heeft **2 ingangen** set A en reset B en **2 uitgangen** die elkaars inverse zijn. De **uitgang Q** blijft behouden wanneer Set en reset zijn, kan veranderen op de klok naar 1 als Set=1 of naar 0 als Reset=1.
-> we realiseren dit met 2 NAND-poorten.

```
entity Flipflop is
  port (A,B: in std_logic;
        Q,Qn: buffer std_logic);
end entity Flipflop;

architecture Struct of FlipFlop is
begin
  NAND2: entity NAND2 port map (Qn,B,Q);
  NAND1: entity NAND2 port map (A,Q,Qn);
end architecture Struct;
```



SR flip-flop



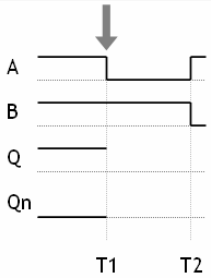
Heeft 2 ingangen S set en R reset. Voor S=1 wordt Qnext 1, voor R=1 wordt Qnext 0, voor beide 0 blijft Q behouden. Beide 1 komt nooit voor.

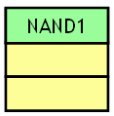
S	R	Q(next)
0	0	Q
0	1	0
1	0	1
1	1	NA

Q	Q(next)	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

⇨ We zullen nu de **delta-cyclus** voor de **simulatie** van de code van de SRff **opstellen 2 simulatietijdstippen** uitwerken :
- T1 en T2 zijn de **2 simulatie-tijdstippen** waarop er ingangen veranderen. Beide zijn aanvankelijk 1, op T1 wordt A=0 en op T2 veranderen ze beide.

⇨ **Voor T1 :**

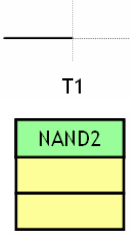


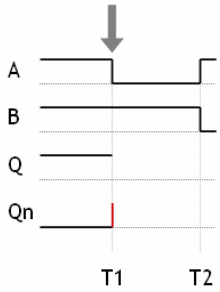


1. Plaats uitdrukkingen met ingangsgebeurtenissen in PEQ
   Hier reageert enkel NAND 1 direct op verandering in A

Deltacyclus 1

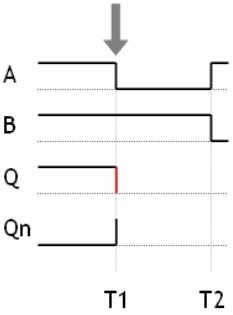
2. Voer uitdrukkingen in PEQ uit en onthoud uitgangen
3. Pas uitgangen aan : We zien dat Qn veranderd van 0->1
4. Voeg uitdrukkingen met gebeurtenissen toe aan PEQ : We zien dat NAND2 zal veranderen door de verandering van Qn





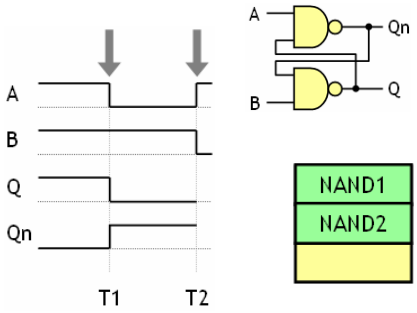
Deltacyclus 2

2. Voer uitdrukkingen in PEQ uit en onthoud uitgangen
3. Pas uitgangen aan : We zien dat Q verander van 1->0
4. Voeg uitdrukkingen met gebeurtenissen toe aan PEQ : We zien dat NAND1 zou kunnen veranderen door Q



Deltacyclus 3

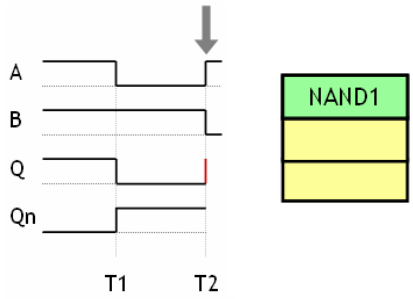
2. Voer uitdrukkingen in PEQ uit en onthoud uitgangen
3. Pas uitgangen aan : We zien dat het veranderen van Q geen effect meer op NAND1 en we moeten dus geen uitgangen meer moeten aanpassen. Er zullen dus ook geen componenten meer zijn die zullen veranderen en dus geen nieuwe uitdrukkingen voor de PEQ
4. Geen uitdrukkingen voor PEQ: T1 deltacyclus-convergentie : We gaan naar T2



⇨ **Voor T1 :** We zien dat beide A en B van waarde veranderen

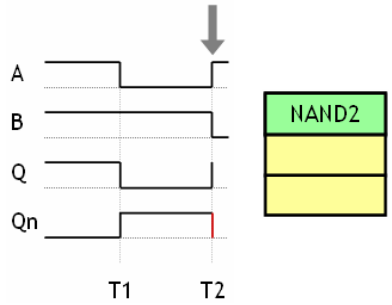
1. Plaats uitdrukkingen met ingangsgebeurtenissen in PEQ : Daar A en B beide veranderen zullen ok beide NAND1 en NAND2 veranderen.

Deltacyclus 1



2. Voer uitdrukkingen in PEQ uit en onthoud uitgangen : We zien dat B op 0 valt en dat heeft direct invloed op Q die naar 1 stijgt. A valt ook, maar moet op de reactie van Q wachten om ook Qn te veranderen, dus voorlopig verandert enkel Q en niet Qn
3. Pas uitgangen aan : We zien dat Q veranderd van 0->1
4. Voeg uitdrukkingen met gebeurtenissen toe aan PEQ : We zien dat door de verandering van Q NAND1 zal veranderen

Deltacyclus 2



2. Voer uitdrukkingen in PEQ uit en onthoud uitgangen
3. Pas uitgangen aan : We zien dat Qn verandert van 1->0
4. Voeg uitdrukkingen met gebeurtenissen toe aan PEQ : een verandering van Qn zou invloed kunnen hebben op NAND2

Deltacyclus 3

2. Voer uitdrukkingen in PEQ uit en onthoud uitgangen
3. Pas uitgangen aan : We zien dat het veranderen van Qn geen effect meer op NAND2 en we moeten dus geen uitgangen meer moeten aanpassen. Er zullen dus ook geen componenten meer zijn die zullen veranderen en dus geen nieuwe uitdrukkingen voor de PEQ
4. Geen uitdrukkingen voor PEQ: T2 deltacyclus-convergentie : We gaan naar T3 enz.

2) Beschrijving tijdsgedrag

Wanneer we het **tijdsgedrag** van een **schakeling** willen **onderzoeken**, zullen we op de **signalen golfvorm ('waveform')** moeten **aanbrengen** om zo de **reactie** van de schakeling in zijn **uitgangen** te bekijken. De **golfvormen** worden dus **toegekend** wordt aan een **signaal**. We zullen deze golfvormen maken door **transacties** te **plannen** op een **constant signaal** (we zullen een constant signaal (1 of 0) producteren dat op **vooropgestelde tijdstippen** zal **veranderen van 1 naar 0 of omgekeert**).

⇨ Deze **transacties** worden **gepland** met waarde een waarde ***expression*** op een **tijdsstip** = ***a_time*** + **huidige simulatietijd** (default *a_time* = 0 fs). We zullen dus wachten tot a_time na de systeemtijd om een vooropgestelde waarde toe te kennen aan een signaal.

```
[delay_mechanism] expression [after a_time]
    [, expression [after a_time]]...
```

⇨ ***delay_mechanism*** geldt **enkel voor het eerste element**; de andere hebben altijd een transportvertraging (tijd die het duurt voor de toekenning die ervoor komen)

```
-- NAND-poort met 10 ns vertraging
y <= a nand b after 10 ns;
-- 20 ns brede resetpuls na 5 ns
rst <= '1' after 5 ns, '0' after 25 ns;
```

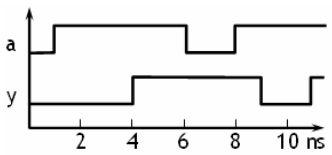
⇨ **Vertragingsmechanismen** :

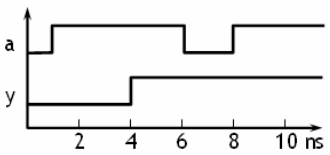
-> de **transportvertraging** '*transport'* is de **tijd** die het duurt **om van de ingang naar de uitgang te geraken** door de schakeling, dus de tijd nodig om te schakelen (kritisch pad)

-> **inertievertraging** '*[*[**reject** *reject time*] **inertial**]' is de **vertraging** ten gevolge van de **inertie** van het systeem t.g.v. de **capaciteit en inductantie** van de verbindingen : We zullen dus een **inertietijd instellen** zodat de **variaties** op de pulsen door de spaciteit en inductantie op de verbindigen te tijd krijgen om te **verdwijnen**

voorbeeld : wanneer we een **golfvorm a** willen **toekennen** aan een **signaal y** :

- wanneer we **transport** toepassen zien we dat **y telkens 3 seconden na a veranderd**. We zien dat dan **elke verandering in a wel doorgegeven** wordt
- wanneer we **geen transport** gebruiken en **gewoon after 3 ns gebruiken**, zien we dat **y met a verander**, maar dat **dat niet meer het geval** is wanneer een verandering **in a minder dan 3ns duurt**. Dit komt door de **inertie** van het systeem. A zal op minder dna 3 ns niet voldoende gedaalt zijn om y mee te kunnen laten dalen voor a weer stijgt.





3) Testbank

We **testen** het **ontwerp** van een schakeling op **logische correctheid** door aan de **ingangen representatieve stimuli** aan te leggen en te **controleren** of de **uitgangen** de **correcte waarden** op het **juiste ogenblik** vertonen.

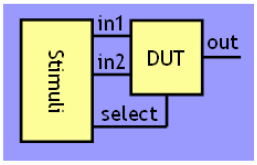
⇨ We gebruiken daarvoor een **VHDL 'testbank'** op het **hoogste hiërarchisch niveau** van een ontwerp. Deze creëert dan een **instantie** van het '**Design Under Test**', **voert stimuli** toe aan de ingangen van DUT, **controleert** de **uitgangen** ervan
door ze te **analyseren**, (bijv. "assertion"- of "report"-uitdrukkingen)
of door ze **als golfvorm te visualiseren**.
Vermits dit het **hoogste niveau** is, heeft de test-instantie **zelf geen ingangen of uitgangen**!

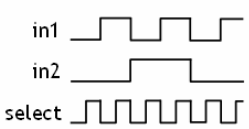
Voorbeeld van een MUX-testbank :

```
entity Testbench is
end entity Testbench;

architecture BehavTest of Testbench is
  signal in1, in2, select, out : bit;
begin
  DUT: entity MUX21(Behav)
    port map (in1,in2,select,out);
  Stimuli: process is
    begin
      in1 <= '0', '1' after 50 ns,
             '0' after 100 ns, '1' after 150 ns;
      in2 <= '0', '1' after 100 ns;
      for i in 1 to 4 loop
        select <= '0', '1' after 25 ns;
        wait for 50 ns;
      end loop;
    end process Stimuli;
end architecture BehavTest;
```

Geen in/uitgangen aan Testbench





Hardware-synthese met VHDL

1) Synthetiseerbare VHDL

Wanneer we **VHDL-code schrijven** is het **uiteindelijke doel** om deze zo optimaal mogelijk **om te zetten naar hardware**. Daarvoor heeft elk **VDHL-programma** een **syntheseprogramma** ('hardware compiler') die de **VHDL-beschrijving omzet** in een **structurele beschrijving** op **lager niveau** (poorten/cellen). Op **RTL-niveau** is deze **omzetting goed gesupporteerd** (comilers kunnen goed code op RTL-niveau omzetten tot poorten), maar synthese van **hoger niveau** is (nog altijd?) **te complex** voor de meeste programma's.

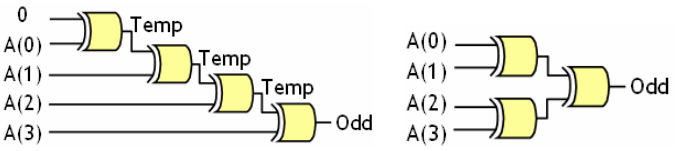
⇨ De **VDHL-programma's** verschillen in de **subsets van VHDL die ze aankunnen**. Elk programma kan namelijk **niet alle VHDL-code** omzetten.

bv. IEEE 1076.6 is standaard voor VHDL RTL-synthese = grootste gemene deler,
bijv. in de 1999-versie is alleen VHDL-87 toegelaten

⇨ De **programma's genereren** bij elk **ontwerp informatie** i.v.m. **tijdsgedrag**. Ze maken daarvoor **automatisch zelf testprogramma's** die de **reële werking nabootsen**. De reële **vertragingen** worden **ingecalculeerd** met :

-> Een *waveform* kan enkel een *expression* zijn:
delay_mechanism of **after** is niet toegelaten
-> Een **wait for** wordt genegeerd

⇨ De **compilers** zullen ook **zoveel mogelijk synthese toepassen** op de code om zo het **aantal poorten te verminderen**. Wanneer we bv. hardware willen ontwikkelen voor een **chip die ingangen vergelijkt**. Hij moet **0 geven** wanneer **alle ingangen gelijk** zijn, en **1 als minstens 1 van de ingangen verschilt** van de anderen. We **beschrijven** dit in de volgende programma-code.
Wanneer dit omzetten naar een ontwerp op poortniveau kunnen we een **waterval van poorten** krijgen. Dit kan echter veel **efficiënter**. We verwachten van de compiler dat hij dit soort vereenvoudigingen zelf zal uitvoeren.



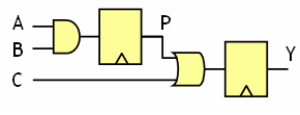

```
entity Parity is
  generic (n : integer);
  port (A: in std_logic_vector (0 to n-1);
        Odd: out std_logic);
end entity Parity;

architecture Struct of Parity is
begin
  Parity: process(A) is
    variable Temp: std_logic;
  begin
    Temp := '0';
    for I in A'low to A'high loop
      Temp := Temp xor A(I);
    end loop;
    Odd <= Temp;
  end process Parity;
end architecture Struct;
```

⇨ Een **sequentieel voorbeeld** waar de **compiler** niet voor het **meest optimale** zal kiezen en waar we dus **zelf verantwoodelijk** zijn voor het zoeken van de **optimale keuze**. In dit sequentiële voorbeeld moeten we bv. opgeletten voor verschil signaal/variabele, want als we hier teveel signalen gebruiken hebben we een veel duurdere schakeling :

-> In dit geval hebben we **P als signaal** gedefiniëert, **hetgeen niet de bedoeling** is (niet optimaal is). We hebben hier meerbepaald een **ff teveel**, en P **hoeft helemaal geen signaal** te zijn want we gebruiken hem **alleen intern** in het proces. Een extra probleem is dat er een **extra vertraging** op P zit (1 klokcyclus)

```
process (clk) is
begin
  if rising_edge(clk) then
    P <= A and B;
    Y <= P or C;
  end if;
end process;
```

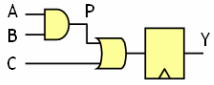


-> We kunnen dus **beter van P een variabele** maken. P wordt dan direct doorgegeven aan de OR-poort en zo is ook de vertaging minder.

```
process (clk) is
  variable P: std_logic;
begin
  if rising_edge(clk) then
    P := A and B;
    Y <= P or C;
  end if;
end process;
```

≡

```
Y <= (A and B) or C;
```



⇨ **Toegelaten datatypes** bij de **synthese** naar **hardware** : we hebben **alleen verbindingen** in hardware die **ofwel 0 ofwel 1** kunnen zijn. We hebben dus alleen types die daarop neekomen :

bit, boolean, std_/u/logic

Wanneer we **meerdere bits samen gebruiken** in de vorm van **meerdere parallelle verbindingen** kunnen we groepen van hardware bits vormen :

-> **integer & subtypes** : een integer kan voorgestelt worden als een **groep bits**

bv. **type** addr **is range** -64 **to** 63; is een getal dat met gebruik van 2's complement een groep van 7 bits vormt

-> **opsommingen** (ook als gedefinieerd door gebruiker) waarbij we dus een **variabele een eindige verzameling van waarden** kunnen geven die we **zelf vooropstellen** : we moeten dan gewoon de **waarden** in die verzameling **coderen** tot een **bitcode** om deze voor te stellen in hardware. Deze **codering** kan **verschillen** van programma tot programma

bv. **enum_encoding attribuut**: waar we de verschillende toestanden van een FMS automatisch coderen in volgorde

bv. **expliciete codering** van elke waarde:

constant reset: bit_vector := "000"; …

constant output: bit_vector := "110";

We kunnen uiteraard ook **vectoren** van bovenstaande voorstellen wanneer we ze **groeperen**.

⇨ **Verdere beperkingen** die zich voordoen **wanneer we in hardware werken** en waar we dus rekening mee moeten houden bij de **omzetting** :

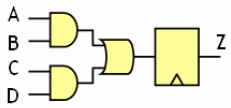
- ⇨ We kunnen **niet alle waarden** uit de std_logic **gebruiken**; enkel deze : **'1'of'H', '0'of'L', 'Z'**
  Waarbij we 'Z' genereren met een 3-state buffer Y <= A **when** Enable **else** 'Z';
- ⇨ We kunnen **geen initiële waarde** voor signalen toekennen bij het opstarten
- ⇨ **Enkel "for"-lussen** zijn toegelaten: om een lus te kunnen ontvouwen moet aantal iteraties vooraf bij het ontwerp gekend zijn. **Voorwaardelijke lussen kunnen niet gemaakt worden**.
- ⇨ Bij **sequentiële schakelingen** :
  -> **Flankgevoelige synchrone schakelingen** kunnen enkel gebruikt worden in de twee vormen die hierna besproken worden. Ander gebruik van "wait" is niet toegelaten!
  -> **Niveaugevoelige synchrone schakelingen** zijn minder gesupporteerd
  -> **Asynchrone schakelingen** kunnen niet gemaakt worden (niet gesupporteerd)
  er bestaan enkel synchrone schakelingen

⇨ We kunnen slechts op **2 manieren** een **flankgevoelige synchrone schakeling** maken :

1. Met een "wait until" uitdrukking

```
entity RegisteredCircuit is
  port (A,B,C,D: in std_logic;
        Clk: in std_logic;
        Z: out std_logic);
end entity RegisteredCircuit;

architecture RTL of RegisteredCircuit is
begin
  process is
  begin
    wait until Clk='1';
    -- combinatorische
    -- schakeling:
    Z <= (A and B) or (C and D);
  end process;
end architecture RTL;
```

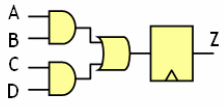


'Wait until' moet de eerste lijn van het proces zijn, gevolgd door de beschrijving van de combinatorische schakeling

2. Met een "event"-attribuut

```
entity RegisteredCircuit is
  port (A,B,C,D: in std_logic;
        Clk,Rst: in std_logic;
        Z: out std_logic);
end entity RegisteredCircuit;

architecture RTL of RegisteredCircuit is
begin
  process (A,B,C,D,Clk,Rst) is
  begin
    if Rst = '1' then
      Z <= '0';
    elsif (Clk'event and Clk='1') then
      -- combinatorial circuit
      Z <= (A and B) or (C and D);
    end if;
  end process;
end architecture RTL;
```



if is de enige uitdrukking in het proces; er is ook geen else

De test op 'Clk'event' is altijd de laatste

2) VHDL-synthese verbeteren

Een **VHDL-code herschrijven** kan het **resultaat** na synthese sterk **beïnvloeden** :

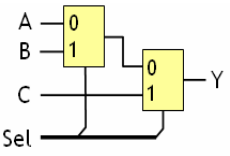
- ⇨ Een synthese-programma kan **alleen maar proberen** te **begrijpen** wat met de code **bedoeld** werd: het weet niet wat essentieel is en wat een gevolg is van de schrijfstijl?
  Het progamma wee niet of het bv een FF i.p.v. een latch mag gebruiken?
- ⇨ Een programma kan zich **niet bewust zijn** van **alle mogelijke implementaties**. Het is onmogelijk om alle mogelijke manieren van impelementeren af te gaan.
  Bv. We kunnen best zelf op zoek gana naar de meest optimale toestandscodering
- ⇨ Een programma **kan niet alle reële beperkingen in rekening brengen**. Een schakeling kan door vele factoren anders reageren dat door de compiler niet voorzien kan worden
  Vermogen, grootte, fan-out, tijdsgedrag (slechts te schatten), …
- ⇨ De **auteur** kan **verkeerde veronderstellingen** maken i.v.m. de beschikbare hardware
  Bv. Gebruik van een asynchrone set hardware die niet aanwezig is

De **mogelijkheden** van het **syntheseprogramma *en*** de **schrijfstijl** van de **code** bepalen het uiteindelijke **resultaat**!

⇨ Een **voorbeeld** waar een **verschil** in **geschreven code** resulteert in een **verschillend hardware-ontwerp**, waarbij het van de **interpretatie** van het **programma afhangt** wet de **beste optie** is : Wanneer we aan de hand van een 2bit Select-signaal een keuze moeten maken tussen 3 ingangen, kunnen we dit op 2 manieren doen :

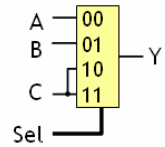
- ⇨ We kunnen dit doen met een **"if"-constructie** of met **toekenning** van **conditionele signalen** : deze methode heeft een **ingebouwde prioriteit** die **al dan niet gewenst** kan zijn. We hebben hier **twee 2-1MUXen** nodig;

```
Y <= C when Sel[1]='1' else
     B when Sel[0]='1' else
     A;
```



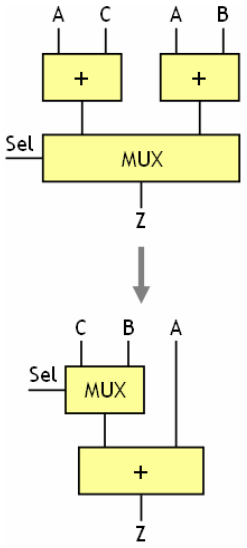
- ⇨ We kunnen ook een "**case"-uitdrukking** of **toekenning van geselecteerde signalen** gebruiken, hetgeen **meestal resulteert** in **eenvoudigere hardware**, maar waar de **prioriteit wegvalt.**

```
with Sel select
  Y <= A when "00",
       B when "01",
       C when others;
```



⇨ Een **ander voorbeeld** is '**Resource sharing**' waar we een **bepaalde bron** voor **verschillende bewerkingen** gebruiken. Stel dat we een getal A willen optellen bij een ander getal. Dat ander getal is B of C, en we hebben een selectie-ingang om te kiezen tussen beide. We kunnen dit uitvoeren met het volgende stukje code :

```
if Sel = '1' then
  Z <= A + B;
else
  Z <= A + C;
end if;
```



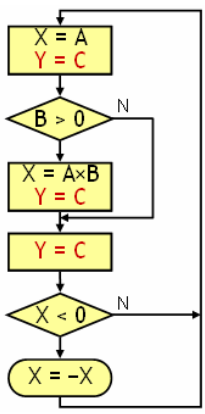
Dit **lijkt zeer goed**, maar in **hardware zelf** zien we dat deze manier van **implementatie niet optimaal is**. De meeste **compilers** zullen dit (enkel binnen éénzelfde proces) **omvormen tot de volgende code**. Dit lijkt helemaal niet beter in code, maar is wel **veel beter in hardware**. We hebben namelijk een opteller minder nodig hetgeen resulteert in een ofwel goedkopere ofwel snellere schakeling.

```
if Sel = '1' then
  X := B;
else
  X := C;
end if;
Z <= A + X;
```

3) Vertaling naar een ASM-kaart

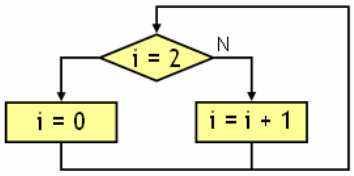
één ASM-blok komt overeen met **één toestand**.
één toestand komt overeen met **alles wat in een proces gebeurt** tussen **twee opeenvolgende "wait until"s**.
Elke toestand bevat **iets van alle processen** die in **dezelfde klokcyclus** actief zijn.

```
process begin
  X <= A;
  if B > 0 then
    wait until Clk='1';
    X <= A*B; end if;
  wait until Clk='1';
  if X < 0 then
    X <= -X; end if;
  wait until Clk='1';
  end process;
process begin
  wait until Clk='1';
  Y <= C; end process;
```



Een **ASM-variabele** is een **register** en is in VHDL een **variabele *of* een signaal**, waarvan de **waarde langer dan 1 klokcyclus bewaard** blijft. Het **verschil** tussen een signaal en een variabele is dat de **variabele** enkel **beschikbaar is** en **gebruikt wordt** in het **proces** zelf als een **soort tussenresultaat**. (zie hiernaast)
Alles wat uit het process als resultaat moet komen is een signaal :

```
process begin
  wait until clk='1';
  i := i + 1;
  if i = 3 then
    i := 0; end if;
  end process;
```



4) Synthese-aspecten voor Xilinx: overdraagbaarheid ↔ performantie

Xilinx is zo een **synthese-programma** dat VHDL-code kan omzetten tot een hardware-ontwerp.
Sommige **beperkingen** van Xilinx zijn **gekend**, **andere** kunnen **vastgelegd** worden door de **gebruiker** :

- ⇨ **Automatische synthese** van o.a. **klokbuffers** om de fan-out van het kolksignaal groter te maken
- ⇨ Dikwijls is de **default-codering one-hot** omdat dit overeen komt met de **CLB-structuur**; maar de **codering** kan in VHDL ook **aangegeven worden met het attribuut enum_encoding**, waar de **gebruiker** dus **zelf de codering** van de toestanden kan **kiezen**

Extra componenten waar Xilinx over beschikt : Extra hardware (vermenigvuldiger, …), LogiCORE-modules, inclusief RAM en I/O-buffer, eventueel met pull-up/down weerstand

Begrippenlijst

Lijst van te kennen woorden.

p6	specificatie	p17	positieve logica
	interface		negatieve logica
	blokschema / flowchart		actief laag
	implementatie / realisatie		actief hoog
	synthese		afgesloten verbindingen
			MOS-transistor
p7	rtl-niveau		gate / basis
	documentatie		source / collector
	analyse		drain / emittor
	capaciteit		
	throughpunt	p18	NMOS-transistor
			PMOS-transistor
p8	getal		CMOS-technologie
	cijfer		tors
	positieel numeriek systeem		relatieve vertraagtijd
	radix		
	radixpunt	p19	Buffer / driver
	MSB		PUN
	LSB		PDN
	radix-conversie		fan-in
p9	carrey	p20	AOI / OAI
	borrow		procesvariaties
	Sign-magnitude		omgevingsvariaties
p10	twee-complement	p21	ruismarge
			onlogische spanningen
p11	overflow		transferfunctie
	fractionele getallen		
		p22	Schmitt- trigger ingang
p12	excess-code		hysteresis
	bias		stijgtijd
	mantisse		daaltijd
	IEEE-formaat		
	enkelvoudige precisie	p23	uitgangsimpedantie
	dubbelvoudige precisie		ingangsimpedantie
	floating point overflow		algemene vertragingstijd
p13	BCD-code	p24	fan-out
	axioma		stroomgedreven technologieën
			spanningsgedreven technologieën
p14	dualiteit		statische toestand
	waarheidstabel		maximale schakelfrequentie

ALFABETISCH

aritmetisch schuiven	decompositie
asm stroomschema	directe adressering
asm-chart	directe verbindingen
assembleertaal	documentatie
asynchrone reset	don't care conditie
asynchrone schakeling	drain / emittor
asynchrone set	dualiteit
asynchrone teller	dubbelvoudige precisie
axioma	duty cycle
barrle rotator	dynamisch geheugen
basis	edge-triggered fliplop
BCD teller	enkelvoudige precisie
BCD-code	equivalente toestanden
bewerkings commando's	essentiële priemimplicanten
bias	essentiële hazard
bidirecionele teller	excess-code
binair schakelen	exhaustieve minimalisering
bi-stabiel element	exitatietabel ff
blokschema / flowchart	FA
borrow	falling edge
Buffer / driver	fan-in
bus	fan-out
busverbindingen	FDM
canonische vorm	FIFO
capaciteit	flankgetriggered
carrey	floating point overflow
carry-lookahead-opteller	FPGA
chaining	fractionele getallen
cijfer	FSMD
CISC	FU sharing
CLA	functionele eenheid / FU
CLB	fundamentele modus
clear	gate / basis
clock skew	gate array
CMOS-technologie	geheugenoperatie
codewoord	geïndexeerde adressering
compatibele toestanden	geklokte D-latch
geklokte SR-latch	maximale schakelfrequentie
gekoppelde optimalisering	maxterm
generische instructiecyclus	mealy-type
getal	metastabilitiet
glitch	minimale and or realisatie
globale kloklijnen	minimale dekking
Gray-code	minimale or and realisatie
gulzige strategie	minimum bit change
HA	minimum pulsbreedte
HAS	minterm
houdtijd	moore-type
hysteresis	MOS-transistor
I/O blokken	MSB
IEEE-formaat	multicycling
implementatie / realisatie	multiplexer / MUX

implementatie technologieën	negatieve logica
implicatietabel	next-state logic
impliciete adressering	N-kubus
incompatibiliteitsranden	NMOS-transistor
indirecte adressering	offset
ingangsimpedantie	omgevingsvariaties
ingangsequentie	one hot
initialiseren	onlogische spanningen
inputgebaseerd model	onmiddellijke adressering
instructie	opcode
instructieregister	open drain / open collector
instructieset	operand
instructietype	operaties
instructiewoorden	optimalisatie
instuctieset stroomschema	optimalisering vd vertraging
interface	output logic
JK ff	outputgebaseerd model
karakteristieke tabel ff	overflow
Karnaugh-kaart	PAL
klok width	parallelle vermenigvuldigers
klokfrequentie	patstelling / deadlock
klokperiode	PDN
knopen / nodes	pipelining
kritisch pad	PLA
kritisch pad	placement & routing
lange lijnen	PLD
latency time	PMOS-transistor
level sensitive latch	pop
levensduur	POS
LIFO	positieel numeriek systeem
linkerrand-algoritme	positieve logica
logisch schuiven	preset
looptijd	priemimplicanten
LSB	prioriteitsencoder
LUT	prioriteitsranden
mantisse	procesvariaties
marginale triggering	program counter
master-slave filipflop	PROM
max cut algoritme	PUN
Quine-McCluskey	push
race	straightforward
radix	stroomgedreven technologieën
radix-conversie	subroutine
RAM	synchrone schakeling
read adress	synthese
redixpunt	T ff
register	technology mapping
register port sharing	terugkeertoestand
register sharing	three-state-buffer
registerbank	throughpunt
registerbewerking	tijdelijk geheugen
registerinstructies	toestands-actie tabel

registertoegangstabel	toestandscodering
relatieve adressering	toestandsdiagram
relatieve adressering	toestandsregister
relatieve vertraagtijd	toestandstabel
responsie	tors
RFC	transferfunctie
ripple-carry-opteller	transitietabel
RISC	transparante latch
rising edge	TTL
ROM	twee-complement
roteren	uitgangsimpedantie
routing channels	uitgangssequentie
rtl-niveau	uitvoeringssnelheid
ruismarge	variable
Schmitt- trigger ingang	velden
schuifoperatie	verbindingen
schuifregister	vergtragings-karakteristieken
sea of gates	vermogenverbruik
selector	verplaatsingsinstructies
sequentiële machine	verwerkingskracht
sequentiële schakeling	waarheidstabel
set-up tijd	wired AND
Sign-magnitude	wired OR
SM	wired-logic
SOP	
source / collector	
spanningsgedreven technologieën	
specificatie	
sprong	
spronginstructies	
SR ff	
SRAM	
SR-latch	
standaard vorm	
standaard-code	
stapelgeheugen	
state box	
statisch geheugen	
statische toestand	
statische/dynamische hazard	
stijgtijd	

VHDL Woordenlijst

De belangrijkste woorden van VHDL.

	package
	library
p145	interface van een module
	generische parameters/constante
	implementatie van een module
p146	parallelle uitdrukkingen
	proces met gevoeligheidslijst

ALFABETISCH

acces type	VHDL-type
array literal	
attributen	
basisspecificatie	
bitreeks	
case	
commentaar	
component declaratie	
component instantie	
conditionele uitdrukking	
constante	
datatypes	
declaratietype	
directe instantie	
entiteit	
enumeration type	
event driven simulatie	
functie	
functionele modelling	
fysisch type	
generische parameters/ constante	
getal	
glijktijdigheid	
identifier	
if	
implementatie van een module	
inertievergtraging	
interface van een module	
karakter	
komma type	
library	
overloading	
package	
parallelle uitdrukkingen	
procedure	
proces met gevoeligheidslijst	
resolved signalen	
resource sharing	
samangestelde type	
samengesteld type	

scalaire type
signaal
slice
subprogramma
synthetiseerbare VHDL
testbank
transportvertraging
variabele
veilige toestand
VHDL object